

BBM 101 – Introduction to Programming I

Fall 2014, Lecture 5

Aykut Erdem, Erkut Erdem, Fuat Akal

1

Today

■ Iteration Control

- Loop Statements
 - **for, while, do-while** structures
- **break** and **continue**
- Some simple numerical programs

2

Loop Statements

- Loop is a group of instructions computer executes repeatedly while some condition remains true
- Counter-controlled Repetition
 - Definite repetition: know how many times loop will execute
 - Control variable used to count repetitions
- Sentinel-controlled Repetition
 - Indefinite repetition
 - Used when number of repetitions not known
 - Sentinel value indicates "end of data"

3

The for Loop

■ Syntax

```
for (initialization; condition; modify)
    statement;
```

- The program will keep executing the *statement* inside the **for** as long as the condition is true (non zero)
- The *condition* is tested **before** each iteration of the loop. The loop terminates when the condition is false.
- The loop is controlled by a variable which is initialized and modified by the *initialization* and *modify* (e.g. increment operation) expressions, respectively.

4

The for Loop (Example)

- Find the sum of numbers between 1 and 100

```
int sum = 0;
for (i = 0; i <= 100; i++) {
    sum = sum + i;
}
```

control variable i = initial value 0

loop continuation condition
(100 is the final value of i for which the condition is true)

increment of control variable i

5

The for Loop (Further Examples)

- Loop from 100 to 1 in increments of -1

```
for (i = 100; i >= 1; i--)
```

value of i when the loop terminates is 0.

- Loop from 7 to 77 in increments of 7

```
for (i = 7; i <= 77; i+7)
```

value of i when the loop terminates is 84.

6

Example: A program that prints the sum of even numbers between 0 and 100

```
/*Summation with for */
#include <stdio.h>

int main()
{
    int sum = 0, number;
    for ( number = 2; number <= 100; number += 2 ){
        sum += number;
    }
    printf( "Sum is %d\n", sum );
    return 0;
}
```

Sum is 2550

7

The while Loop

- Syntax

```
while (condition)
    statement;
```

- The program will repeatedly execute the *statement* inside the **while** as long as the condition is true (non zero)
- The *condition* is tested **before** each iteration of the loop. The loop terminates when the condition is false.
- If the condition is initially false (0), the statement will not be executed.

8

The while Loop (Example)

- Find the sum of numbers between 1 and 100

```
int sum = 0, i = 1;
while (i <= 100) {
    sum = sum + i;
    i = i + 1;
}
```

9

Counter Controlled Repetition (Example)

- A class of 10 students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz.

- The algorithm

- Set total to zero
- Set grade counter to one
- While grade counter is less than or equal to 10
Input the next grade
Add the grade to the total
Add one to the grade counter
- Set the class average to the total divided by ten
- Print the class average

10

```
/* Class average program with counter-controlled repetition */
#include <stdio.h>

int main()
{
    int counter, grade, total, average;

    /* initialization phase */
    total = 0;
    counter = 1;

    /* processing phase */
    while ( counter <= 10 ) {
        printf( "Enter grade: " );
        scanf( "%d", &grade );
        total = total + grade;
        counter = counter + 1;
    }

    /* termination phase */
    average = total / 10.0;
    printf( "Class average is %d\n", average );

    return 0; /* indicate program ended successfully */
}
```

```
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81
```

11

Sentinel Controlled Repetition (Example)

- Revisiting the class average problem: Arbitrary number of students took the quiz this time.
 - i.e., number of students will not be known when the program runs
 - How is the program going to know when to end?
- Use sentinel value
 - Also called *signal value*, *dummy value*, or *flag value*
 - Indicates *end of processing*
 - Loop ends when user inputs the sentinel value
 - Sentinel value is chosen in a way that it cannot be confused with a regular input

12

```

/* Class average program with sentinel-controlled repetition */
#include <stdio.h>
int main()
{
    float average;
    int counter, grade, total;

    /* initialization phase */
    total = 0;
    counter = 0;

    /* processing phase */
    printf( "Enter grade, -1 to end: " );
    scanf( "%d", &grade );
    while ( grade != -1 ) {
        total = total + grade;
        counter = counter + 1;
        printf( "Enter grade, -1 to end: " );
        scanf( "%d", &grade );
    }

    /* termination phase */
    if( counter != 0 ) {
        average = ( float ) total / counter;
        printf( "Class average is %.2f", average );
    } else
        printf( "No grades were entered\n" );

    return 0; /* indicate program ended successfully */
}

```

```

Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50

```

13

The do-while Loop

■ Syntax

```

do {
    statement;
} while ( condition )

```

- The program will definitely execute the statement at least once and then repeatedly keep executing the *statement* inside the **do-while** as long as the condition is true (non zero)
- The *condition* is tested **after** each iteration of the loop. The loop terminates when the condition is false.
- If the condition is initially false (0), the statement will be executed anyways.

14

The do-while Loop (Example)

- Find the sum of numbers between 1 and 100

```

int sum = 0, i = 1;
do {
    sum = sum + i;
    i = i + 1;
} while ( i <= 100)

```

Which example better
suits for the use of do-
while loop?

- Try until the user enters a valid number

```

int number;
do {
    printf("Enter a number from 0 to 100: ");
    scanf("%d", &number);
} while ( number >= 0 && number <= 100)

```

15

Nesting Control Structures

■ Problem

- A college has a list of test results (1 = pass, 2 = fail) for 10 students
- Write a program that counts the number of passed and failed students

■ Notice that

- The program must process 10 test results
 - Counter-controlled loop will be used
- Two counters can be used
 - One for number of passes, one for number of fails
- Each test result is a number—either a 1 or a 2
 - If the number is not a 1, we assume that it is a 2

16

Nesting while loop and if structure

```
#include <stdio.h>
int main()
{
    int passes = 0, failures = 0, student = 1, result;

    while(student <= 10){
        printf( "Enter result: 1(Pass), 2(Fail): " );
        scanf( "%d", &result);

        if(result == 1)
            passes++;
        else
            failures++;

        student = student + 1;
    }

    printf("Passed: %d Failed: %d\n", passes, failures);

    return 0;
}
```

Nesting while loop and switch structure

```
#include <stdio.h>

int main()
{
    char grade;
    int aCount=0, bCount=0, cCount=0, dCount=0, fCount=0 ;

    printf( "Enter the letter grades. Enter X to exit. \n" );

    while((grade = getchar()) != 'X') {

        switch ( grade ) {
            case 'A': case 'a': ++aCount; break;
            case 'B': case 'b': ++bCount; break;
            case 'C': case 'c': ++cCount; break;
            case 'D': case 'd': ++dCount; break;
            case 'F': case 'f': ++fCount; break;
            default:printf( "Incorrect letter grade entered." );
                    printf( "Enter a new grade.\n" );
                    break;}
    }
}
```

Nested Loops

- When a loop body includes another loop construct this is called a *nested loop*.
- In a nested loop structure the inner loop is executed from the beginning every time the body of the outer loop is executed.

```
value = 0;
for (i=1; i<=10; i=i+1)
    for (j=1; j<=5; j=j+1)
        value = value + 1;
```

- How many times the inner loop is executed? → 50 times

Nested Loops (Example)

- How many times the inner loop is executed?

```
for (i=1; i<=5; i=i+1){
    for (j=1; j<=i; j=j+1)
        printf("\n");
    printf("\n");
}
```

i	j
1	1
2	1, 2
3	1, 2, 3
4	1, 2, 3, 4
5	1, 2, 3, 4, 5

Output

```
*
**
***
****
*****
```

→ 15 times

Nesting while and for Loops

```
int main()
{
    int num, count, total = 0;

    printf("Enter a value or a negative number to end: ");
    scanf("%d", &num );

    while( num >= 0 ) {
        for (count = 1; count <= num; count++)
            total = total + count;

        printf("%d %d", num, total);
        printf( "Enter a value or a negative number to end:");
        scanf( "%d", &num );
        total = 0;
    }
    return 0;
}
```

This program reads numbers until the user enters a negative number. For each number read, it prints the number and the summation of all values between 1 and the given number.

21

The break Statement

- Causes immediate exit from a *while*, *for*, *do...while* or *switch* statement
- Program execution continues with the first statement after the containing block
- Common uses of the break statement
 - Escape early from a loop
 - Skip the remainder of a switch statement

22

The break Statement (Example)

```
#include <stdio.h>

int main(){
    int x;

    for (x = 1; x <= 10 ; x++) {
        if ( x == 5 )
            break;
        printf("%d ", x);
    }

    printf("\nBroke out of the loop at x =%d ", x);
    return 0;
}
```

```
1 2 3 4
Broke out of the loop at x = 5
```

23

The continue Statement

- Skips the remaining statements in the body of a *while*, *for* or *do...while* statement
 - Proceeds with the next iteration of the loop
- *while* and *do...while* loops
 - Loop-continuation test is evaluated immediately after the continue statement is executed
- *For* loop
 - Increment expression is executed, then the loop-continuation test is evaluated

24

The continue Statement (Example)

```
#include <stdio.h>

int main() {
    int x;

    for (x = 1; x <= 10 ; x++) {
        if ( x == 5 )
            continue;
        printf("%d ", x);
    }

    printf("\nUsed continue to skip printing the value 5");
    return 0;
}
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5
```

25

Exhaustive Enumeration

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main() {
    int x, ans;
    printf("Enter an integer: ");
    scanf("%d", &x);
    ans = 0;
    while (pow(ans,3)<abs(x))
        ans++;

    if (pow(ans,3)!=abs(x))
        printf("%d is not a perfect cube\n", x);
    else {
        if (x<0) ans = -ans;
        printf("Cube root of %d is %d\n", x, ans);
    }
    return 0;
}
```

*This program finds the cube root of a perfect cube using a variant of **guess and check** technique called **exhaustive enumeration**.*

- Enumerate all possibilities until we get the right answer or exhaust the space of possibilities.

26

Approximate Solutions

- Suppose we want to find the square root of any non-negative number?
- Can't guarantee exact answer, but just look for something close enough
- Start with exhaustive enumeration
 - Take small steps to generate guesses in order
 - Check to see if close enough

Square root of any non-negative number

```
int x = 25;
double epsilon = 0.01;
double step = epsilon*epsilon;
int numGuesses = 0;
double ans = 0.0;
while (fabs(ans*ans - x) >= epsilon && ans <= x) {
    ans += step;
    numGuesses++;
}
printf("numGuesses = %d\n", numGuesses);
if (fabs(ans*ans - x) >= epsilon)
    printf("Failed on square root of %d", x);
else
    printf("%.3lf is close to square root of %d", ans, x);
```

```
numGuesses = 49990
4.999 is close to square root of 25
```

Square root of any non-negative number

```
int x = 25;
double epsilon = 0.01;
double step = epsilon*epsilon;
int numGuesses = 0;
double ans = 0.0;
while (fabs(ans*ans - x) >= epsilon && ans <= x) {
    ans += step;
    numGuesses++;
}
printf("numGuesses = %d\n", numGuesses);
if (fabs(ans*ans - x) >= epsilon)
    printf("Failed on square root of %d", x);
else
    printf("%.3lf is close to square root of %d", ans, x);
```

```
numGuesses = 49990
4.999 is close to square root of 25
```

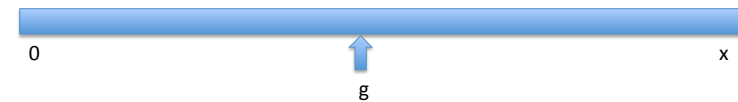
Step could be any small number

- If too small, takes a long time to find square root
- If make too large, might skip over answer without close enough

29

Bisection Search

- We know that the square root of x lies between 0 and x , from mathematics
- Rather than exhaustively trying things starting at 0, suppose instead we pick a number in the middle of this range



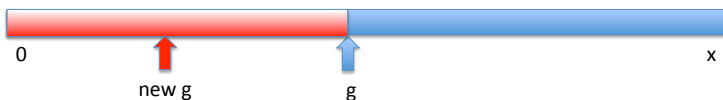
- If we are lucky, this answer is close enough

Slide credit: E. Grimson, J. Guttag and C. Terman

30

Bisection Search

- If not close enough, is guess too big or too small?
- If $g^2 > x$, then know g is too big; but now search



- And if this new g is, for example, $g^2 < x$, then know too small; so now search



- At each stage, reduce range of values to search by half

Slide credit: E. Grimson, J. Guttag and C. Terman

31

Approximating Square Root using Bisection Search

```
int x = 25;
double epsilon = 0.01;
int numGuesses = 0;
double low = 0.0;
double high = MAX(1.0, x);
double ans = (high + low)/2.0;
while (fabs(ans*ans - x) >= epsilon) {
    printf("low = %.5lf high = %.5lf ans = %.5lf\n", low, high, ans);
    numGuesses++;
    if (ans*ans < x)
        low = ans;
    else high = ans;
    ans = (high + low)/2.0;
}
printf("numGuesses = %d\n", numGuesses);
printf("%.5lf is close to square root of %d", ans, x);
```

32

Approximating Square Root using Bisection Search

```
int x = 25;
double epsilon = 0.01;
int numGuesses = 0;
double low = 0.0;
double high = MAX(1.0, x);
double ans = (high + low)/2;
while (fabs(ans*ans - x) > epsilon)
    printf("low = %.5lf high = %.5lf ans = %.5lf\n", low, high, ans);
    numGuesses++;
    if (ans*ans < x)
        low = ans;
    else high = ans;
    ans = (high + low)/2.0;
}
printf("numGuesses = %d\n", numGuesses);
printf("%.5lf is close to square root of %d", ans, x);
```

```
low = 0.00000 high = 25.00000 ans = 12.50000
low = 0.00000 high = 12.50000 ans = 6.25000
low = 0.00000 high = 6.25000 ans = 3.12500
low = 3.12500 high = 6.25000 ans = 4.68750
low = 4.68750 high = 6.25000 ans = 5.46875
low = 4.68750 high = 5.46875 ans = 5.07812
low = 4.68750 high = 5.07812 ans = 4.88281
low = 4.88281 high = 5.07812 ans = 4.98047
low = 4.98047 high = 5.07812 ans = 5.02930
low = 4.98047 high = 5.02930 ans = 5.00488
low = 4.98047 high = 5.00488 ans = 4.99268
low = 4.99268 high = 5.00488 ans = 4.99878
low = 4.99878 high = 5.00488 ans = 5.00183
numGuesses = 13
5.00031 is close to square root of 25
```

Approximating Square Root using Bisection Search

```
int x = 25;
double epsilon = 0.01;
int numGuesses = 0;
double low = 0.0;
double high = MAX(1.0, x);
double ans = (high + low)/2;
while (fabs(ans*ans - x) > epsilon)
    printf("low = %.5lf high = %.5lf ans = %.5lf\n", low, high, ans);
    numGuesses++;
    if (ans*ans < x)
        low = ans;
    else high = ans;
    ans = (high + low)/2.0;
}
printf("numGuesses = %d\n", numGuesses);
printf("%.5lf is close to square root of %d", ans, x);
```

```
low = 0.00000 high = 25.00000 ans = 12.50000
low = 0.00000 high = 12.50000 ans = 6.25000
low = 0.00000 high = 6.25000 ans = 3.12500
low = 3.12500 high = 6.25000 ans = 4.68750
low = 4.68750 high = 6.25000 ans = 5.46875
low = 4.68750 high = 5.46875 ans = 5.07812
low = 4.68750 high = 5.07812 ans = 4.88281
low = 4.88281 high = 5.07812 ans = 4.98047
low = 4.98047 high = 5.07812 ans = 5.02930
low = 4.98047 high = 5.02930 ans = 5.00488
low = 4.98047 high = 5.00488 ans = 4.99268
low = 4.99268 high = 5.00488 ans = 4.99878
low = 4.99878 high = 5.00488 ans = 5.00183
numGuesses = 13
5.00031 is close to square root of 25
```

- *Bisection search radically reduces computation time – being smart about generating guesses is important*
- *Should work well on problems with “ordering” property – value of function being solved varies monotonically with input value*
 - *Here ans*ans which grows as ans grows*

Summary

■ Iteration Control

- Loop Statements
 - for, while, do-while structures
- break and continue
- Some simple numerical programs

Next week

■ Functions

- Definitions
- Invocation
- Parameter Lists
- Return Values
- Prototypes

■ Variable Scopes

- Block Structure
- Global and Local Variables
- Static Variables

■ Recursion