Review: Algorithms & Programming (Part 1)

Intro, Variables, Operators, Control Structures, Loops

Assoc. Prof. Dr. Fatih ABUT

Software

- The act of designing and perfecting a **program** is referred to as programming. More recently, the techniques of design have become formalized in the study of software engineering.
- Software takes many forms and uses:
  - Operating systems
  - Application programs
    - Editors, word processors
    - Email, ftp, communications
    - Engineering, Science, Social Science, Finance …..
  - Data structures (protocols)
  - File systems

Compilers and Linkers

- Machine code is the representation of code that a compiler generates by processing a source code file.
- Note: Different compilers for different OSs.
- A linker is typically used to generate an executable file by linking object files together.
- Object files often also contain data for use by the code at runtime, relocation information, program symbols (names of variables and functions) for linking and/or debugging purposes, and other debugging information.

Programming paradigms and languages

- Many different programming languages have been developed (including only a few):
  - C Pascal Modula
  - PL/I COBOL Ada SQL
  - LISP Miranda Simula Prolog
  - Java C++ SmallTalk
- These languages are often grouped into conceptual paradigms (ways of thinking and planning):
  - Procedural Functional Declarative
  - Logic Object Oriented ... and others

C is an example of a **strongly typed**, **procedural** programming language.

It is sometimes also called a *systems* programming language.

Data types in C

- Only really four basic types:
  - char
  - int (short, long, long long, unsigned)
  - float
  - double

- Sizes of these types can *vary* from one machine to another!

- No Boolean or String types!

Type	Size (bytes)
char	1
int	4
short	2
long	8
long long	8
float	4
double	8

Enumerated Types in C (1)

- C provides the `enum` as a list of named constant integer values (starting a 0 by default)
- Names in `enum` must be distinct
- Often a better alternative to `#define`

- Example

```
enum boolean { FALSE, TRUE };

enum months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP,
   OCT, NOV, DEC };
```

Enumerated Types in C (2)

```
enum Color { RED, WHITE, BLACK, YELLOW };
enum Color  my_color = RED;
```

The new type name is "`enum Color`"

Alternative style:

```
enum AColor { COLOR_RED, COLOR_WHITE,
    COLOR_BLACK, COLOR_YELLOW };
typedef  enum AColor  color_t;
color_t  my_color = COLOR_RED;
```

Overview of C Operators

+	Addition	!	Logical NOT
-	Subtraction	&&	Logical AND
*	Multiplication	||	Logical OR
/	Division	~	Bitwise NOT
%	Modulus	&	Bitwise AND
++	Increment	|	Bitwise OR
--	Decrement	<<	Bitwise Left Shifting
= =	Equality	>>	Bitwise Right Shifting
! =	Inequality	?:	Conditional Selection

Possible categorizations of C operators

- Assignment
- Arithmetic
- Relational
- Logical
- Bitwise

- Unary
- Binary
- Ternary

Assignment Operator

- The **set equal to** symbol is used to denote the concept of *assignment* of a value to a variable
  - This also means that data is being stored in RAM (usually, rarely in the CPU)
  - Examples:
    - **int N = 0 ; /* declare N and store 0 */**
    - **N = 5 ; /* Store 5 at location N, replace 0 */**

Assignment Operator

- The assignment operator must be used with care and attention to detail
  - Avoid using **=** where you intend to perform a comparison for equivalence (equality) using **==**
  - You may use = more than once in a statement
    - This may be confusing and should be avoided when it is necessary to assure clarity of codes.

- Coding standards
  - X == 5 -> not recommended
  - 5 == X -> recommended

Arithmetic Operators

- Arithmetic operators are used to express the logic of numerical operations
  - This logic may depend on data type

- The operators may be grouped as follows:
  - Addition and Subtraction : + -
  - Multiplication : *
  - Integer Division : / %
  - Floating point Division : /
  - Auto-Increment and Auto-Decrement
    - ++ and --
    - Pre- versus Post-

Arithmetic Operators : + - *

- Unary versus Binary
  - It is meaningful to say –X (negative X) so C permits use of the minus symbol (hyphen) as a **unary** operator. It also permits use of + as unary.
    - Ex. A = -3 ;
    - Clearly, multiplication (*) of numbers does not make sense as a unary operator, but we will see later that * does indeed act unarily on a specific data type
  - All operators have typical use as **binary** operators in arithmetic expression units of the general form
    - **Operand1 arith_op Operand2**

Arithmetic Operators : ++ --

- A common programming statement involves adding (or subtracting) 1 to (from) a variable used for counting
  - `N = N + 1 ;` `N = N - 1 ;`
  - The **addition** of 1 to an integer variable is called **incrementation**
  - Similarly, **subtracting** 1 from an integer variable is called **decrementation**

Arithmetic Operators : ++ --

- The C language supports two operators that automatically generate increment or decrement statements on integer variables
  - Auto-Increment ++
  - Auto-Decrement --
  - **Examples: (Equivalent statements)**

Explicit	*Post-auto*	*Pre-auto*
N = N + 1 ;	**N++ ;**	**++N ;**
N = N - 1 ;	**N-- ;**	**--N ;**

Arithmetic Operators : ++ --

- There is a very important difference between using these operators before versus after a variable symbol
  - **AFTER (POST) :**
    - **If an expression contains N++, the expression is evaluated using the value stored at the location N. After the expression is evaluated, the value at N is incremented by 1.**
  - **BEFORE (PRE) :**
    - **If an expression contains ++N, the value at N is incremented by 1 and stored at N, before any other parts of the expression are evaluated. The expression is then evaluated using the new value at N.**

Arithmetic Operators : ++ --

- Assume the declarations with initial values specified
  - **int A, B, N = 4, M = 3 ;**
- What are the final values of A, B, N and M ?
  - **A = N++ ;**
  - **B = ++M + N-- ; /* watch out ! */**
  - **A = --A ;**
  - ANSWER: **A = 3 B = 9 N = 4 M = 4**

Augmented Assignment Operators

- Operator *augmentation* involves combining two operator symbols to form a new symbol with extended meaning

- **Arithmetic Assignment** operators combine the expressiveness of arithmetic and assignment and permit abbreviation of coding

  - **+=** and **-=**
  - ***=**
  - **/=** and **%=**

  - In some cases they may lead to hardware optimization of executable code.

Augmented Assignment Operators

- Although these operations have a certain kind of elegance, they may create ambiguity.
  - However, programmers should ensure that programs have clarity.
  - **Examples:**

Longhand	*Shorthand*
`X = X + Y ;`	`X += Y ;`
`X = X * Y ;`	`X *= Y ;`
`X = X % Y ;`	`X %= Y ;`

Relational Operators

- Relational operators are used to express the concept of **comparison of two values**
  - Based on the Boolean notions of True and False

- This is vital to decision making logic where we do something – or not – based on evaluating an expression

  - **while (Age > 0) .....**

  - **if (Num <= 0) .....**

Relational Operators

- Formally, these operators are defined as
  - Equivalence (Equal to) : ==
  - Non-equivalance (Not equal to) : !=
  - Open Precursor (Less than) : <
  - Closed Precursor (Less than or equal to) : <=
  - Open Successor (Greater than) : >
  - Closed Successor (Greater than or equal to) : >=

Logical Operators

- Boolean Set Theory defines several operations that act on values 0 and 1
  - These values apply to relational expressions and also integer variables (limited to these two values)

- Complement (Not) : !
  - Unary **! (X < Y)**

- Intersection (And) : &&
  - Binary **(X < Y) && (Age > 20)**

- Union (inclusive Or) : ||
  - Binary **(X < Y) || (Age > 20)**

Logical Operators

- The
ope

- The
  -
  -

PROPOSITION
I will go to the movies if:

I have $20 in my pocket
AND I have enough gas in my car
OR it is $10 Tuesday special night
AND I have $10 in my pocket
AND I am able to walk to the movie theater

C has a Ternary Operator ! ?:

- C is one of only a few languages that contains a ternary operator, an operator that acts on three operands
- This operator is used for simplified expression of decision logic intended to provide a result
- **(A > B) ? 10 : 20**
- If it is *true* that A > B, the expression evaluates to 10 – otherwise 20.

Expressions

- Complex expressions can be constructed using the various operators seen so far
  - Such expressions must be constructed with care, taking into account the issue of data type compatibility
  - It is also important to avoid ambiguity in how the expression is to be interpreted (both by the compiler and by the programmer)

- Parentheses **( )** are often used to encapsulate sub-expression terms
  - Sub-expressions within parentheses are compiled before other terms.

Expressions

- When an expression is constructed using parenthesized sub-expressions, these sub-expressions themselves may be further broken down into parenthesized sub-sub-expressions

- This is referred to as ***nesting*** of expressions
  - Innermost nested sub-expressions are evaluated **first** by compilers (and during execution)

Expressions

- Example:

- (1 + 5) * 3 – (4 – 2) % 3

Expressions

- Example:
- (1 + 5) * 3 - (4 - 2) % 3
- (6) * 3 - (2) % 3
- 18 - 2
- 16

Expressions

- Example:

- **(1 + 5) * (3 - (4 - 2) / (5 - 1)) % 3**

Expressions

- Example:

- (1 + 5) * (3 - (4 - 2) / (5 - 1)) % 3
- (6) * (3 - (2) / (4)) % 3
- 6 * (3 - 0) % 3
- 6 * 3 % 3
- 18 % 3 = 0

C Operator Precedence

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	( )	Function call	
	[ ]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(*type*) { *list* }	Compound literal(C99)	
2	++ --	Prefix increment and decrement[note 1]	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of[note 2]	
	_Alignof	Alignment requirement(C11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13	?:	Ternary conditional[note 3]	Right-to-Left
14[note 4]	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

Question 1.

Consider the following C language program.

```
#include <stdio.h>
int main ( ) {
int X = 5;
int W = 3;

X = ++X + --W;
printf("Output 1: %d \n", X);
printf("Output 2: %d\n", W);

W *= X % 2;
printf("Output 3: %d\n", W);

X = ((4 < (3 * (5 + 6) / 5)) || 1 && (W > 0 ));
printf("Output 4: %d\n", X ) ;

printf("Output 5: %d\n", (W > X) ? 2 : 1);

return 0;
}
```

Output 1:

Output 2:

Output 3:

Output 4:

Output 5:

Question 2.

Consider the following C language program.

```
#include <stdio.h>
int main ( ) {
int X = 5;
int W;
W = X++;
printf("%d\n", X);  //Output 1
printf("%d\n", W);  //Output 2


X = ++X + --W;
printf("%d\n", X);  //Output 3
printf("%d\n", W);  //Output 4


W += X % 2;
printf("%d\n", X);  //Output 5
printf("%d\n", W);  //Output 6


X = ((4 < (3 * (5 + 6) / 5)) || 1 && (W > 0 )) ;
printf("%d\n", X ) ;  //Output 7

return 0;
}
```

Output 1: X =

Output 2: W =

Output 3: X =

Output 4: W =

Output 5: X =

Output 6: W =

Output 7: X =

Question 3.

Consider the following C language program.

```
#include <stdio.h>
int main ( ) {
int X = 10;
int W = 5;

X = ++X + --W;
printf("Output 1: %d \n", X);
printf("Output 2: %d\n", W);

W *= (X % 2) + 4.3;
printf("Output 3: %d\n", W/2);

X = ((8 < (2 * (7 + 6) / 4)) || 0 && (W < 0 ));
printf("Output 4: %d\n", X ) ;

printf("Output 5: %d\n", (W == X) ? 2 : 1);

W = X = 0;
printf("Output 6: %d\n", !(!(++W == X++)));

return 0;
}
```

Output 1:

Output 2:

Output 3:

Output 4:

Output 5:

Output 6:

Program Control Structures

- **Decision control**
  - **(Nested) if-else control structures**
  - **(Nested) switch control structure**

- **Repetition control**
  - **While and do-while control structures**
  - **The for control structure**

If-Else If-Else Structure

- If Statement is simple, use no *brace*
  - **if (condition)**
    Statement ;

- If Statement is compound, use *braces* and indentation

  - ```
    if ( condition ) {
        Statement1 ;
              ……
        StatementN ;
    }
    ```

  - Indentation <u>improves</u> the readability (hence, understanding) of the code for humans.
    ```

Review

Many styles exist – use one style consistently in program code.

```
If ( condition ) {
    T_stmts ;
} else {
    F_stmts ;
}
```

- **Selection:**
  - **if (condition1)**
    T_statement ; /* do when condition1 is True */
  - **else if (condition2)**
    T_statement ; /* do when condition2 is True */
    else
    F_statement ; /* do when all previous conds are False
    ***/**

- Once again, if using compound statements, or if placing a simple statement on a following line, use indentation to improve code readability.
  - This is an *either-or* situation – perform EITHER the True clause OR the False clause, but not both!

Multiple selection : switch

- Solution using switch :

```
printf ( “Enter operation code >” ) ;
 scanf ( “%d”, &Code ) ;
 switch ( Code ) {
    case 1 :  C = A + B ;
              break ;
    case 2 :  C = A - B ;
              break ;
    case 3 :  C = A * B ;
              break ;
    case 4 :  C = A / B ;
              break ;
    default :  printf ( “Error in input\n” ) ;
              break ;
 }
```

If - else if - else

```c
/*****************************************
 * C Program to find the student's grade
*****************************************/

#include<stdio.h> // include stdio.h

int main()
{
    float marks;
    char grade;

    printf("Enter marks: ");
    scanf("%f", &marks);

    if(marks >= 90) {
        grade = 'A';
    }
    else if(marks >= 80 && marks < 90) {
        grade = 'B';
    }
    else if(marks >= 70 && marks < 80) {
        grade = 'C';
    }
    else if(marks >= 60 && marks < 70) {
        grade = 'D';
    }
    else if(marks >= 50 && marks < 60) {
        grade = 'E';
    }
    else {
        grade = 'F';
    }

    printf("Your grade is %c", grade);

    return 0;
}

```

switch - case

```c
#include <stdio.h>

int main(){

    int Acnt = 0, Bcnt = 0, Alphacnt = 0 ;
    char Ch ;

    do
    {

      Ch = getchar();

      switch( Ch )
      {

          case 'a' :    /* lower case */
          case 'A' :    /* upper case */
                Acnt++;
                break ;

          case 'b' :    /* lower case */
          case 'B' :    /* upper case */
                Bcnt++;
                break;

           default :
               if ((Ch > 'b' && Ch <= 'z') ||
                           (Ch > 'B' && Ch <= 'Z'))  {
                   Alphacnt++;
               }

               break;
       }

    }
    while(Ch != '0');


    printf("Acnt: %d\n", Acnt);
    printf("Bcnt: %d\n", Bcnt);
    printf("Alphacnt: %d\n", Alphacnt);


    return 0;
}
```

Nested if-else control structures

- **Problem:**
  - Enter three integer values and find out the smallest one
    - All values inputted are distinct

```c
#include <stdio.h>

int main(){

int a, b, c, result;

printf("Input three different numbers: ");
scanf("%d %d %d", &a,&b,&c);

if (a < b)
{
   if(a < c) {
      result = a;
    }

   else {
      result = c;
   }
}

else  //a b'den büyük
{
   if(b < c) {
      result = b;
    }

   else{  //b c'den büyük
      result = c;
   }
}

printf("The smallest is %d", result);

}
```

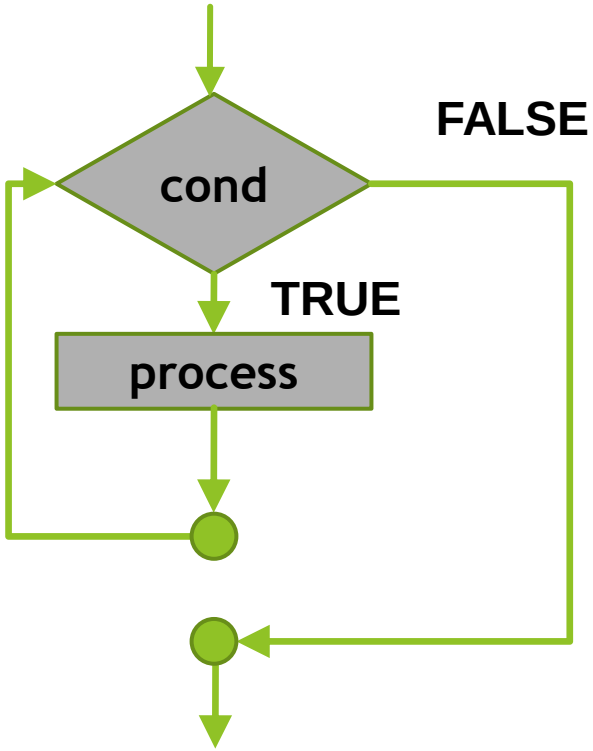
Repetition Control

- Repetition logic may be of two forms
  - Pre-condition testing : enter, or re-enter, the loop body if the condition is true.
  - Post-condition testing : enter the loop body in all cases (performing the body a minimum of once), then repeat the loop body only if the condition is true.
- C supports three forms of repetition control structures
  - while
  - do-while
  - for

Repetition : while

- while (condition_expression)
  statement ;

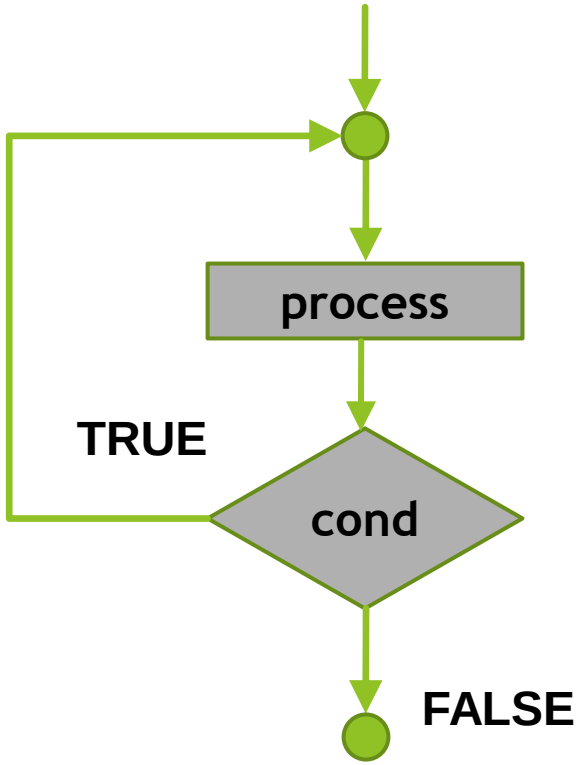
- while (condition_expression) {
  statement1 ;
  .....
  statementN ;
  }



Repetition : do-while

- do
  statement ;
  while (condition_expression) ;

- do {
  statement1 ;
  ……
  statementN ;
  } while (condition_expression) ;

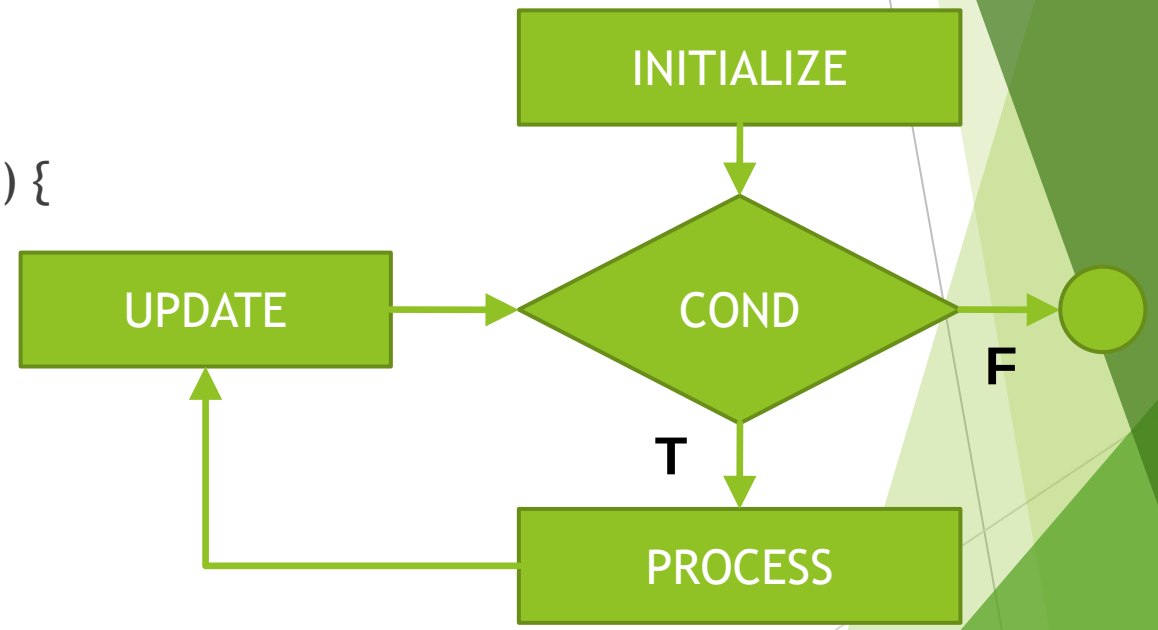


- **MUST** execute the body (process) **at least once!**

Repetition : for

- for (init_stmt ; cond_expr ; update_stmt)
  statement ;

- for (init_stmt ; cond_expr ; update_stmt) {
  statement1 ;
  .....
  statementN ;
  }



Repetition: While and Do While Examples

```c
//Power of a Number Using while Loop

#include <stdio.h>
int main()
{
    int base, exponent;

    long long result = 1;

    printf("Enter a base number: ");
    scanf("%d", &base);

    printf("Enter an exponent: ");
    scanf("%d", &exponent);

    while (exponent != 0)
    {
        result *= base;
        exponent--;
    }

    printf("Answer = %lld", result);

    return 0;
}
```

```c
//Example 2: do...while loop
// Program to add numbers until user enters zero

#include <stdio.h>

int main()
{
    double number, sum = 0;

    // loop body is executed at least once
    do
    {
        printf("Enter a number: ");
        scanf("%lf", &number);
        sum += number;
    }
    while(number != 0.0);

    printf("Sum = %.2lf",sum);


    return 0;
}
```

Repetition: For – Example
Greatest Common Divisor (GCD)

Note:
for(;;)

```c
#include <stdio.h>

int main() {

    int sayi1, sayi2, kucukSayi;
    int i, sonuc = 1;

    printf("Birinci Sayiyi Giriniz: ");
    scanf("%d", &sayi1);
    printf("Ikinci Sayiyi Giriniz: ");
    scanf("%d", &sayi2);

    if (sayi1 < sayi2)
        kucukSayi = sayi1;
    else
        kucukSayi = sayi2;

    for (i = 2; i <= kucukSayi; i++) {
        if (sayi1 % i == 0 && sayi2 % i == 0)
            sonuc = i;
    }

    printf("OBEB(%d,%d) = %d", sayi1, sayi2, sonuc);
    return 0;

}
```

Nested if-else and For control structures

- **Problem:**
  - Check whether an integer is prime or not

```c
//C program check if an integer is prime or not

    #include <stdio.h>

    int main()
    {
      int n, c;

      printf("Enter a number\n");
      scanf("%d", &n);

      if (n == 2)
        printf("Prime number.\n");
      else
      {
        for (c = 2; c <= n - 1; c++)
        {
          if (n % c == 0)
            break;
        }
        if (c != n)
          printf("Not prime.\n");
         else
           printf("Prime number.\n");
      }
      return 0;
    }
```

Break and Continue

- C defines two *instruction statements* that cause immediate, non-sequential alteration of normal sequential instruction processing

- Break Logic
  - Execution of a **break ;** statement at any location in a loop-structure causes immediate exit from the loop-structure. Break is also used to exit from a switch structure.

- Continue Logic
  - Execution of a **continue ;** statement at any location in a loop-structure causes execution to continue at the beginning of the loop structure (at the next loop iteration) while skipping the remaining statements.

Break and Continue

- Continue Logic
  - Execution of a **continue ;** statement at any location in a loop-structure causes execution to continue at the beginning of the loop structure (at the next loop iteration) while skipping the remaining statements.

-
```
for ( k = 0 ; k < 5 ; k++ ) {
    if ( k == 3 ) continue ;
    printf ( “%d, ”, k ) ;
}
```

- Produces output : ?

Break

- Break Logic
  - Execution of a **break ;** statement at any location in a **loop-structure** causes immediate exit from the loop-structure

-
```
for ( k = 0 ; k < 10 ; k++ ) {
    if ( k == 5 ) break ;
    printf ( "%d, ", k ) ;
}
```

- Produces output : ?

Break

- Break Logic
  - Execution of a **break ;** statement at any location in a **switch-structure** causes immediate exit from the switch-structure

- ```
  switch ( cond ) {
      ......
     Case 53 :  Stmt ;  .....
                  break ;
      ......
  }
  ```
  ```

Break – Example
Least Common Multiple

```c
//ekok
#include <stdio.h>


    int main(void) {

    int sayi1;
    int sayi2;
    int ekok;

    printf("1.sayin");
    scanf("%d", & sayi1);

    printf("2.sayin");
    scanf("%d", & sayi2);

    int minMultiple = (sayi1>sayi2) ? sayi1 : sayi2;

    while (1)
    {

        if ((minMultiple % sayi1 == 0) &&
                (minMultiple % sayi2 == 0)) {
        ekok = minMultiple;
        break;
        }

        minMultiple++;
    }

    printf("EKOK %d\n", ekok);


    return 0;
    }
```