EE 107 Intro. to ECET with software development, Spring 2020

Department of Electrical and Computer Engineering & Technology, MNSU
Dr. Qun Zhang

Handout 4 most related to Chapters 4

Recap: Enough for expressions (skip)

Part 1. Why we need loops? Motivational example: TAT-14 system design

- Review and investigate examples with algorithms of iteration

1. Figure out a solution to x^2=2
2. Bisection method in search
3. New: From Binomial distribution to normal distribution (detailed explanation)

- C loop statements
  A loop is a group of instructions that the computer executes repeatedly while some condition stays true.

  Two basic forms of loops:
  - Counter-controlled repetition.
  - Event/sentinel-controlled repetition.

```
#include <stdio.h>
void main(void)
{
   int counter;
   counter=1;
   while (counter <= 23)
    {
                printf("The value of mycounter is:%d\n", counter);
                ++counter;
        }
}
```

vs.

```
#include <stdio.h>
void main(void)
{
```

```
    int x=0;
    int sum = 0;
    printf("Enter your numbers to add. Enter <EOF> if you wish to stop \n");
    /* EOF is <ctrl+z> or <ctrl+d>*/
    while(scanf("%d", &x) != EOF)
        sum+=x;
  printf("I am out of the loop \n");
  printf("The total is %d\n", sum);
}
```

Three C loop statements

- *while* loops
- *for* loops
- *do…while* loops

1. some details of while loop vs. for loop:

- Syntax

```
while (expression)
{
   statement-1;
   statement-2;
     ……
   statement-n;
}
```

- A pretest loop: in each iteration, the loop control expression is tested first. If it's true, the loop body (statements between the braces) is executed. If it's false, the loop is terminated
- Braces are not required if the loop body consists of only one statement
- No semicolon is needed at the end of the while statement!

```
for(statement1;statement2;statement3)
{
    loop_body
}
```

- statement1: contains the initial value of the control variable
- statement2: contains the final value of the control variable
- statement3: increments/decrements the control variable
- Braces are not required if the loop body consists of only one statement
- Pre-test: loop-continuation condition (statement2) is tested before the loop.
- Example:

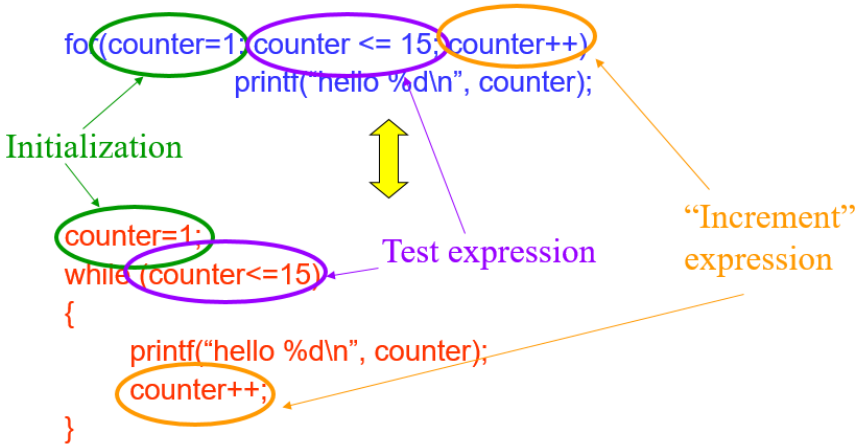
```
for(counter=1; counter <= 15; counter++)
    printf("hello %d\n", counter);
```

- Note: The 3 expressions in the *for* structure are optional. The two semicolons are required.

- If the counter variable is being initialized elsewhere in the program, then *statement1* can be omitted.
- If *statement2* is omitted, then C assumes the testing condition is true and creates an infinite loop.
- *Statement3* can be omitted if the counter variable is being incremented (or decremented) elsewhere in the program.

Examples: 4.1 and 4.2. (first, what are triangular numbers?)

Equivalence of for-loop to a while loop:



Exercise: can you use while-loops for example 4.2 ?
Exercise:

Find error(s), if any, in the following for statements

- for (day=1,day<3,day++) printf("good morning\n")

  ; instead of , should be used!
  ; is missing at the end of printf()

- for (day=3;day<=10;day++) ;

  No syntax error, the do nothing statement ; will be executed!

- for (day=7;day<3;day++) printf("good morning\n");

  No syntax error, but its loop body will not be executed

- for (day=7;day<7;day--) printf("good morning\n");

  No syntax error, but its loop body will not be executed

2. The nested for-loops
   - Loop(s) within a loop
   - What's the output of this program?

```
#include <stdio.h>
void main(void)
{
```

```
int a;
int b;
for(a =1; a <= 3; a++)
        {
                for (b=1; b<=4; b++)
                        printf("%d", a);
                printf("\n");
        }
}
```

Solutions: The above nested *for* loop prints a series numbers on multiple lines as follows:
1 1 1 1
2 2 2 2
3 3 3 3

3. Do-while loop - example 4.9
   - Syntax

```
do
{
   statement-1;
   statement-2;
     ……
   statement-n;
} while (expression);
```

   - A post-test loop: in each iteration, the loop body is executed. Then the loop control expression is tested. If it's true, a new iteration is started; otherwise, the loop terminates
   - Braces are not required if the loop body consists of only one statement
   - The loop body is executed at least once
   - Semicolon is needed at the end of the *do…while* statement!!
   - while vs. do-while

```
while (expression)
   {
      statement-1;
      statement-2;
        ……
      statement-n;
   }
```

- Pre-test: loop-continuation condition is tested before the loop.

```
do
{
   statement-1;
   statement-2;
     ……
   statement-n;
} while (expression);
```

- Post-test: loop-continuation condition is tested after the loop.

4. break/continue
   - The *break* and *continue* statements are used in loops to change the flow of control.

- *break* is used to escape from a loop (causes a loop to terminate).
- *continue* is used to skip the remaining statements in the body of a structure and skip to the next iteration.
- Use integer variables in controlling loops.
- Indent appropriately.
- **Avoid using break or continue in the loops**

```
#include <stdio.h>
void main(void)
{
    int a;
    for(a =1; a <= 7; a++)
    {
            if(a == 4)
                break;
          printf("%d\n", a);
    }
    printf("I got out of the loop at a==%d\n",a);
}
```

vs.

```
#include <stdio.h>
void main(void)
{
    int a;
    for(a =1; a <= 7; a++)
  {
            if (a == 4)
                continue;
            printf("%d\n",a);
    }
}
```

- Use integer variables in controlling loops.
- Indent appropriately.
- Avoid using break; or continue; in the loops

- But some instructors like to use ‘break’ to solve the loop-and-a-half problem (e.g. at Stanford Univ.) and they argue that other than this ‘break’ many not be recommended.

1). Infinite loops (also how to correct it?)

```
while (n>=0){   // this is just a piece of code; not a complete program}
   Dsum+=n%10;
   n/=10;
   }
```

2). Loop-and-a-half

a. e.g. adding final exam scores

1. read in a value
2. if the value is equal to the sentinel, exit from the loop
3. perform summations (or whatever data processing)

b. Naturally:

```
while (TRUE){   // this is just a piece of code; not a complete program}
   promp user and read in a value
   if (value == sentinel)
            break;
   process the data value
   }
```

c. Not a very good way:

1. read in a value
2. while loop with test if it equal to the sentinel
3. perform summations (or whatever data processing)
4. read in a value (inside while loop)

problem 1 from the above algorithm: no usual logic (not natural)
problem 2: duplication of code (a serious code maintenance problem).