MODULE 2

Syllabus

- Basic structure of C program:
  - Character set, Tokens, Identifiers in C, Variables and Data Types, Constants, Console IO Operations, printf and scanf
- Operators and Expressions:
  - Expressions and Arithmetic Operators, Relational and Logical Operators, Conditional operator, size of operator, Assignment operators and Bitwise Operators. Operators Precedence
- Control Flow Statements:
  - If Statement, Switch Statement, Unconditional Branching using goto statement, While Loop, Do While Loop, For Loop, Break and Continue statements.(Simple programs covering control flow)

- **History of C Programming Language**
  - C programming language was developed in 1972 by Dennis Ritchie at AT&T Bell laboratories, U.S.A.
  - C is structured, high level, machine independent language

- **Character set**
  - C language supports a total of 256 characters.
  - The characters in C are grouped into the following
    - Letters: a-z, A-Z
    - Digits: 0-9
    - Special characters: Some special symbols are

.	,	:	;	?	‘	“	|	/	\	~	_	$	&
#	%	*	-	+	<	>	(	)	[	]	{	}	

    - White spaces
      - Blank space, Horizontal tab, Carriage return, Newline, Form feed

- **Tokens**
  - Every smallest individual unit of a C program is called token.
  - Every instruction in a C program is a collection of tokens.
  - C has six types of tokens
    - Keyword
    - Identifiers
    - Constants
    - Strings
    - Special Symbols
    - Operators

  - **Keywords**
    - These are specific reserved words in C.
    - All keywords have fixed meanings and that cannot be changed.
    - These meanings have already been explained to the C compiler.
    - It is the basic building blocks for program statements.
    - Keywords are always in lowercase
    - There are total of 32 keywords in C

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

- **Identifiers**
  - These are user-defined names
  - Underscore character, uppercase letters and lowercase letters are permitted.
  - Identifiers refer to the names of variables, functions, arrays, pointers, structures etc.
  - **Rules for Identifiers**
    1. Identifiers consist of only alphabet, digits or underscore.
    2. First character must be an alphabet or an underscore.
    3. Only first 31 characters are significant
    4. Keywords should not be used as identifiers
    5. Must not contain white space
  - Examples:

    Valid Identifiers: mark1, C_mark, _mark1

    Invalid Identifiers: 1mark, mark$, #mark
- **Constants**
  - These are fixed values that do not change during program execution.
  - Different types of constants are
    - Integer Constants
    - Real(floating point) Constants
    - Single Character Constants
    - String Constants
    - Backslash Character Constants(Escape Sequences)

    - **Integer Constants**
      - It is a sequence of digits.
      - There are three types of integers
        - **Decimal integer**
          - It is a set of digits, 0 through 9, preceded by an optional – or + sign.
          - Examples: 57 -125 +187
        - **Octal integer**
          - Any combination of digits from the set 0 through 7 with a leading 0.
          - Examples: 025 0124
        - **Hexadecimal integer**
          - A sequence of digits preceded by 0x or 0X is considered as hexadecimal integer.
          - They may also include alphabets A through F (or a through f). The letters A through F represent the numbers 10 through 15.
          - Examples: 0X5 0x3E 0Xabc
    - **Real(floating point) Constants**
      - These constants have a whole number part followed by a fractional part
      - These may be written in one of the two forms
        - Fractional form
          - At least one digit before and after the decimal point.
          - The number may be either positive or negative
          - Example: 0.0045 -0.23 254.37 +125.0

- Exponent form
  - It consists of 2 parts. Mantissa and Exponent
  - The exponent is an integer number with optional + or - sign
  - Example:
    - 215.65 can be written as 2.1565E2. Here E2 means multiply by 10^2.
    - Here Mantissa=2.1565 and Exponent=2

- **Single Character Constants**
  - A Single Character constant represent a single character which is enclosed in a pair of single quote marks.
  - Examples: '3' 'b' ';'

- **String Constants**
  - It is a set of characters surrounded by double quotes.
  - The characters in a string constant sequence may be an alphabet, number, special characters and blank space.
  - Examples: "HELLO" "321" "?ab!"
  - Each string will be ends with special character '\0'

- **Backslash Character Constants(Escape Sequences)**
  - Backslash character constants are special characters used in output functions.
  - It is the combination of 2 characters.

Constant	Meaning
\a	Bell
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab

Constant	Meaning
\v	Vertical tab
\'	Single quote
\"	Double quote
\?	Question mark
\\	Back slash
\0	Null

- **Strings**
  - It is a set of characters surrounded by double quotes.
  - The characters in a string constant sequence may be an alphabet, number, special characters and blank space.
  - Examples: "HELLO" "321" "?ab!"
  - Each string will be ends with special character '\0'.
- **Special Symbols**
  - Some special symbols are

Symbol	Name
[ ]	Bracket
( )	Parentheses
{ }	Braces
,	Comma
;	Semicolon
*	Asterisk
#	Hash
=	Equal to

- **Operators**
  - Operators are divided into
    - Arithmetic operators
    - Assignment operators
    - Increment and Decrement operators
    - Relational operators
    - Logical operators
    - Bitwise operators
    - Shift operators
    - Conditional operator
    - Special operators

  - **Arithmetic operators**

Operator	Meaning
+	Addition/Unary plus
-	Subtraction/Unary minus
*	Multiplication
/	Division
%	Modulo division

    - **Binary operators** require two operands to operate up on.
    - **Unary operators** require only one operand.
      - Examples:
```
c=a+b;   //here + is a binary operator with 2 operands
c=-a;    //here - is a unary operator with 1 operand
```
    - Arithmetic operations are of 3 type
      - **Integer Arithmetic**: Both operands are integers. The result is also integer.
      - **Real Arithmetic**: Both operands are real numbers. The result is also real.
      - **Mixed mode Arithmetic**: Here one operand is integer and the other one is real. The result is always a real number.
      - Examples:
```
20/10=2          //integer arithmetic
25.0/10.0=2.5    //real arithmetic
25/10.0 = 2.5    //mixed mode arithmetic
```
    - Integer division truncates fractional part
    - Modulo division(%) produces the remainder of an integer division
    - The sign of modulo division result is the sign of first operand.
    - % cannot be used on floating point numbers
      - Examples:
```
int a=7,b=3,c,d,e;
c=a/b;    //now c is 2
d=a%b;    //now d is 1
e=-a%b;   //now e is -1
```

  - **Assignment operators**

Operator	Meaning
=	Assignment
+=	Assign sum
-=	Assign difference

Operator	Meaning
*=	Assign product
/=	Assign quotient
%=	Assign Modulo

    - a+=b means a=a+b

- **Increment and Decrement operators**

Operator	Meaning
++	Increment operator. Add one to the operand
--	Decrement operator. Subtract one from the operand

  - Both are unary operators
  - ++a and a++ will increment the value of a by one.
  - Examples: a=10;
    b=++a; //now a=b=11. First increment a by 1 and then assign it to b.
    c=a++; //now c=11 and a=12. First assignment, then increment

- **Relational operators**

Operator	Meaning
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
==	Equal to
!=	Not equal to

  - The result of relational expression is either **true** or **false**
  - When arithmetic expressions are used on either side of a relational operator, arithmetic expressions will be executed first and then result is compared.
  - Example: **if(age>=40){ }**
    If the age is 40 and above, then this relation will return true. Otherwise it is false.

- **Logical operators**

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

  - These are used when we want to test more than one condition and make decisions.
  - Example1: **if(age>=40 && salary>50000) { }**
    There are 2 conditions. If both are satisfied, then the condition is true and execute corresponding block of codes.
  - Example2: **if(age>=40 || salary>50000) { }**
    There are 2 conditions. If anyone satisfies, then the condition is true and execute corresponding block of codes.
  - Example3: **if(!(age>=40)) { }**
    First take the relation and find the complement. Here the condition is true if age is less than 40.

- **Bitwise operators**

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR

  - These manipulate the data at bit level
  - These operators cannot be applied on float or double numbers.

- Example1: 9&5
  - First take the binary form of 9 and 5.
  - Then perform bitwise AND operation

```
1 0 0 1  &
0 1 0 1
---------
0 0 0 1      ➔ The answer is 1
```

- Example2: 9|5
  - First take the binary form of 9 and 5.
  - Then perform bitwise OR operation

```
1 0 0 1  |
0 1 0 1
---------
1 1 0 1      ➔ The answer is 13
```

- Example3: 9^5
  - First take the binary form of 9 and 5.
  - Then perform bitwise XOR operation

```
1 0 0 1  ^
0 1 0 1
---------
1 1 0 0      ➔ The answer is 12
```

- **Shift operators**

Operator	Meaning
<<	Left shift
>>	Right shift

- These manipulate the data at bit level
- These operators cannot be applied on float or double numbers.
- The left-shift and right-shift operators are equivalent to multiplication and division by 2 respectively
- Example1: 12<<2
  - Write the binary form of the digit in 16 bits.

0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0

  - Then shift two position to the left

0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0

  - Now the number becomes 48
- Example2: 12>>1
  - Write the binary form of the digit in 16 bits.

0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0

  - Then shift one position to the right

0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0

  - Now the number becomes 6
- The left shift and right shift operators should not be used for negative numbers

- **Conditional operator**
  - ? : is the conditional operator.
  - Syntax: expression1 **?** expression 2 **:** expression 3 ;
    expression1, expression2 and expression3 are the three expressions.
    If expression1 is true, then expression2 is evaluated. Otherwise expression3 is evaluated.
  - Example: a=10, b=20;
    x=(a>b)?(2*a) : (2*b);
    Here expression1 is (a>b), expression2 is (2*a) and expression3 is (2*b).
    Here expression1 is false. So expression3 is evaluated and x becomes 40

- **Special operators**

Operator	Meaning
.	Direct component selector
→	Indirect component selector
,	Comma operator

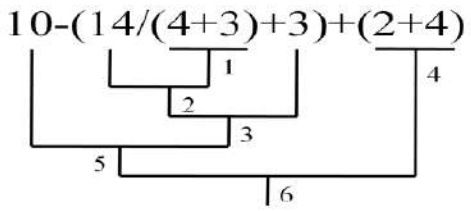
Operator	Meaning
sizeof	sizeof operator
*	Pointer operator
&	Pointer operator

  - Comma operator can be used to link related expressions together. It evaluates the expressions from left to right and the value of the rightmost expression is the value of the combined expression.
    - Example: value=(a=10, b=20, a+b);
      First assigns 10 to a, then assigns 20 to b and finally assigns 10+20=30 to value.
  - **sizeof** operator returns the number of bytes the operand occupies.
    - Example: m=sizeof(sum); // m will be the size of the variable sum

- **Operator precedence**
  - It is the priority in which the operations in an arithmetic statement are performed.

Operator	Description	Associativity	Rank
( )	Function call	Left to right	1
[ ]	Array element reference	Left to right	1
+	Unary plus	Right to left	2
-	Unary minus	Right to left	2
++	Increment	Right to left	2
--	Decrement	Right to left	2
!	Logical negation	Right to left	2
~	Ones complement	Right to left	2
*	Pointer reference	Right to left	2
&	Address	Right to left	2
sizeof	Size of an object	Right to left	2
(type)	Type casting	Right to left	2
*	Multiplication	Left to right	3
/	Division	Left to right	3
%	Modulus	Left to right	3
+	Addition	Left to right	4
-	Subtraction	Left to right	4
<<	Left shift	Left to right	5
>>	Right shift	Left to right	5
<	Less than	Left to right	6
<=	Less than or equal to	Left to right	6
>	Greater than	Left to right	6
>=	Greater than or equal to	Left to right	6
==	Equality	Left to right	7
!=	Inequality	Left to right	7
&	Bitwise AND	Left to right	8
^	Bitwise XOR	Left to right	9
|	Bitwise OR	Left to right	10
&&	Logical AND	Left to right	11
||	Logical OR	Left to right	12
?:	Conditional expression	Right to left	13
=,+=, -=, *=,/=, %=, &=,|=,^=, <<=,>>=	Assignment operators	Right to left	14
,	Comma operator	Left to right	15

- **Arithmetic Expressions**
  - It is a combination of variables, constants and operators arranged as per the syntax of the language.
  - An expression is evaluated using an assignment statement.
  - Example: x=a+b/c; //Here a+b/c is an expression
  - Rules for Evaluation of Expression
    - First, parenthesized sub expression from left to right is evaluated.
    - If parentheses are nested, the evaluation begins with the innermost sub-expressions
    - The precedence rule is applied in determining the order of application of operators in evaluating sub expressions.
    - The associativity rule is applied when two or more operators of the same precedence level appear in a sub expression
    - Arithmetic expressions are evaluated from left to right using the rule of precedence
    - When parentheses are used, the expressions within parentheses assume highest priority
  - Example: x = 10-(14/(4+3)+3)+(2+4)



- **Data Types**
  - Data types are used to specify what kind of value can be stored in a variable.
  - The memory size and type of the value of a variable are determined by the variable data type.
  - Data types are classified as follows
    - Primary data types (Basic/Fundamental/Predefined data types)
    - Derived data types (Secondary data types)
    - User defined data types (Enumerated data types)
  - **Primary data types (Basic/Fundamental/Predefined data types)**
    - **Character**
      - Signed Character/Character
      - Unsigned Character
    - **Integer**
      - Signed Integer/Integer
      - Unsigned Integer
      - Signed Short Integer/Short Integer
      - Unsigned Short Integer
      - Signed Long Integer/ Long Integer
      - Unsigned Long Integer
      - Signed Long Long Integer/ Long Long Integer
      - Unsigned Long Long Integer
    - **Floating point**
      - Floating point
      - Double precision floating point
      - Extended Double precision floating point
    - **Void**

Data type	Keyword	Memory size	Range of values	Format specifier
Character/ Signed Character	char/ signed char	1 Byte	-128 to +127	%c
Unsigned Character	unsigned char	1 Byte	0 to 255	%c
Integer/ Signed Integer	int/ signed int	2 Bytes	-32768 to +32767	%d, %i
Unsigned Integer	unsigned int	2 Bytes	0 to 65535	%u
Short Integer/ Signed Short Integer	short int/ signed short int	1 Byte	-128 to +127	%hd, %hi
Unsigned Short Integer	unsigned short int	1 Byte	0 to 255	%hu
Long Integer/ Signed Long Integer	long int/ signed long int	4 Bytes	-2^{31} to $+2^{31}-1$	%ld, %li
Unsigned Long Integer	unsigned long int	4 Bytes	0 to $2^{32}-1$	%lu
Long Long Integer/ Signed Long Long Integer	long long int/ signed long long int	8 Bytes	-2^{63} to $+2^{63}-1$	%lld, %lli
Unsigned Long Long Integer	unsigned long long int	8 Bytes	0 to $2^{64}-1$	%llu
Floating point	float	4 Bytes	3.4e-38 to 3.4e+38	%f, %e
Double precision floating point	double	8 Bytes	1.7e-308 to 1.7e+308	%lf
Extended Double precision floating point	long double	10 Bytes	3.4e-4932 to 1.1e-4932	%Lf
Void	void			

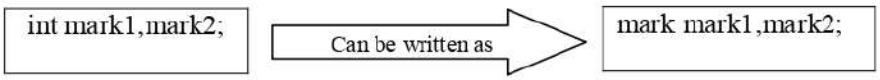
*Size of each data types vary based on machines

- Precedence rules decides the order in which different operators are applied
- Associativity rule decides the order in which multiple occurrences of the same level operator are applied.
- Floating point numbers are stored in 32 bits with 6 digits of precision
- When the accuracy provided by the float number is not sufficient, the type double can be used. It provides 14 digits of precision.
- **void data type:**
  - The void data type means nothing or no value.
  - It is used to specify a function which does not return any value

- **Derived data types (Secondary data types)**
  - Derived data types are constructed from primary data types. They are
    - **Arrays**: It is the collection of similar data referenced by a common name.
    - **Functions**
    - **Pointers**: It is a variable that holds the memory address. This address is the location of another variable in memory.
    - **Structure**: It is a collection of variables of different data types referenced under one name
    - **Union**
      - Structure and union are same but different in memory allocation

- Defining a union is similar to defining a structure
- A union is a memory location that is shared by two or more different variables, generally of different types at different times

- **User defined data types (Enumerated data types)**
  - **typedef**: It allows users to define an identifier that would represent an existing data type
    - Syntax: typedef data_type identifier;
    - Example: typedef int mark;



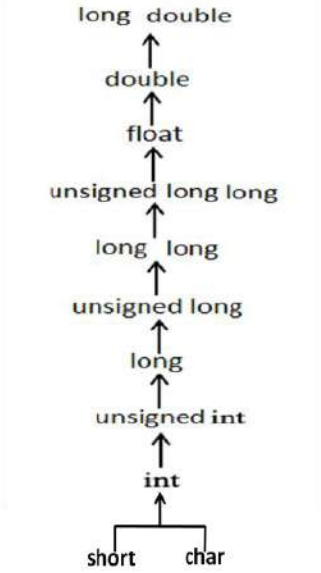
  - **Enumeration**:
    - It can be used to declare variables that can have one of the values enclosed within the braces.
      - Syntax: enum identifier{value1, value2, ... , valuen}
    - After this definition, we can declare variables to be of this new type
      - Syntax: enum identifier v1,v2, ... vn;
      - v1,v2, . . .vn can only have one of the values value1, valu2, ... valuen

- **Variables**
  - Variables are identifiers that used to store data values.
  - Variables may take different values at different times during the execution of a program
  - Examples:
    - Valid variables:- abc _max avg_mark
    - Invalid variables:- $abc 123 1ab

  - **Variable Declaration of Primary data type**
    - Syntax:
      data_type $v_1,v_2,\ldots v_n$;
      data_type is any primary data type
      $v_1,v_2,\ldots v_n$ are the names of variables
    - Declaration must ends with ;
    - Example:
      int sum;
      float mark1,mark2;

  - **Variable Initialization**
    - The process of giving initial values to variables is called initialization.
    - Syntax: data_type var_name=value;
    - Example: int x=10;
  - There are two values associated with each variable.
    - Data value(r- value): Data stored in that variable.
    - Location value(l-value): This is the address in memory at which its data value is stored.

- **Type conversion in expressions**
  - **Implicit Type Conversion**
    - C automatically converts some values to some other types. This is known as implicit type conversion.
    - Rules:
      - All **short** and **char** are automatically converted to int.
      - If one of the operand is **long double**, then the other will be converted to long double and the result will be long double.
      - Else If one of the operand is **double**, then the other will be converted to double and the result will be double.
      - Else If one of the operand is **float**, then the other will be converted to float and the result will be float.
      - Else If one of the operand is **unsigned long int**, then the other will be converted to unsigned long int and the result will be unsigned long int.
      - Else if one of the operands is **long int** and the other is **unsigned int**, then
        - If unsigned int can be converted to long int, the unsigned long int operand will be converted as such and the result will long int
        - Else both operands will be converted to unsigned long int and the result will be unsigned long int
      - Else If one of the operand is **long int**, then the other will be converted to long int and the result will be long int.
      - Else If one of the operand is **unsigned int**, then the other will be converted to unsigned int and the result will be unsigned int.



    - Final result of an expression is converted to the type of the variable in LHS.
      - Float to int causes truncation of the fractional part
      - Double to float causes rounding of digits
      - Long int to int causes dropping of the excess higher order bits.

  - **Explicit Type Conversion**
    - Syntax: **(data_type) expression;**
    - Example: x= (int) (a+b/c);
      x= (int)10.5/(int)2.0; //perform 10/2 and the result would be 5

C language provides getchar( ), getch( ) and getche() for reading single character and putchar( ), putch( ) for displaying single character on screen.

getchar() and putchar() Functions:header file (stdio.h>

1. getchar( ):

It reads a single character from input device This function is defined in <stdio.h> header file.

Syntax: var_name=getchar( );

Where var_name is of type char.

getchar( ) requires Enter key to be pressed following the character that you typed. It echoes typed character

2. putchar():

It displays or writes a single character to the standard output device

Syntax:putchar(var_name);

Where var_name is of type char.

```
#include<stdio.h>
int main()
{
char x;
x=getchar();
putchar(x);
return 0;
}
```

K

K

getch(), getche() and putch() Functions:header file <conio.h>

1. getch():

This function is used to read a character from the console but does not echo to the screen. This function is included in header file <conio.h>

Syntax: var_name=getch( );

where var_name is of type char.

getch() function is a non-buffered function. It does not use any buffer, so the entered character is immediately returned without waiting for the enter key.

The character data read by this function is directly assigned to a variable rather it goes to the memory buffer.

Another use of this function is to maintain the output on the screen until you have not to press the Enter key. getch() works only on dos like TC compiler. It does not work on a Linux platform.

2. getche():

getche( ) function is used to read a character from the console and echoes that character to the screen. This function is included in header file <conio.h>.

Syntax: var_name=getche( );

It does not use any buffer, so the entered character is immediately returned without waiting for the enter key. getche() works only on

DOS-like TC compiler. It does not work on a Linux platform.

The main difference between getch() and getche() is getch() does not echo character after reading, while getche() echoes character after reading.

3. putch():

putch() function displays or writes single character to the standard output device(i.e. stdout). This function is defined in <conio.h> header file.

Syntax: putch(var_name);

Where, var_name is of type char.

putch() does not translate linefeed characters (\n) into carriage-return/linefeed pairs. The putch() function returns the character written or EOF if an error occurs.

```
//Learnprogramo
#include<stdio.h>
#include<conio.h>
int main()
{
char x;
x=getch();
putch(x);
return 0;
}
```

K

```
#include<stdio.h>
#include<conio.h>
int main()
{
char x;
x=getche();
putch(x);
return 0;
}
```

KK

- **Console IO Operations**
  - The printf() and scanf() functions are inbuilt library functions, defined in stdio.h header file.
  - **printf()**
    - The **printf()** is used for output.
    - It prints the given statement to the console/monitor.
    - Syntax: *printf("format string",arguments);*
  - **scanf()**
    - The **scanf()** is used for input.
    - It reads the input data from the console/keyboard
    - Syntax: *scanf("format string",arguments);*
  - The **format string** can be %d (integer), %c (character), %s (string), %f (float) etc.
  - stdio.h is a header file, for standard input output functions. It activate keyboard and monitor.

- **Structure of a C Program**

C Program Sections	Description
Comments	Documentation Section
#include files	Linkage Section. #include is a pre-processor directive
#define	Definition Section
Declaration of Variables and Functions	Global declaration section

main()
{
Declaration part
Executable part
}
	Main function section

Function 1 Function 2 . Function n	Function definition section

  - Every statement in C should end with a semicolon.
  - Comments:
    - Single line comment is represented using //
    - Multiple line comments are embedded between /* and */
  - Comment lines are not executable statements and therefore anything between/* and */ is ignored by the compiler.
  - Program execution begins from main( )
  - Every program must have exactly one main function.
  - main() has no parameters

- **Control Flow Statements**
  - They are divided into
    - **Decision making and branching:**
      - If statement
      - Switch statements
    - **Looping**: Looping is deciding how many times to take a certain action
      - for loop
      - while loop
      - do-while loop

- **Jump Statement**
  - goto statement
- **Break and Continue Statements**

- **if Statement**
  - **Simple if**
    - Syntax:

```
if(condition)
{
        block of statements;
}
statement-x;
```

    - If the condition is true, then block of statements gets executed. Otherwise these statements are skipped. In both cases control is transferred to the statement-x.

  - **if-else**
    - Syntax:

```
if(condition)
{
        block of statements-1;
}
else
{
        block of statements-2;
}
statement-x;
```

    - If the condition is true, then block of statements-1 gets executed. Otherwise block of statements-2 gets executed. In both cases control is transferred to the statement-x.

  - **else-if ladder**
    - Syntax:

```
if(condition-1)
{       block of statements-1;
}
else if(condition-2)
{       block of statements-2;
}
else if(condition-3)
{       block of statements-3;
}
else
{       block of statements-4;
}
statement-x;
```

    - If the condition-1 is true, then Block of statements-1 gets executed. Otherwise check condition-2. If it is true then Block of statements-2 gets executed. Otherwise check condition-3. If it is true then Block of statements-3 gets executed. Otherwise Block of statements-4 gets executed.
    - Any one block of statements gets executed and finally control will transfer to statement-x.
    - Any number of else-if blocks is possible. Else block is optional.

- **Nested if**
  - Syntax:

```
if(condition-1)
{
        if(condition-2)                 //Nested if
        {
                block of statements-1;
        }
}
```

  - One if statement inside another if is called nested if.

- **Switch**
  - The switch statement is much like a else-if ladder statement.
  - Switch statement can be slightly more efficient and easier to read
  - Syntax:

```
switch( expression )
{
        case value1:    block1;
                        break;
        case value2:    block2;
                        break;
        case value3:    block3;
                        break;
                        . . . . .
                        . . . . .
        default     :   default block;
}
```

  - First, the expression is evaluated. It checks for matching case statements. When a match is found, the corresponding blocks gets executed. If no match is found, default block gets executed.
  - break statement causes an exit from the switch statement.
  - It is possible to nest the switch statement.
  - The case can be arranged in any order.

- **goto statement(Unconditional Jump)**
  - The goto statement allows us to transfer control of the program to the specified label.
  - Syntax:

```
goto label;
... .. ...
... .. ...
label:  statement;
```

  - Example: Write a C program to find the given number is odd or even using goto statement

```
#include<stdio.h>
void main()
{
        int num;
        printf("Enter the number: ");
        scanf("%d",&num);
```

```
        if(num%2==0)
        {       printf("%d is an even number",num);
                goto END;
        }
        printf("%d is an odd number",num);
        END: printf("\nProgram terminated!!!!");
}
```

Output

```
Enter the number: 13
13 is an odd number
Program terminated!!!!
```

- **break statement**
  - A break statement may appear inside a loop or a switch statement
  - A break statement inside a loop/switch will abort the loop/switch and transfer control to the statement following the loop/switch.
  - Example: Write a C program to read a natural number and check whether the number is prime or not

```
void main()
{       int num,i,count=0,flag=0;
        printf("Enter the number: ");
        scanf("%d",&num);
        for(i=2;i<=num/2;i++)
        {       if( num%i ==0 )
                {       flag=1;
                        break;
                }
        }
        if(flag==0)
                printf("%d is a prime number",num);
        else
                printf("%d is not a prime number",num);
}
```

Output

```
Enter the number: 13
13 is a prime number
```

- **continue statement**
  - When continue is encountered inside any loop, control automatically passes to the beginning of the loop.
  - Syntax: continue;
  - Example:

```
void main()
{       int i;
        for(i=1;i<=4;i++)
        {       if(i==2)
                        Continue;
                printf("%d  ", i);
        }
}
```

Output

```
1   3   4
```

- **for loop**
  - Syntax:
```
for( expression1; expression2; expression3)
{
        Block of statements;
}
```
  - expression1 - Initializes variables
  - expression2 - Conditional expression, as long as this condition is true, loop will keep executing.
  - expression3 - expression3 is the modifier which may be simple increment/decrement of a variable.
  - Initially evaluate expression1 then evaluate expression2. If expression2 is true, then Block of statements gets executed. During the beginning of next iteration, evaluate expression3, then check expression2. If it is still true the statements get executed again. This cycle repeats until expression2 evaluates to false.
  - During first iteration, expression1 and expression2 are executed.
  - During the subsequent iterations expression3 and expression2 are executed.

- **while loop**
  - Syntax:
```
while ( condition )
{
        Block of statements;
}
```
  - If the test condition is true, then the Block of statements get executed. After the statements have been executed, the test condition is checked again. If it is still true the statements get executed again. This cycle repeats until the test condition evaluates to false.

- **Do-while loop**
  - Syntax:
```
do
{
        Block of statements;
} while ( condition );
```
  - The Block of statements of do-while loop is executed at least once. Only then, the condition is evaluated. If it is true, then the statements get executed again. This cycle repeats until the test condition evaluates to false.

Example Programs

1. Write a C program to display "Hello World"

Program

```
#include<stdio.h>
void main()
{
    printf("Hello World");
}
```

Output

```
Hello World
```

2. Write a C program to read two numbers, add them and display their sum

Program

```
#include<stdio.h>
void main()
{
    int a,b,sum;
    printf("Enter two numbers: ");
    scanf("%d%d",&a,&b);
    sum=a+b;
    printf("%d + %d = %d",a,b,sum);
}
```

Output

```
Enter two numbers: 10    20
10 + 20 = 30
```

3. Write a C program to read the radius of a circle, calculate its area and display it

Program

```
#include<stdio.h>
void main()
{
    float radius,area;
    printf("Enter the radius of the circle: ");
    scanf("%f",&radius);
    area=3.14 * radius * radius;
    printf("Area of the circle = %f",area);
}
```

Output

```
Enter the radius of the circle: 10
Area of the circle = 314.000000
```

4. Write a C program to Evaluate the arithmetic expression ((a -b / c * d + e) * (f +g)) and display its solution.

Program

```
//((a -b / c * d + e) * (f + g))
//((20-10/ 2 * 3 + 4) * (1 + 2))
#include<stdio.h>
void main()
{
    int a,b,c,d,e,f,g,result;
    printf("Enter a b c d e f g values: ");
    scanf("%d%d%d%d%d%d%d",&a,&b,&c,&d,&e,&f,&g);
    result=((a - b / c * d + e) * (f + g));
    printf("((%d - %d / %d * %d + %d) * (%d + %d)) = %d",a,b,c,d,e,f,g,result);
}
```

Output

```
Enter a b c d e f g values: 20   10   2   3   4   1   2
((20 - 10 / 2 * 3 + 4) * (1 + 2)) = 27
```

5. Write a C program to swap two numbers using temporary variable

Program

```
//swap two numbers using temporary variable
#include<stdio.h>
void main()
{
    int a,b,temp;
    printf("Enter two numbers: ");
    scanf("%d%d",&a,&b);
    printf("Before swapping a=%d   b=%d",a,b);
    temp=b;
    b=a;
    a=temp;
    printf("\nAfter swapping a=%d   b=%d",a,b);
}
```

Output

```
Enter two numbers: 10   20
Before swapping a=10   b=20
After swapping a=20   b=10
```

6. Write a C program to swap two numbers without using temporary variable

Program

```
//swap two numbers without using temporary variable
#include<stdio.h>
void main()
{
    int a,b;
    printf("Enter two numbers: ");
    scanf("%d%d",&a,&b);
    printf("Before swapping a=%d   b=%d",a,b);
```

```
    a=a+b;
    b=a-b;
    a=a-b;
    printf("\nAfter swapping a=%d   b=%d",a,b);
}
```

Output

```
Enter two numbers: 10   20
Before swapping a=10   b=20
After swapping a=20   b=10
```

7. Write a C program to read 2 integer values and find the largest among them.

Program

```
//find the largest of two numbers
#include<stdio.h>
void main()
{
    int n1,n2;
    printf("Enter two number: ");
    scanf("%d%d",&n1,&n2);
    if(n1>n2)
            printf("largest number=%d",n1);
    else
            printf("largest number=%d",n2);
}
```

Output

```
Enter two number: 20    10
largest number=20
```

8. Write a C program to find the largest among two numbers using conditional operator.

Program

```
//largest among two numbers using conditional operator
#include<stdio.h>
void main()
{
    int n1,n2,largest;
    printf("Enter two numbers: ");
    scanf("%d%d",&n1,&n2);
    largest=(n1>n2)?n1:n2;
    printf("Largest among %d and %d is %d",n1,n2,largest);
}
```

Output

```
Enter two numbers: 10   20
Largest among 10 and 20 is 20
```

9. Write a C program to read 3 integer values and find the largest among them.

Program

```
//Largest among 3 numbers
#include<stdio.h>
void main()
{
    int num1,num2,num3,largest;
    printf("Enter three numbers: ");
    scanf("%d%d%d",&num1,&num2,&num3);
    if(num1>num2)
    {
        if(num1>num3)
            largest=num1;
        else
            largest=num3;
    }
    else            //num2>=num1
    {
        if(num2>num3)
            largest=num2;
        else
            largest=num3;
    }
    printf("The largest number is %d",largest);
}
```

Output

```
Enter three numbers: 2    1    3
The largest number is 3
```

10. Write a C program to read 3 integer values and find the largest among them using conditional operator.

Program

```
//Largest among three numbers using conditional operator
#include<stdio.h>
void main()
{
    int n1,n2,n3,largest;
    printf("Enter three numbers: ");
    scanf("%d%d%d",&n1,&n2,&n3);
    largest=(n1>n2)?((n1>n3)?n1:n3):((n2>n3)?n2:n3);
    printf("Largest=%d",largest);
}
```

Output

```
Enter three numbers: 20 10 30
Largest=30
```