Introduction to Programming Language (C)

(EEF 110E)

Dr. Emre DİNCEL

LECTURE NOTES

(WEEK 3)

❖ Loops in C Programming

Loop statements allow us to execute a statement or group of statements multiple times. The following loop statements exist in language C to create repetitive tasks:

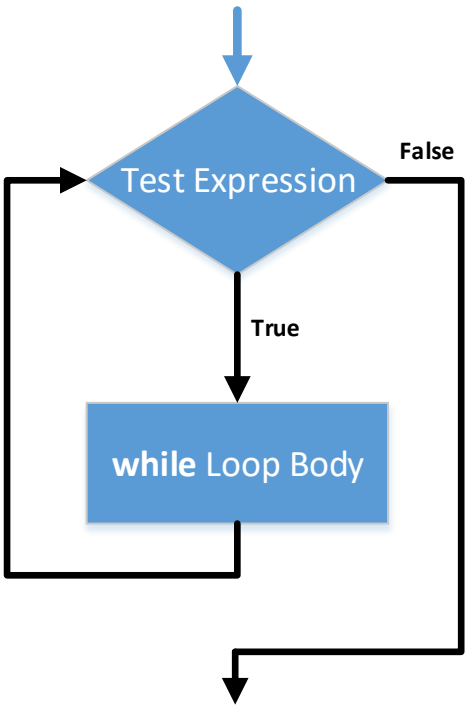
- While loop
- Do … while loop
- For loop

- While Loops:

The purpose of the **while** loop is to repeatedly execute a statement over and over while a given condition is 1 (or logic true). The syntax is similar to **if** statement and can be given as follows:

```
while (expression)
{
   Statements...
}
```

While the expression is nonzero (i.e., true) the statements written inside the while block are executed.



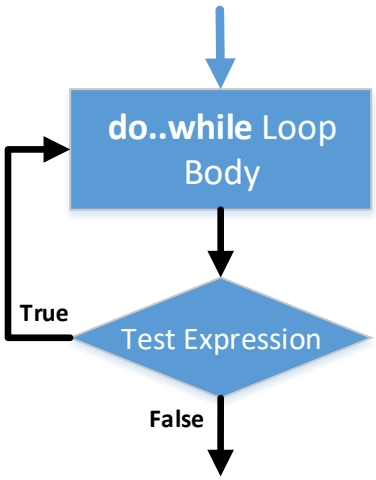
Please examine the following example:

```c
#include <stdio.h>

int main()
{
   int i;
   printf("Enter a positive number: ");
   scanf("%d", &i);

   while (i > 0)
   {
      printf("2 * %d = %d\n", i, 2*i);
      i--;
   }
   return 0;
}
```

Note that in a while loop, first the expression in while statement (the termination condition) is calculated. If the expression is zero than the statement(s) in the loop are not executed.

- Do … While Loops:

Sometimes the statement(s) inside the loop is required to be executed at least once before checking for the termination condition. In such cases, it is possible to use the **do ... while** loops.



Consider the following example in which the usage of do … while loop is illustrated:

```c
#include <stdio.h>

int main()
{
   int ch;
   printf("Enter a character (press x to exit):\n");
   do
   {
      ch = getchar();
      putchar(ch);
   } while (ch != 'x');
   printf("Out of the loop!\n");
   return 0;
}
```

- For Loops:

A **for** loop is actually a special form of the while loop. Check the syntax:

```c
for (initializer; condition; post_statement)
{
   Statements...
}
```

Note that in the while loop, the above form can be constructed as follows:

```c
initializer;
while (condition)
{
   Statements...
   post_statement;
}
```

Here, first of all, the “initializer” is executed. This generally involves an initialization of (giving first value to) a variable (usually referred as the counter or *loop variable*). Then “condition” is evaluated. While this expression evaluates to a nonzero value (logic true), the loop (statement) is repetitively executed. At the end of each loop “post statement”, which is usually an increment or decrement of a counter variable is executed.

Let us present the similar example used in while loop;

```c
#include <stdio.h>

int main()
{
   int i, n;
   printf("Enter a positive number: ");
   scanf("%d", &n);

   for (i = n; i > 0; i--)
      printf("2 * %d = %d\n", i, 2*i);

   return 0;
}
```

 Note that there is no semicolon after the "**for**" statement. If you accidentally put a semicolon it will not produce a syntax error since semicolon is a null (do nothing) statement in C. However, your program will not work as you wish it to work.

```c
for (i = 1; i < 20; i++);
```

The above code means that count from 1 to 20 and do nothing else. Please check the more examples given for the "**for"** loops:

Statement	Counter Values
`for (k=1; k<=3; k++)`	1, 2, 3
`for (k=1; k<12; k+=3)`	1, 4, 7, 10
`for (p=-15; p<=15; p+=5)`	-15, -10, -5, 0, 5, 10, 15
`for (p=12; p>10; p-=3)`	12
`for (j=3; j>5; j--)`	Loop is skipped

Exercise: Write a program that asks the marks of students in a class until a negative number is entered. Then the average of positive numbers entered should be calculated and printed on the screen.

- Using assignments as a condition:

It is possible (but could be dangerous) to use an assignment in a condition expression. The result of the assignment (zero or nonzero) is then interpreted as the result of the condition. Check the below example:

```c
#include <stdio.h>

int main()
{
   int c;
   printf("Enter a character: ");
   while (c = getchar() && (c != 'x' && c != 'X'))
      putchar(c);

   printf("Out of the loop!");

   return 0;
}
```

Similarly, we can also call functions, which return an integer number, inside the loops and use it as a condition.

```c
while (myFunction(a,b)) {
...
}
```

As long as the return value of the function "myFunction" is not zero (logic false), the above loop will continue to run.

- Nested Loops:

It is possible to use loops inside the other loops. For instance,

```c
for (i=1; i<=3; i++)
   for (j=i; j<=5; j+=2)
      printf("(%d, %d)\n", i, j);
```

The above code would print (1, 1), (1, 3), (1, 5), (2, 2), (2, 4), (3, 3), (3, 5) on the screen in separate lines.

Exercise: Write a C program that calculates the following mathematical (double) sum operation:

$$\sum_{\substack{k=1 \\ (\text{odd } k)}}^{n} \sum_{j=1}^{k} x * j$$

Here, x is a real number and n is a positive integer to be entered by the user.

- Multiple expressions in a for statement:

It is possible to give more than one statements in *expression1* and *expression3* parts of a "**for**" statement. In such a case, the expressions should be separated by commas. For instance,

```
for (i=0, j=10; i!=j; i++, j--)
   printf("(%d, %d)\n", i, j);
```

The above code prints out (0, 10), (1, 9), (2, 8), (3, 7), (4, 6) on the screen in separate lines.

- Break and Continue keywords:

Sometimes it is required to exit a loop. This usually happens when an error is detected, or a certain condition related to the nature of the loop is encountered. In such cases, **break** statement can be used for this purpose.

In execution time, when a "break" statement is encountered the execution continues from the line that follows the loop.

On the other hand, sometimes it is required to jump to the end of a loop without exiting the loop. It is possible to use the **continue** statement for such purposes.

In execution time, when a "continue" statement is encountered the execution continues from the brace that closes the loop. This means expresion3 in the for-loop will be executed and the loop will continue to do its work.

Check the below examples:

```
#include <stdio.h>

int main()
{
   int k;

   for (k=0; ; k++)
   {
      if (k > 10) break;
      else if (k % 3 == 0) continue;
      printf("%d", k);
   }

   return 0;
}
```

In the above code, no condition is given in for-loop but instead the condition which ends the loop is defined inside the loop. It means that if $k > 10$ then the loop breaks and jumps to the line where loop ends. On the other hand, if the mod 3 of the variable k is zero (i.e., k=3, 6, 9, ...), then the program skips the *printf* function and return back to the top of for loop (in which k is increased and loop continues to do its work).

As a result, on the screen, we will see the following numbers: 1, 2, 4, 5, 7, 8, and 10.

- Infinite Loops:

Sometimes it is required to exit a loop when a certain condition is met and unless this condition is met the loop is required to continue infinitely. Such loops are called infinite loops. In C language, it is possible to use loops with the following syntax in order to implement infinite loops:

```
for (;;) { ... }
while (1) { ... }
```

Obviously so as to avoid the loop go infinitely there has to be a break (or similar such as return) statement in the loop (most probably inside an **if** construct). It is programmer's responsibility to make sure that such a break statement is reached in execution time.