

# Lecture 3

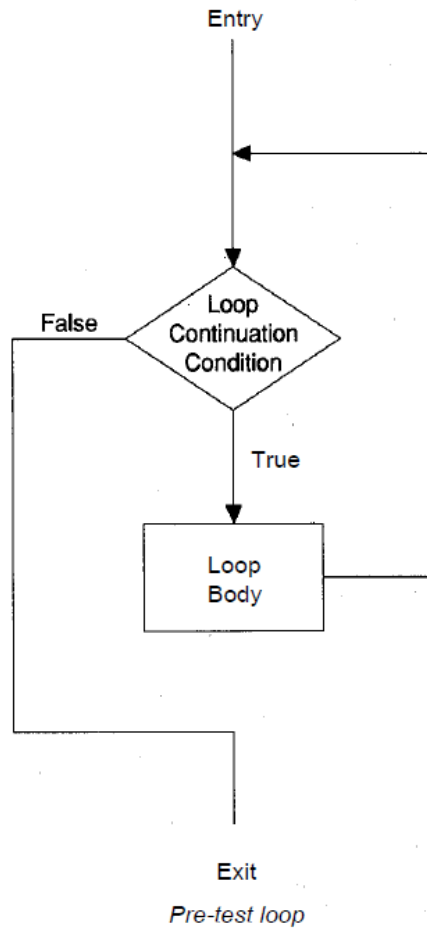
## Repetitive Structures

Dr. Hacer Yalım Keleş

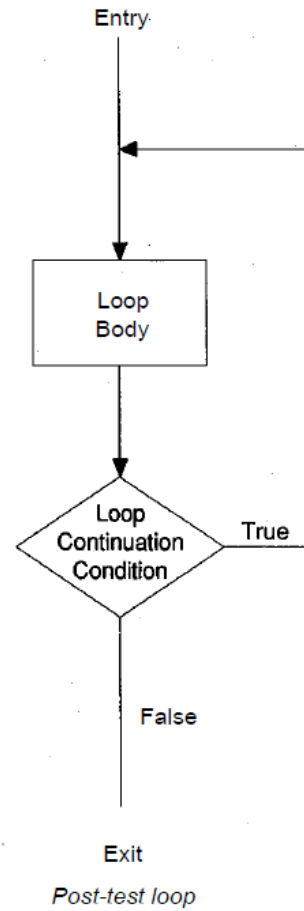
Ref: Programming in ANSI C, Kumar

The repetitive structure allows a sequence of program statements to be executed several times even though those statements appear only once in the program. It consists of an *entry point* that may include initialization of certain variables, a *loop continuation condition*, a *loop body*, and an *exit point*.

# Repetitive Structure-1



# Repetitive Structure-2



A repetitive structure must always be entered at the entry point to ensure that the appropriate initialization takes place.

The loop body consists of statements that are normally executed several times.

The exit point is the first statement following the loop body.

The number of repetitions in a repetitive structure is controlled by a *loop continuation condition*, which is tested once for each execution of the loop body. This condition normally involves testing the value of a *loop control variable* whose value is changed every time the loop body is executed.

For example,

The use of a counter to control the number of times a loop body is executed.

The value of the counter is changed, usually by 1,

for every execution of the loop body,

and when the counter attains a predetermined value,

the repetition is terminated.

The loop continuation condition may be tested before the loop body is executed, in which case the loop is referred to as a *pre-test* loop, or the condition may be tested after the execution of the loop body, in which case the loop is referred to as a *post-test* loop. Note that the body of a post-test loop is always executed at least once; the body of a pre-test loop may possibly never be executed.

## Example:

Consider the program for finding the sum of the first  $n$  terms of the series

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}$$

Here is an algorithm for solving this problem:

1. Read the value of  $n$ .
2. Initialize an accumulator *sum* to 0.
3. Repeat step (4)  $n$  times.
4. In the  $i$ th iteration, determine the value of the term  $1/i$ , and add it to *sum*. Thus,  $1/1$  is added to *sum* in the 1st iteration,  $1/2$  in the 2nd iteration,  $1/3$  in the 3rd iteration, etc.
5. Print *sum*.



```

#include <stdio.h>

int main(void)
{
    int i, n;
    float sum;

    scanf("%d", &n);    /* read n */
    i = 1;               /* initialize the loop control variable i */
    sum = 0;             /* initialize the accumulator sum */

    while (i <= n)       /* iterate n times */
    {
        sum += 1.0/i;    /* add the ith term to sum */
        i++;             /* increment the loop control variable i */
    }

    printf ("sum = %f\n", sum);    /* print sum */

    return 0;            /* successful completion */
}

```

*loop initialization*

```
while ( expression )  
    statement-1
```

*statement-2*

When the `while` statement is encountered, *expression* is evaluated. If *expression* evaluates to a nonzero value (*true*), *statement-1* is executed. *Expression* is then again evaluated, and if the result of this evaluation is nonzero, *statement-1* is again executed. This process continues until the result of *expression* evaluation becomes zero (*false*). The iteration is then terminated and the program execution continues from *statement-2*. If the result of *expression* evaluation is zero at the very outset, *statement-1* is not executed at all and the program control immediately transfers to *statement-2*.

```
int ch, count;

count = 0;
ch = getchar();

while (ch != EOF)
{
    putchar(ch);
    count++;
    ch = getchar();
}

printf("total characters echoed = %d\n", count);
```

## Infinite Loop

```
int n, sum = 0;

scanf("%d", &n);
while (n != 0)
{
    sum += n;
    n--;
}
printf("sum = %d\n", sum);
```

This loop correctly determines the desired sum as long as the user provides a nonnegative value for n.

```
float Celsius = 0;
while (celsius != 1.0)
{
    printf("%f %f\n", celsius, 1.8 * celsius + 32);
    celsius += 0.005;
}
```

The problem here is that a floating-point value such as 0.005 is represented inside a computer as an approximation of 0.005, and may not exactly equal 0.005. The loop continuation test in this example can be made more robust by changing it to

```
celsius <= 1.0
```

```
while (more = 1)
{
    .
    .
    .
    more = should_i_c.ontinue () ;
}
```

```
int i = 0, sum = 0, n;  
while (i < 25)  
{  
    scanf("%d", &n);  
    sum += n;  
}
```

```

#include <stdio.h>

int main(void)
{
    int c, vowel, vowelcnt, charcnt;

    vowelcnt = charcnt = 0;

    while ((c = getchar()) != EOF)
    {
        /* if c is an uppercase letter,
           convert it into a lowercase letter */
        c = (c >= 'A' && c <= 'Z' ) ?
            (c - 'A' + 'a') : c;

        /* test if c is a vowel */
        vowel = c == 'a' || c == 'e' ||
            c == 'i' || c == 'o' || c == 'u';

        if (vowel) vowelcnt++;
        charcnt++;
    }

    printf("vowels = %f\n",
        (float)vowelcnt/charcnt);

    return 0;
}

```



## do-while LOOP

*loop initialization*

```
do  
    statement-1  
while ( expression );
```

*statement-2*

When the keyword `do` is encountered, *statement-1* is executed, followed by an evaluation of *expression*. If the result of the expression evaluation is nonzero, *statement-1* is again executed. This process continues until the result of the expression evaluation becomes zero. The iteration is then terminated and the program execution continues from *statement-2*. If the result of the expression evaluation is zero at the very outset, the program control transfers to *statement-2*. Thus, in this case, *statement-1* is executed only once.

```
int number, digits, sum;

digits = sum = 0;
scanf("%d", &number);

do
{
    sum += number % 10;
    number /= 10;
    digits++;
}
while (number > 0);

printf("number of digits = %d sum = %d\n",
       digits, sum);
```

**Example 1**

*The present populations of two countries, Curvedland and Flatland, are 817 million and 1.088 billion respectively. Their rates of population growth are 2.1% and 1.3% respectively. Write a program that determines the number of years until the population of Curvedland will exceed that of Flatland.*

```
#include <stdio.h>
#define CURVED_POPULATION    0.817
#define FLAT_POPULATION     1.088
#define CURVED_GROWTH        1.021
#define FLAT_GROWTH          1.013

int main(void)

{
    float curvedland, flatland;
    int years;

    curvedland = CURVED_POPULATION;
    flatland = FLAT_POPULATION;
    years = 0;

    do
    {
        curvedland *= CURVED_GROWTH;
        flatland *= FLAT_GROWTH;
        years++;
    }
    while (curvedland <= flatland);

    printf("%d\n", years);

    return 0;
}
```

## for LOOP

```
for (expression-1; expression-2; expression-3)
    statement-1

statement-2
```

*Expression-1* defines the entry point of the structure and is used to perform loop initializations.

*Expression-2* provides the loop continuation condition.

*Statement-1* is any valid simple or compound C statement, whose execution is followed by an evaluation of *expression-3*.

*Statement-1* together with *expression-3* constitute the loop body.

*Statement-1* is also referred to as the *scope* of the for statement.

*Expression-3* is the *reinitialization expression*, and is generally used to alter the value of the loop variables before the next iteration begins.

*Statement-2* is the exit point of the structure.

```
int i, sum = 0;
```

```
for (i = 1; i <= n; i++)  
    sum += i;
```

```
printf("%d\n", sum);
```

A for loop is equivalent to the following while loop:

*expression-1;*

**while** ( *expression-2* )

{

*statement-1*

*expression-3 ;*

}

*statement-2*

the preceding for loop can also be written as

```
i = 1;
for ( ; i <= n; )
{
    sum += i;
    i++;
}

printf("%d\n", sum);
```



### Example 1

*Write a program that reads a positive integer and classifies it as being deficient, perfect, or abundant.*

The *proper* divisors of a positive integer  $N$  are those integers that are less than  $N$  and divide  $N$  evenly. A positive integer is said to be a *deficient*, *perfect*, or *abundant* number as the sum of its proper divisors is less than, equal to, or greater than the number. For example, 4 is deficient since its divisors are 1 and 2, and  $1+2 < 4$ ; 6 is perfect since  $1+2+3 = 6$ ; 12 is abundant since  $1+2+3+4+6 >$

```
int main(void)
{
    int n, i, divisor_sum = 0;

    scanf("%d", &n);

    for (i=1; i < n; i++)
        if (n % i == 0) divisor_sum += i;

    if (divisor_sum < n)
        printf("%d is deficient\n", n);
    else if (divisor_sum > n)
        printf("%d is abundant\n", n);
    else
        printf("%d is perfect\n", n);

    return 0;
}
```

### Example-2

*Write a program that reads an integer  $N$  and computes  $N!$*

```
int main(void)
{
    int number, factorial;

    do
    {
        printf("nonnegative number please\n");
        scanf("%d", &number);
    }
    while (number < 0) ;

    for (factorial = 1; number > 0; number--)
        factorial *= number;

    printf("%d\n", factorial);

    return 0;
}
```

## NESTED LOOPS

```
for (rate = 10; rate <= 15; rate += 0.5)
{
    for (years = 1; years <= 5; years++)
    {
        interest = PRINCIPAL * (rate/100) * years;
        printf("%f %f %d %f\n",
               PRINCIPAL, rate, years, interest);
    }
    printf ("\n") ; /* exit point for the inner years loop */
}
```

```
for (principal = 1000; principal <= 10000; principal += 1000)
{
    for (rate = 10; rate <= 15; rate += 0.5)
    {
        for (years = 1; years <= 5; years++)
        {
            interest = principal * (rate/100) * years;
            printf("%f %f %d %f\n",
                principal, rate, years, interest);
        }
        printf ("\n") ;/* exit point for the inner years loop */
    }
    printf ("\n");/* exit point for the middle rate loop */
}
```

### Example 1

*Four different tests are given to a class of  $n$  students. The test data is arranged so that each student's ID is followed by the student's scores in the four tests. Write a program that calculates the average score of every student and the class average over all tests.*

```
#include <stdio.h>
#define TOTAL_STUDENTS 10
#define TOTAL_TESTS 4

int main(void)
{
    int student_id, score;
    float total_score, grand_total;
    int i, j;

    grand_total = 0;

    for (i = 1; i <= TOTAL_STUDENTS; i++)
    {
        total_score = 0;
        scanf("%d", &student_id);

        for (j = 1; j <= TOTAL_TESTS; j++)
        {
            scanf("%d", &score);
            total_score += score;
        }

        printf("%d %f\n",
               student_id, total_score/TOTAL_TESTS);
        grand_total += total_score;
    }

    printf("Class Average = %f\n",
           grand_total/(TOTAL_STUDENTS*TOTAL_TESTS));

    return 0;
}
```



### Example 2

Three positive integers  $a$ ,  $b$ , and  $c$ , such that  $a < b < c$ , form a Pythagorean triplet if  $a^2 + b^2 = c^2$ . Write a program that generates all Pythagorean triplets  $a$ ,  $b$ ,  $c$ , where  $a, b \leq 25$ .

```
#include <math.h>
#include <stdio.h>
#define LIMIT 25

int main(void)
{
    int a, b, c, c_sqr;

    for (a =1; a < LIMIT; a++)
        for (b = a+1; b <= LIMIT; b++)
        {
            c_sqr = a * a + b * b;
            c = sqrt (c_sqr) ; /* truncate fraction */
            if (c * c == c_sqr)
                printf ("%d %d %d\n", a, b, c) ;
        }

    return 0;
}
```

## LOOP INTERRUPTION

It is sometimes desirable to control loop exits other than by testing a loop termination condition at the top or the bottom of the loop.

Similarly, it is sometimes desirable that a particular iteration be interrupted without exiting the loop.

## **break Statement**

When a `break` statement of the form

```
break;
```

is encountered within a loop body, the execution of the loop body is interrupted, and the program control transfers to the exit point of the loop.


```
while (1)
{
    printf("principal, rate, and years of investment? ");

    if (scanf("%f %f %d", Sprincipal, &rate, Syears) == EOF)
        break;

    interest = principal * rate / 100 * years;

    printf("interest = %f\n", interest);
}
```

*sum of prime numbers between 10 and 100*

```
for (i = 10; i <= 100; i++)  
{  
    /* check if i is prime */  
  
    for (j = 2; j <= sqrt(i); j++)  
        if (i % j == 0) /* i is not prime */  
            break;  
     breaks which loop?  
  
    if (j > sqrt (i)) /* i is prime */  
        sum += i ;  
}
```

## **continue Statement**

The `continue` statement is another loop interruption statement, but unlike `break`, it does not terminate a loop; it only interrupts a particular iteration. The `continue` statement is of the form

```
continue;
```

When a `continue` statement is encountered within the loop body of a `while` or `do-while` loop, all the remaining statements in the loop body following the `continue` statement are skipped and the loop continuation condition is evaluated next. Thus, the repetition behaves as if the last statement of the loop body has just been completed. In the case of a `continue` in the body of a `for` loop, any statements following the `continue` in the scope of the particular `for` statement are skipped, and the reinitialization expression (third expression) is evaluated next. The execution of the repetition continues thereafter as normal.

```
for (i = 1; i <= n; i++)  
{  
    if (i % 5 == 0)  
        continue;  
    sum += i;  
}
```



```
i = 1;
while (i <= n)
{
    if (i % 5 == 0) continue;
    sum += i;
    i++;
}
```

## NULL STATEMENT

C permits a statement consisting of a solitary semicolon to be placed wherever a program statement can appear. This statement, known as the *null* statement, is of the form

;

Execution of a null statement has no effect, and hence may seem quite useless. Its use, however, becomes necessary when, although no action is desired, the language syntax requires a statement. Consider, for example, the following iterative statement:

```
for (count=0; getchar() != EOF; count++)  
    ;
```

## COMMA OPERATOR

```
int i;  
float x;
```

the expressions

```
i = 1
```

and

```
x = i + 1
```

can be combined by the comma operator into a single expression as

```
i = 1, x = i + 1
```

The left operand of the comma operator assigns 1 to i, and is discarded. The right operand is then evaluated, and its type (float) and value (2) become the type and value of the compound expression. Similarly, the assignment statements

```
t = x;  
x = y;  
y = t;
```

that interchange the values of  $x$  and  $y$ , can be combined into a single statement as

```
t = x, x = y, y = t;
```

The assignments are made from left to right; first  $x$  is assigned to  $t$ , then  $y$  to  $x$ , and finally  $t$  to  $y$ .

When writing a `for` loop, very often, more than one variable requires initialization. Similarly, sometimes one likes to have more than one variable controlling a `for` loop, and all these loop variables require reinitialization before the next iteration begins. The comma operator is specially useful in such situations as it can group several expressions into a single expression. For example, the `for` statement

```
for (x = .85, y = 1.05, z = 0;  
     x <= y; x *= 1.06, y *= 1.04) z++;
```

```
while ((c = getchar()) != EOF) putchar(c);
```

**can be rewritten as**

```
while (c = getchar(), c != EOF) putchar(c);
```

**separating the reading of the character from testing the end-of-file.**