COMP 208
Computers in Engineering

Lecture 10

Jun Wang
School of Computer Science
McGill University

Fall 2007

Review

- Loops
  - Definite iterator (counter-controlled loop)

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

  - Indefinite iterator (expression-controlled loop)
    - must use `EXIT` statement to terminate
    - EXIT can appear anywhere inside the loop

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

  - DO WHILE loop
    - a special type of indefinite iterator that checks the termination condition at the beginning of the loop body

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

An exercise on loops

- A game works like this: a coin is flipped, if it is heads, you win the coin, and another coin is flipped. You keep winning until you get tails, at which point you get to keep the coin but the game is over. We want to know on average how many coins you can expect to win.

- (This is an example of the concept known as expectation in the probability theory)

- Write a program to calculate

  $1/2.0 + 2/4.0 + 3/8.0 + 4/16.0 + 5/32.0 + ... + n/2^n$, where n is an input from user.
  - 1. Ask the user for the number n
  - 2. Write a loop to ensure n is between 10 and 50
  - 3. Write a loop to calculate $1/2.0 + 2/4.0 + 3/8.0 + ... + n/2^n$
  - 4. Print out the result

Try to do it without using **

Solution

```
!==========================================================
! Calculate 1*/2.0 + 2/4.0 + 3/8.0 + ... + n/2**n
!==========================================================
PROGRAM expectation
    INTEGER :: n, i
    REAL :: result, denom

    DO
      WRITE (*,*) "Enter a number between 10 and 50:"
      READ (*,*) n
      IF (n >= 10 .AND. n <= 50) EXIT
    END DO

    result = 0.0
    denom = 2.0

    DO i=1, n
      result = result + i/denom
      denom = denom * 2.0
    END DO

    WRITE (*,*) "Expectation = ", result
END PROGRAM expectation
```

The Ins and Outs of FORTRAN

Input and Output

Nathan Friedman

FORTRAN Formatted Input/Output

- The READ and WRITE statements we have seen so far are called ***free-format*** statements. They are easy to use but we have no control over the placement of the input or appearance of the output.

- To control the appearance of the input and output, Fortran allows us to use **format specifications**

How much was that?

```
PROGRAM cost
  IMPLICIT NONE
  REAL :: price, gst, pst
  READ(*,*) price

  gst = 0.07*price
  pst = 0.075*(price + gst)
  WRITE(*,*) "Price: ",price
  WRITE(*,*) "GST: ", gst
  WRITE(*,*) "PST: ", pst
  WRITE(*,*) "Total Cost: ", price+gst+pst
END PROGRAM cost
```

The results aren’t very pretty.

```
>
136.95
 Price:     136.9500
 GST:     9.586500
 PST:     10.99024
 Total Cost:     157.5267
```

Wouldn’t this be nicer?

```
>
136.95
             Price:  136.95
               GST:    9.59
               PST:   10.99
        Total Cost:  157.53
```

Formats

- FORTRAN formats allow us to specify the placement of values both in output and input
- Using format descriptors we can control the appearance of output values
- Format descriptors specify
  - The appearance of output values
  - Repetition
  - Vertical positioning
  - Horizontal positioning

Cost With Formatting

```
PROGRAM cost
    IMPLICIT NONE
    REAL :: price, gst, pst
    READ(*,*) price
    gst = 0.07*price
    pst = 0.075*(price + gst)
    WRITE(*,100) "Price: ",price
    WRITE(*,100) "GST: ", gst
    WRITE(*,100) "PST: ", pst
    WRITE(*,100) "Total Cost: ",price+gst+pst
100 FORMAT (A15,F7.2)
END PROGRAM cost
```

label

Fortran Formats

The FORMAT **statement** in the previous example specifies a format

```
100 FORMAT (A15,F7.2)
```

A **format** is list of descriptors inside parentheses

```
( .... format descriptors .... )
```

Fortran Formats – Method 1

There are three possible ways to specify a format.

In the first, we write the format as a character string and use it to replace the second asterisk in `WRITE(*,*)`.

```
WRITE(*,"(A15,F7.2)") "Total Cost: ", price+gst+pst
```

Fortran Formats – Method 2

The most common method uses a FORMAT statement

A FORMAT statement has the syntax:

```
label FORMAT format-code
```

To use the format, we specify its label in the WRITE statement

```
    WRITE(*,100) "Total Cost: ", price+gst+pst
100 Format (A15,F7.2)
```

FORMAT statement is NOT executable

Format Codes

- We will look at some of the many format codes available in FORTRAN for specifying:
  1. Real values
  2. Integer values
  3. Character values
  4. Horizontal spacing
  5. Vertical spacing

Real Values
Fixed Point Notation

```
100 Format (A15,F7.2)
```

- The second format code in the list specifies that we are to print a real number using two decimal points
- The in the code tells the computer to allow seven spaces to fit the number into

Real Numbers – Fixed Point

- The **`F`** descriptor
- The general format code has the form

  `Fw.d`
- The d specifies the number of decimal places
- The w specifies the field width and includes space for
  1. d decimal digits
  2. The decimal point
  3. The whole number
  4. The sign, if the number is negative

F Format Example

Example

```
       REAL :: x=1.0, y=1100.1003
       WRITE(*, 900) x, y
900    FORMAT (F3.1, F10.4)
```

F3.1 is format code for x and F10.4 is for y

```
1.0 1100.1003
```

Variations on a Theme

Example:

```
real :: x=1.0, y=1100.1003
write (*,"(F3.1,F11.4)") x, y
write (*,"(F3.1,F10.4)") x, y
write (*,"(F3.1,F9.4)") x, y
write (*,"(F3.1,F8.4)") x, y
```

Results:

```
1.0  1100.1003
1.0 1100.1003
1.01100.1003
1.0********
```

Oops!

- What happened in the last example?
- Whenever a value to be output does not fit into the allocated field width, w, the computer just outputs w *'s\
- This is true of any type of value, not just real numbers

Real Numbers Exponential Notation

- The E format descriptor has the form

  `Ew.d`

- They are displayed as a normalized number between 0.1 and 1.0, multiplied by a power of 10
- The output is in the form

  `±0.ddddE±ee`

- The number of significant digits is specified by d, the exponent uses 2 places
- `w` specifies the width
- We must have $w \geqslant d+7$

E code variations

Example:

```
real :: y=1100.1003
write (*,"(E15.8)") y
write (*,"(E15.4)") y
write (*,"(E15.2)") y
write (*,"(E12.8)") y
```

Results:

```
 0.11001003E+04
---------------
     0.1100E+04
---------------
       0.11E+04
---------------
************
------------
```

Integer Numbers “I” Format Codes

- The general format code for has the form

    `Iw`

- The w specifies the field width
- Numbers are right justified
- If a number doesn’t fit, *’s are output

Character Values "A" Format Code

- The general format code for has the form

  `Aw`

- The w specifies the field width
- Strings are right justified
- If a number doesn't fit, the first w characters are output
- If w is left out, the entire character string is printed

Repetition Factors

- A format code or group of codes can be repeated by putting a value in front
- For example:
  - `10I1` means output (or input) 10 digits
  - `5(A3, I5)` is equivalent to

    `A3, I5, A3, I5, A3, I5, A3, I5, A3, I5`

Horizontal Spacing

- To skip a space horizontally, we have the format code X
- Using a repetition factor, nX, indicates “skip n spaces”

```
      INTEGER :: a=1000
      WRITE (*,100) "a=", a
 100  FORMAT(A, 4X, I4)
```

- Output

```
a=    1000
----------
```

Vertical Spacing

- To skip a space vertically, we have the format code /
- Using a repetition factor, n/, indicates “skip n lines”

```
       INTEGER::a=1000
       WRITE(*,100) "a=", a
100    FORMAT(A, 2/, I4)
```

- Output

```
a=

1000
```

1 empty line here

The / descriptor may not work with `f77`

Format on Input

Example

```
      INTEGER :: a,b
      READ(*,100) a,b
  100 FORMAT(2I5)
```

This reads the first 5 characters on the input line, converts them to an integer and stores the result in a.

It then reads the next five characters, converts them and stores the result in b

Formatted Read

Correct inputs for Format code 2I5:

```
1234567890     → a=12345, b=67890
123456         → a=12345, b=6
####12345#     → a=1, b=2345
###1234567890  → a=12, b=34567
```

Incorrect inputs:

```
1234,5678
123456789a
12, 14
```

sign is used here to represent a white-space

Reading Fixed Point Reals

Example

```
READ(*, "(F5.1)") x
```

Results

```
##3.4        → x=3.4
123.456      → x=123.4
12345        → x=1234.5
```

Real numbers may be entered without decimal point.

Histogram Example

- Suppose we have a list of grades for all students in the class
- We would like to count how many received A, B, C, D and F
- To help visualize the distribution, we output a histogram with a line for each category and a "*" for each grade within that category

Sample Input

The input data consists of the number of students followed by the grades received by each student.

For example:

```
20
78   95   68   85   55
88   82   75   63   90
85   76   82   40   68
37   59   67   49   78
```

Expected Output

For the given input, we would like the following output:

```
Histogram:
 ******* (7)
 ****** (6)
 **** (4)
 (0)
 *** (3)
```

Maintaining the Grades

How do we keep track of the grades?

Use an array to store the grades

- The number of data elements depends on the size of the class
- To make the program general and capable of handling different size classes, we allocate an array large enough for any anticipated class size but we only input the grades based on the actual class size

Maintaining the Groupings

How do we keep track of how many grades are in each category (A, B, C, D and F)?

Use an array of size 5 called Bucket

- Each cell of the array will hold a count of the number of grades in a category
- Bucket(5) counts the number of A's, Bucket(4) counts the number of B's, etc.

Histogram – v1 Initialization

```
! ----------------------------------------------------
! Plot a histogram showing the number of grades in the
! ranges [0,49], [50,54], [55,64], [65,79] and [80,100].
! ----------------------------------------------------
PROGRAM  Histogram
  IMPLICIT NONE
  INTEGER :: Grades(300)
  INTEGER :: ClassSize, i
  INTEGER :: Bucket (5)

  READ(*,*)  ClassSize, (Grades(i), i = 1, ClassSize)
  ! ensure classSize <= 300

  DO i = 1, 5   ! initialize buckets to 0
    Bucket(i) = 0
  END DO
```

Histogram: computation

```
! Distribute each grade into the appropriate bucket
DO i = 1, ClassSize
  IF (Grades(i) < 0 .OR. Grades(i) > 100) THEN
    WRITE(*,*) "Invalid grade for student", i
  ELSE IF (Grades(i) <= 49) THEN
    Bucket(1) = Bucket(1) + 1
  ELSE IF (Grades(i) <= 54) THEN
    Bucket(2) = Bucket(2) + 1
  ELSE IF (Grades(i) <= 64) THEN
    Bucket(3) = Bucket(3) + 1
  ELSE IF (Grades(i) <= 79) THEN
    Bucket(4) = Bucket(4) + 1
  ELSE
    Bucket(5) = Bucket(5) + 1
  END IF
END DO
```

Histogram Display

```
! For each bucket, display a line of '*'s
! The number of '*'s displayed is the size of the
  bucket

  WRITE(*,*) "Histogram:"
  WRITE(*,*)
  DO i = 5, 1, -1
    WRITE(*,*) &
      ('*', j=1,Bucket(i)), ' (',   Bucket(i), ')'
  END DO

END PROGRAM  Histogram
```

implied loop

Using Files

- It’s a lot of work to enter the grades for a large class
- It’s also very prone to errors
- These values are often generated by other programs such as spreadsheets or by word processors and stored in files
- We would like to read the values directly from these files and be able to write them to other files

File Input and Output

- `READ(*,*) and WRITE(*,*)` read from and write to the standard input (keyboard) and output (screen) devices.
- To read from a file, we have to specify the name of the file and give the program some way of identifying it
- We use this identification to refer to the file in the program

File input/output

Three steps are required in using a file

1. Open the file
2. Input/output using READ and WRITE
   - READ: read data from the opened file
   - WRITE: write data to the opened file
3. Close the file (A **file** that has not been closed can usually not be accessed afterwards.)

OPEN a File

- To open a file, we provide a way for the program to reference a file maintained by the operating system.
- We have to specify the name of the file used by the operating system (a full path name)
- We also have to specify how the program will refer to that file **internally**
- In Fortran we use a **unit number** (rather than a name) to reference the file

OPENing a File

Example:

```
OPEN(UNIT=10, FILE=“expData.txt”)
OPEN(10, FILE=“expData.txt”)
```

FILE refers to the name of a file in the operating system

- If the file is in the same directory as the program, the name is enough
- Otherwise we must specify the path to the file (e.g. `“C:My Documents\208\expData.txt”`

OPENing a File

General Syntax:

```
OPEN ([olist])
```

where, olist is a list of keyword clauses of the form

```
keyword "=" value
```

We use the keywords UNIT and FILE. There are many others we do not use in this course.

UNIT assigns an number as an internal “name” for the program to reference the file

File is the external system name for the file

FILE input/output

- Once we have opened a file we can read the data that was stored there or we can output data to the file.
- We use the internal unit number to reference the file

```
READ(unit, *) …
WRITE(unit, *) …
```

CLOSE

When we finish using the file we must close it. A **file** that has not been closed can usually not be read again.

Syntax:

```
CLOSE ([UNIT=]u)
```

For example:

```
CLOSE (10)
CLOSE (UNIT=10)
```

Output Example

```
! Input 10 integers and write them to "Data.txt"
PROGRAM fileTest
    IMPLICIT NONE
    INTEGER::count, a
    OPEN(UNIT=10,FILE="Data.txt")
    DO count=1,10
     READ(*,*) a
     WRITE(10,*) a
   END DO
   CLOSE(10)
END PROGRAM
```

Histogram Program

```
PROGRAM  Histogram
  IMPLICIT NONE
  REAL :: Grades(300)
  INTEGER :: ClassSize, i
  INTEGER :: Bucket (5)

  ! Read data from file
  OPEN (UNIT=13, FILE="histogramdata.txt")
  READ(13,*) ClassSize, (Grades(i),i = 1, ClassSize)
  CLOSE(UNIT=13)

  ! computation part omitted: same as before

  ! Write result to file
  OPEN (UNIT=15, FILE="hist.txt")
  DO i = 5, 1, -1
    WRITE(15,*)  Bucket(i)
  END DO
  CLOSE(UNIT=15)
END PROGRAM  Histogram
```

Multidimensional Arrays

Nathan Friedman

Fall, 2007

A Two Dimensional Array

A small hotel with four floors and six rooms on each floor could use a table with four columns and six rows to represent the number of occupants in each room:

$$\begin{pmatrix} 2 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 2 & 3 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 2 \\ 0 & 2 & 0 & 0 & 1 & 0 \end{pmatrix} \quad \begin{bmatrix} 5 \\ 6 \\ 4 \\ 3 \end{bmatrix}$$

One dimensional array can describe total number of guests on each floor

Think of this as an array whose elements are also arrays

Multidimensional Arrays

- Think of a table with rows and columns.
- That forms a two dimensional array.
- A pile of these tables one on top of the other is a three dimensional array
  - e.g. 3 tables showing the occupants over 3 days
- Fortran allows up to 7 dimensions

Using Tables

- We must give the table a name to be able to reference it
- The rows and columns are referenced by a row index and a column index

rooms

$$\begin{pmatrix} 2 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 2 & 3 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 2 \\ 0 & 2 & 0 & 0 & 1 & 0 \end{pmatrix}$$

row indices: 1 2 3 4

column indices: 1 2 3 4 5 6

Accessing Values

To find the number of people occupying a room, we specify

1. the floor (the row index) and
2. the room (the column index)

For example the number of occupants in the 5th room on the 2nd floor could be accessed by indexing as **`rooms(2,5)`**