

UNIT-1

1. What is an algorithm? Explain?

Algorithm:

An algorithm is a finite set of instructions (steps) each of which may require one or more operations. Every operation may be characterized as either a simple or complex.

(Or)

An algorithm is a set of steps for solving a particular problem.

Properties of algorithm:

1. Finiteness
2. Definiteness
3. Input
4. Output
5. Effectiveness

1. Finiteness: The algorithm must terminate after a finite number of steps.

2. Definiteness: Each instruction in the algorithm must be clear. Each operation must be clear and must be definite meaning that it must be perfectly clear what should be done.

3. Input: An algorithm must have zero or more inputs.

4. Output: It must produce at least one or more outputs.

5. Effectiveness: Each operation should be effective i.e. the operation must be able to carry out in finite amount of time.

Example:

Algorithm to add 3 numbers

BEGIN

STEP 1: Read number1, number2, number3

STEP 2: sum= number1+ number2+ number3

STEP 3: Average= sum/3

STEP 4: Write Average

END

Types of algorithm:

An algorithm can have different patterns in it.

1. Sequential: A sequential pattern is one where different steps occur in a sequence.

2. Conditional: A Conditional pattern is one where different steps are executed based on a condition.





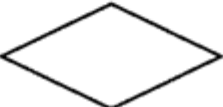
3. Iterational: In an iterational pattern, a task is repeated more than once. In Programming languages an iterational pattern is implemented using loops. An iterational construct is also known as 'repetitive' construct.

2. Define a flowchart? List and explain the various symbols used in flowchart with figures.

Flowchart:

A flowchart can be used to pictorially represent an algorithm. Flowchart is one of the ways software engineers document an algorithm.

Flowchart symbols:

Name	Symbol	Function
Start/End		Used to markup the starting and ending point
Arrows		Used for connection
Input/Output		Used for input and output information
Process		Used to represent single step
Decision		Used for branching or decision making

3. With an example, explain the structure of C program.

Basic Structure of a C Program:

Documentation section	
Link section	
Definition section	
Global declaration section	
main () Function section	
{	
	Declaration part
	Executable part
}	
Subprogram section	
Function 1	
Function 2	
.....	
.....	
Function n	
	(User defined functions)

1. Documentation section: The documentation section consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later.

2. Link section: The link section provides instructions to the compiler to link functions from the system library.

3. Definition section: The definition section defines all symbolic constants.

4. Global declaration section: There are some variables that are used in more than one function. Such variables are called global variables and are declared in the global declaration section that is outside of all the functions. This section also declares all the user-defined functions

5. main () function section : Every C program must have one main function section. This section contains two parts; declaration part and executable part

Declaration part: The declaration part declares all the variables used in the executable part.

Executable part: There is at least one statement in the executable part. These two parts must appear between the opening and closing braces. The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function is the logical end of the program. All statements in the declaration and executable part end with a semicolon.

6. Subprogram section: The subprogram section contains all the user-defined functions that are called in the main () function. User-defined functions are generally placed immediately after the main () function, although they may appear in any order.

Sample C Program:

```
#include<stdio.h> <-----Preprocessing Directive
```

```
void main()
```

```
{ <-----Start of a Program
```

```
/* .....body of main( )..... */
```

```
Printf("Learn at every moment");
```

```
} <-----End of a Program
```

- In C many library functions are grouped category-wise and stored in different files known as header files. Ex. stdio.h→standard input output header file
- To use the functions defined in the header file that need to be included in the program
- This can be achieved by the preprocessing directive “#include”
- “#include” includes the content of header file(stdio.h) at the beginning of program.

4. Explain different steps involved in software development method.

Software Development Method:

Steps involved in software development method:

1. Specify the problem requirements
2. Analyze the problem
3. Design the algorithm to solve the problem
4. Implement the algorithm
5. Test and verify the completed program
6. Maintain and update the program

1. Specify the problem requirements

State the problem clearly and gain a clear understanding of what is required for its solutions and eliminate unimportant aspects

2. Analyze the problem

Identify the problem Input, Output, and Additional requirements or constraints

3. Design the algorithm to solve the problem

Develop a list of steps (called algorithm) to solve the problem and to then verify that the algorithm solves the problem as intended.

4. Implement the algorithm

Convert each algorithm step into one or more statements in a programming language

5. Test and verify the completed program

Testing the completed program to verify that it works as desired. Run the program several times using different set of data to make sure that it works correctly for every situation provided for in the algorithm.

6. Maintain and update the program

Modify a program to remove previously undetected errors and to keep it up-to-date

5. What is a C-token? Explain different types of C-tokens?

C-token:

A passage of text, individual words and punctuation marks are called as tokens. Similarly in C language the smallest individual units are known as C tokens.

C has six types of tokens and those are used to write the C programs.

1. Keywords
2. Identifiers
3. Constants
4. Strings
5. Special symbols
6. Operators

1. Keywords:

Keywords are the words whose meaning has already been explained to the C compiler. The keywords cannot be used as variable names.

Every C word is classified as either a keyword or an identifier. The keywords are also called as 'reserved words' there are only 32 keywords in C89. Those are as follows:

auto	Continue	enum	if	short	switch	Volatile
break	default	Extern	int	signed	typedef	while
case	do	float	Long	sizeof	union	
char	double	for	register	static	unsigned	
const	else	goto	return	struct	void	

2. Identifiers:

Identifiers refer to the names of variables, functions and arrays.

Rules for constructing identifiers:

1. An identifier is a sequence of letters and digits.
2. The first character must be a letter.
3. The underscore counts as a letter.

4. Upper case and lower case letters are different.

Ex: amount, case_1, data are valid. \$turn, _loan, 3loan are not valid.

3. Constants:

A constant is a fixed value that cannot be changed / altered during the execution of a program. C constants can be classified into two categories. They are:

1. Primary constants
2. Secondary constants

C has five types of primary constants. They are integer, float, character, logical and string. Here we discussed about only primary constants.

Secondary Constants are array, structure, union, pointer, etc... .

4. Strings:

A string constant is a sequence of characters enclosed in double quotes. The characters may be letters, numbers, blank spaces or special symbols.

Ex: "welcome", " ", "this is a string", "a+b=7", "B", etc...

5. Special Symbols:

Special symbols are used for special purposes. Ex: <, >, =, +, -, \$, #, &, *, %, ^, _, |, ~, ,, ., ;, :, ?, ", "'", !, (,), *, +, ,, -, /, \

6. Operators:

An operand is a data item on which operators perform the operations.

⇒ **Example:**

A+B

Here **A**, **B** are the two operands and **+** is an operator.

⇒ C operators can be classified into a number of categories. They include:

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators
7. Bitwise operators
8. Special operators

1. Arithmetic operators:-

⇒ The arithmetic operators are used for numeric calculations between two constant values.

⇒ These are called Binary operators.

⇒ The below table shows different arithmetic operators that are used in 'C' with example.

Operator	Meaning	Example
+	Addition	11+3=14
-	Subtraction	11-3=8
*	Multiplication	11*3=33
/	Division	11/3=3 (decimal part truncated)
%	Modular division (Modulo division)	11%3=2 (remainder of division)

Table: Arithmetic operators

2. Relational Operators:-

⇒ These operators are used to compare two values, to see whether they are equal to each other, unequal, or whether one is greater than the other.

⇒ These operators provide the relationship between the two values. If the relation is true then it returns a value 1, otherwise 0 for false relation.

⇒ The relation operators with example and return values are shown in below table.

Operator	Meaning	Example	Return value
>	Greater than	5>4	1
<	Less than	10<9	0
<=	Less than or equal to	10<=10	1
>=	Greater than or equal to	11>=5	1
==	Equal to	2==3	0
!=	Not equal to	3!=3	0

Table: Relational operators

3. Logical operators:-

- ⇒ The logical relationship between two expressions is checked with logical operators.
- ⇒ Using these operators two expressions can be joined.
- ⇒ After checking the condition this operators provide logical true (1), or false (0) status.
- ⇒ The logical operators with example and return values are shown in below table.

Operator	Meaning	Example	Return value
&&	Logical AND	5>3&&5<10	1
	Logical OR	8>5 8<2	1
!	Logical NOT	!(8==8)	0

Table: Logical operators

- ⇒ From the above table following rules can be followed for logical operations.
 1. The logical AND (&&) operator provides true (1) result when both expressions are true otherwise false (0).
 2. The logical OR (||) operator provides true (1) result when one of the expression is true otherwise false (0).
 3. The logical NOT (!) provides 0 if the condition is true otherwise 1.

⇒ The below table shows the logical AND, logical OR operation between two expressions.

Operands		Results	
exp1	exp2	exp1&&exp2	exp1 exp2
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

4. Assignment operator:-

⇒ Assignment operator is used to assign the result of an expression to a variable.

Syntax:-

V op = exp;

- ⇒ Where **V** is a variable, **exp** is an expression and **op** is a C binary arithmetic operator.
- ⇒ Here the operator '**op=**' is known as the "**short hand assignment operator**".
- ⇒ The statement,

V op = exp;

is equivalent to

V = V op (exp);

⇒ **EXAMPLE:**

$x += y + 1;$

This statement is same as,

$x = x + y + 1;$

⇒ The below table shows the short hand operators.

Statement with simple assignment operator	Statement with short hand assignment operator
$a = a + 1$	$a += 1$
$a = a - 1$	$a -= 1$
$a = a * (n+1)$	$a *= n+1$
$a = a / (n+1)$	$a /= n+1$
$a = a \% 1$	$a \% = 1$

⇒ The short hand assignment operator has 3 advantages.

1. What appears on the left-hand side need not be repeated and therefore it becomes easier to write.
2. The statement is more concise and easier to read.
3. The statement is more efficient.

5. Increment (++) and decrement (--) operator:-

⇒ The operator ++ adds one to its operand.

⇒ The operator – subtract one from its operand.

⇒ These operators can be used either before (prefix) or after (postfix) their operand in shown in below,

Operator	Meaning
a++	Post-increment
++a	Pre-increment
a--	Post-decrement
--a	Pre-decrement

⇒ Prefix and postfix operators have same effect if they are used in an isolated C statement.

Example:-

a++

++a

a--

--a

⇒ However, prefix and postfix operators have different effects when used in association with some other operator in a C statement.

Example:-

Assume a=5, then execution of statement,

b=++a;

will first increase the value of a to 6 and then assign that new value to b.

This effect is exactly same as,

a=a+1;

b=a;

⇒ On other hand, execution of the statement,

b=a++;

will first set the value of b to 5 and then increase the value of a to 6.

This effect is same as,

```
b=a;  
a=a+1;
```

- ⇒ The decrement operators are used in a similar way, of course, the values of a and b are decreased.

Prefix decrement:

```
b=a--;
```

This is same as,

```
a=a-1;  
b=a;
```

Post decrement:

```
b=a--;
```

This is same as,

```
b=a;  
a=a-1;
```

6. Conditional operator (Ternary operator):-

- ⇒ The conditional operator contains a condition followed by two statements or values.
⇒ If the condition is true, the first statement otherwise the second statement.
⇒ The conditional operator (?) and (:) are sometimes called ternary operator, because they take 3 arguments.
⇒ **Syntax:**

Condition? (expression1) : (expression2)

- ⇒ Here two expressions are separated by a colon (:).
⇒ If the condition is true expression1 gets evaluated otherwise expression2.
⇒ The condition always written before question mark (?).

7. Bitwise operators:-

- ⇒ Bitwise operators are similar to that of logical operator; expect that they work on binary bits.
⇒ When bitwise operator operators are used with variables. They are converted to binary numbers and then bitwise operators are applied on individual bits.
⇒ These operators work with integers and character data types. They cannot use with floating point numbers.
⇒ The below table shows different bitwise operators that are used in 'C'.

Operator	Meaning
~	One's complement
>>	Right shift
<<	Left shift
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR

Table: - Bitwise operators

One's complement operator(~):-

- ⇒ This operator operates on a single variable.
⇒ On taking one's complement of a number, all 1's present in the number are changed to 0's and all 0's are changed to 1's.

⇒ **Example:**

1. One's complement of 1010 is 0101.

2. One's complement of 65 is,

Binary equivalent for 65 is,

0000 0000 0100 0001

One's complement of this is,

1111 1111 1011 1110

Right shift operator (>>):-

⇒ This operator operates on a single variable.

⇒ It is represented by >> and it shifts each bit in the operand to the right.

⇒ The number of places the bits are shifted depends on the number following the operand.

⇒ Thus, `ch>>3` would shift all bits 3 places to right. Similarly `ch>>5` would shift all bits 5 places to right.

⇒ **Example:**

Variable `ch` contains the bit pattern 11010111, then

`ch>>1` gives, 01101011

`ch>>2` gives, 00110101

Left shift operator (<<):-

⇒ This operator operates on a single variable.

⇒ It is represented by << and it shifts each bit in the operand to the left.

⇒ The number of places the bits are shifted depends on the number following the operand.

⇒ Thus, `ch<<3` would shift all bits 3 places to left. Similarly `ch<<5` would shift all bits 5 places to left.

⇒ **Example:**

Variable `ch` contains the bit pattern 11010111, then

`ch<<1` gives, 10101110

`ch<<2` gives, 01011100

Bitwise AND operator (&):-

⇒ This operator operates on two operands. While operating upon these two operands they are compared on a bit-by-bit basis.

⇒ Hence both operands must be of the same type (either `char` or `int`).

⇒ The first operand is called original bit pattern. The second operand is called an AND mask.

⇒ The bitwise AND operator (&) operates on a pair of bits that yields a resultant bit.

⇒ The results that decided the value of the resultant bit are shown below.

First bit	Second bit	First bit & Second bit
0	0	0
0	1	0
1	0	1
1	1	1

⇒ The bitwise AND operator (&) returns 1, when both two bits are 1.

⇒ **Example:**

7	6	5	4	3	2	1	0
1	0	1	0	1	0	1	0

Original bit pattern

7	6	5	4	3	2	1	0
1	1	0	0	0	0	1	1

AND mask

7	6	5	4	3	2	1	0
1	0	0	0	0	0	1	0

Result

Bitwise OR operator (|):-

- ⇒ This operator operates on two operands. While operating upon these two operands they are compared on a bit-by-bit basis.
- ⇒ Hence both operands must be of the same type (either char or int).
- ⇒ The first operand is called original bit pattern. The second operand is called an OR mask.
- ⇒ The bitwise OR operator (|) operates on a pair of bits that yields a resultant bit.
- ⇒ The results that decided the value of the resultant bit are shown below.

First bit	Second bit	First bit Second bit
0	0	0
0	1	1
1	0	1
1	1	1

- ⇒ The bitwise OR (|) operator returns 1, when any one of the two bits or two bits are 1.

⇒ **Example:**

7	6	5	4	3	2	1	0	
1	0	1	0	1	0	1	0	Original bit pattern
7	6	5	4	3	2	1	0	
1	1	0	0	0	0	1	1	OR mask
7	6	5	4	3	2	1	0	
1	1	1	0	0	0	1	1	Result

Bitwise XOR operator (^):-

- ⇒ This operator operates on two operands. While operating upon these two operands they are compared on a bit-by-bit basis.
- ⇒ Hence both operands must be of the same type (either char or int).
- ⇒ The first operand is called original bit pattern. The second operand is called an XOR mask.
- ⇒ The bitwise XOR operator (^) operates on a pair of bits that yields a resultant bit.
- ⇒ The results that decided the value of the resultant bit are shown below.

First bit	Second bit	First bit & Second bit
0	0	0
0	1	0
1	0	1
1	1	0

- ⇒ The bitwise XOR returns 1, only if one of the two bits is 1.

⇒ **Example:**

7	6	5	4	3	2	1	0	
1	0	1	0	1	0	1	0	Original bit pattern
7	6	5	4	3	2	1	0	
1	1	0	0	0	0	1	1	XOR mask
7	6	5	4	3	2	1	0	
0	1	1	0	1	0	0	1	Result

8. Special operators:-

Comma operator:-

- ⇒ This operator is used to separate the related expressions.
- ⇒ The expressions separated by comma operator need not be included within the parenthesis.
- ⇒ **Example:**

```
a =2, b=4, c=a+b;  
(a=2, b=4, c=a+b);
```

Here first assigns the value 2 to a, then assigns 4 to b, and finally assigns 6 (i.e., 2+4) to c.

- ⇒ Some applications of comma operator are:
 - In for loops:
for (n=1,m=10; n<=m; n++,m++)
 - In while loops:
while(c=getchar(), c!= '\0')
 - Exchanging values:
t =x, x=y, y=t

Member operators (. and ->):-

- ⇒ These are used to access members of structures and unions.
- ⇒ **Example:**
var.member1; /* when var is a structure variable */
var->member2; /* when var is a pointer to structure variable*/

Unary Minus (-):-

- ⇒ Unary minus (-) is used to indicate or change the sign of a value.
- ⇒ **Example:-**
int x=-50;
int y=x;
- ⇒ Here, assigns the value of -50 to x and the value of -50 to y through x.

Address Operator(&):-

- ⇒ The address operator (&) prints the address of the variable in memory.
- ⇒ **Example:**
Printf(" address of variable: %u", &a);
Here 'a' is a variable, &a prints address of 'a'.

sizeof() Operator:-

- ⇒ The sizeof () operator gives the bytes occupied by the operand in memory.
- ⇒ The operand may be a variable, a constant, or a data type.

6. What is a variable? How to declare a variable in c?

Variable:

A variable is a named location in memory that is used to hold a value that can be modified by the program. All variables must be declared before they can be used. A declaration specifies a type and contains a list of one or more variables of that type. The general form of a variable declaration is:

Data type variable_list;

Here data type refers to any basic data type. And variable list consists one or more variables of that type.

Ex:

```
int i, j, l;
```

```
short int si;
unsigned int ui;
double balance, profit, loss;
```

7. What is a data type? Explain data types in C?

Data Type:

Data type defines the the type of data that stores in a variable. C defines five foundational data types.

1. Character (Char)
2. Integer (int)
3. Floating point (float)
4. Double floating point double)
5. Valueless (void)

These types form the basis for several other types. The size and range of these data types may vary among processor types and compilers. However, in all cases an object of type char is 1 byte. The size of an int is usually the same as the word length of the execution environment of the program.

For most 16-bit environments, such as DOS or Windows 3.1, an int is 16 bits. For most 32-bit environments, such as Windows 95/98/NT/2000, an int is 32 bits.

The exact format of floating-point values will depend upon how they are implemented. Variables of type char are generally used to hold values defined by the ASCII character set. Values outside that range may be handled differently by different compilers.

The range of float and double will depend upon the method used to represent the floating-point numbers. Standard C specifies that the minimum range for a floating-point value is $1E-37$ to $1E+37$. The minimum number of digits of precision for each floating-point type is shown in below Table.

The type void either explicitly declares a function as returning no value or creates generic pointers. Both of these uses are discussed in subsequent chapters.

Type	Size in Bits	Minimal Range
char	8	−127 to 127
unsigned char	8	0 to 255
signed char	8	−127 to 127
Int	16	−32,767 to 32,767
unsigned int	16	0 to 65,535
signed int	16	−32,767 to 32,767
short int	16	32,767 to 32,767
unsigned short int	16	0 to 65,535
signed short int	16	32,767 to 32,767
long int	32	−2,147,483,647 to 2,147,483,647
long long int	64	−($2^{63} - 1$) to $2^{63} - 1$
signed long int	32	−2,147,483,647 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295
unsigned long long int	64	$2^{64} - 1$
float	32	$1E-37$ to $1E+37$ with six digits of precision
double	64	$1E-37$ to $1E+37$ with ten digits of precision
long double	80	$1E-37$ to $1E+37$ with ten digits of precision

By default all data types are signed. Signed means that the data type is capable of storing both positive and negative values.

unsigned

unsigned restricts the data type so that it can only store positive values.

short

The short modifier reduces the size of the data type to half its regular storage capabilities. There are two types of short, signed and unsigned.

long

The data type modifier long doubles the storage capacity of the data type being used. A long can be either signed and unsigned.

long long

The long long modifier is guaranteed to be at least 64 bits. Again as with both the short and long we have two types of the long long, signed and unsigned.

Void type:

The type void was introduced in ANSI C. two normal uses of void are:

1. To specify the return type of a function when it is not returning any value.
2. Indicates any empty argument list to a function.
3. In declaration of generic pointers.

Ex: void function_name (void);

UNIT-2

1. Explain Conditional statements in C?

Conditional/ selection statements:

C supports two selection statements: if and switch. In addition, the ?: operator is an alternative to 'if' in certain circumstances.

If:

Simple if:

Syntax:

```
if (condition)
{
    Statement;
}
```

If the condition is true then the statement will be executed otherwise it skips the statement.

Ex:

```
void main
{
    int ch;
    clrscr ();
    printf ("enter the value of ch\n");
    scanf ("%d",&ch);
    if (ch>=1)
        printf ("ch is not zero");
    getch ();
}
```

Output:

```
enter the value of ch : 2
ch is not zero
```

2. if-else:

Syntax:

```
if (condition)
{
    statement;
}
else
{
    statement;
}
```

Where statement is a single or a block of statements or nothing. The else clause is optional. If the condition is true the statement immediately after the 'if' is executed. Otherwise the statement after the else is executed. Only the code associated with 'if' or the code associated with 'else' executes not both.

Ex: /* example of if-else statement*/

```
#include<stdio.h>
#include<conio.h>
void main
{
    int x=3,y;
    clrscr ();
```

```

printf ("enter the value of y\n");
scanf ("%d",&y);
if (x==y)
    printf ("x and y are equal\n");
else
    printf("x and y are not equal")
    getch ();
}

```

Output:

```

enter the value of y: 5
x and y are not equal

```

3. if-else-if ladder:

```

    if (condition)
    {
        statement;
    }
    else if (condition)
    {
        statement;
    }
    else if (condition)
    {
        statement;
    }
    ---
    ---
    ---
    else
    {
        statement;
    }

```

This sequence of statements is the most general way of writing a multi-way decision. The conditions are evaluated in order. If any condition is true, the statement associated with it is executed. And this terminates the whole chain. The code for each statement is either a single statement or a group of statements in braces.

The last else part is executed when none of the above are satisfied.

Ex: //EXAMPLE PROGRAM FOR IF-ELSE-IF LADDER

```

#include<stdio.h>
#include<conio.h>
void main
{
    int s1,s2,s3,s4,s5,per;
    clrscr ();
    printf ("enter the marks for subjects:\n");
    scanf ("%d%d%d%d%d",&s1,&s2,&s3,&s4,&s5);
    per= ((s1+s2+s3+s4+s5)/500)*100;
    printf ("RESULT IS: ");
    if (per>=75)
        printf ("DISTINCTION\n");
    else if (per>=60)
        printf("FIRST CLASS\n");
    else if (per>=50)

```

```

        printf("SECOND CLASS\n");
    else if (per>=40)
        printf("THIRD CLASS\n");
    else
        printf("FAIL\n");
    getch ();
}

```

Output:

enter the marks for subjects:

65 59 87 91 69

RESULT IS: FIRST CLASS

2. Switch():

C has a built-in multiple branch selection statement called switch. Which successively tests the value of an expression against a list of integer or char constants when a match is found the statement associated with that constant are executed.

General format of switch:

```

switch(integer expression)
{
    case constant1:
        Statement sequence;
    case constant2:
        statement sequence;
    ..
    default:
        Statement sequence;
}

```

The expression must evaluates to an integer type. Thus we can use character or integer value. But floating point expressions are not allowed. The default statement is executed if no matches are found. The default is optional. C

The switch differs from the if in that switch can only test for equality. Whereas if can evaluated only type of relational or logical expression.

No two case constants in the same switch can have identical values. But a switch enclosed by an outer switch may have case constants that are in common.

If character constants are used in the switch statement they are automatically converted into integer. The case statement is label.

Ex: //EXAMPLE OF SWITCH ()

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int i;
    clrscr();
    printf(" Enter the value of i :");
    scanf("%d ", &i );
    switch( i )
    {
        case 1:
            printf("case 1");
            break;
        case 2:
            printf("case 2");

```



```

        break;
    case 3:
        printf("case 3");
        break;
    default:
        printf("default");
        break;
}
getch();
}

```

Output:

Enter the value of : 2

Case2

2. Explain iterative statements in C?

Loop control/Iteration statements:

The loop statement allows a set of instructions to be performed repeatedly until a certain condition is reached.

Loop :

A loop is defined as a block of statements which are repeatedly executed for certain number of times.

Loop variables:

It is the variable used in the loop.

Initialization:

It is the first step in which starting and final value is assigned to the loop variable. Each time the updated value is checked by the loop itself.

Incrementation / Decrementation:

It is the numerical value added or subtracted to the variable in each round of the loop.

C supports three types of loop statements.

1. While
2. do-while
3. for

1. while:-

It is also called as entry-controlled loop.

Syntax:

```

while (condition)
{
    /*body of the loop */
}

```

The test condition is evaluated and if it is true, the body of the loop is executed. On execution of the body, test condition is repetitively checked and if it is true the body is executed. The process of execution continues until the test condition becomes false. The control is transferred out of the loop. The block of the loop may contain a single statement or a set of statements.

The working of while loop:-

Ex:- Printing First n numbers using while loop

```

#include<stdio.h>
int main()

```

```

{
    int i,n;
    printf("Enter a number");
    scanf("%d",&n);
    i=1;
    while(i<=n)
    {
        printf("%d\t",i);
        i++;
    }
    return 0;
}

```

Output:

Enter a number 5

1 2 3 4 5

2. do-while

It is also called as exit controlled loop.

Syntax:

```

do
{
    /* body of the loop */
} while ( condition );

```

The working of do-while loop:-

Ex:- Printing First n numbers using do-while loop

```
#include<stdio.h>
```

```
int main()
```

```

{
    int i,n;
    printf("Enter a number");
    scanf("%d",&n);
    i=1;
    do
    {
        printf("%d\t",i);
        i++;
    }while(i<=n);
    return 0;
}

```

Output:

Enter a number 5

1 2 3 4 5

3. for

The for loop is the simplest and most commonly used loop in c. This loop consists of three expressions .The first expression is used to initialize the index value the second is used to check whether or not the loop is to be continued again and the third to change the index value.

Syntax:

```
for (exp1; exp2; exp3)
{
    //body of loop
}
```

Exp1 is the initialization expression, exp2 is the test expression and exp3 is the update expression.

```
for (initialization; test condition; update expression)
```

The working of for loop:-

The for loop is frequently used, usually the loop will be traversed a fixed number of times.

Ex:- Printing First n numbers using for loop

```
#include<stdio.h>
int main()
{
    int i,n;
    printf("Enter a number");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
        printf("%d\t",i);
    return 0;
}
```

Output:

Enter a number 5

1 2 3 4 5

A for loop may contain multiple initialization/update expression.

```
for (i =1, sum=0; i <=10; sum +=i, i++);
```

A for loop may be used as the infinite loop.

For (; ;)

3. Define an array? Explain different types of arrays in C?

Array:

An array is a collection of similar data type that is used to allocate memory in a sequential manner.

Syntax:

```
<data type> <array name>[<size of an array>]
```

Subscript or indexing:

A subscript is property of an array that distinguishes all its stored elements because all the elements in an array having the same name (i.e. the array name). So to distinguish these, we use subscripting or indexing option.

Ex:

```
int ar[20];
```

First element will be: `int ar[0];`

Second element will be: `int ar[1];`

Third element will be: `int ar[2];`

Fourth element will be: `int ar[3];`

Fifth element will be: `int ar[4];`

Sixth element will be: `int ar[5];`

And So on.....

Last element will be: `int ar[19];`

Advantage of an array:

- Multiple elements are stored under a single unit.
- Searching is fast because all the elements are stored in a sequence.

Types of Array:

1. One Dimensional Array
2. Two Dimensional Arrays.
3. Multi Dimensional Array

One Dimensional Array

- An array has only one subscript, it is known as One Dimensional Arrays.

Declaring 1-D Arrays:

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows –

Type arrayName [arraySize];

The arraySize must be an integer constant greater than zero and type can be any valid C data type.

For example, to declare a 10-element array called balance of type double, use this statement –

double balance[10];

Here balance is a variable array which is sufficient to hold up to 10 double numbers.

Initializing 1-D Arrays:

- You can initialize an array in C either one by one or using a single statement as follows –

double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};

- The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [].
- If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write –

double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};

- You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array –

balance[4] = 50.0;

- The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above –

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

Accessing 1-D Array Elements:

- An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example –
double salary = balance[9];
- The above statement will take the 10th element from the array and assign the value to salary variable.

1. Two Dimensional Arrays

- An array has two subscripts, it is known as Two Dimensional Arrays.

Declaring 2-D Arrays:

- To declare a two-dimensional integer array of size [x][y], you would write something as follows –

type arrayName [x][y];

- Where type can be any valid C data type and arrayName will be a valid C identifier.
- A two-dimensional array can be considered as a table which will have x number of rows and y number of columns. A two-dimensional array A, which contains three rows and four columns can be shown as follows –

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

- Thus, every element in the array A is identified by an element name of the form a[i][j], where 'A' is the name of the array, and 'i' and 'j' are the subscripts that uniquely identify each element in 'a'.

Initializing 2-D Arrays:

- Multidimensional arrays may be initialized by specifying bracketed values for each row.
- Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {  
    {0, 1, 2, 3}, /* initializers for row indexed by 0 */  
    {4, 5, 6, 7}, /* initializers for row indexed by 1 */  
    {8, 9, 10, 11} /* initializers for row indexed by 2 */  
};
```

- The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example –

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing 2-D Array Elements:

- An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example –

```
int val = a[2][3];
```

- The above statement will take the 4th element from the 3rd row of the array.

2. Multi Dimensional Arrays:

An array has more than two subscripts, it is known as Multi Dimensional Arrays

Example:

```
int a[5][2][3]; //3-D Arrays
```

4. Write a C program to multiply two matrices using functions?

```
#include<stdio.h>
void display(int a[10][10],int,int);
void multiply(int a[10][10],int b[10][10],int,int,int);
int c[10][10];
int main()
{
    int a[10][10],b[10][10],m,n,p,q,i,j;
    printf("Matrix A Size");
    scanf("%d%d",&m,&n);
    printf("Matrix B Size");
    scanf("%d%d",&p,&q);
    if(n!=p)
    {
        printf("Matrix multiplication is not possible");
        return 0;
    }
    printf("Enter %d elements in to matrix A",m*n);
    for(i=0;i<m;i++)
        for(j=0;j<n;j++)
            scanf("%d",&a[i][j]);
    printf("Enter %d elements in to matrix B",p*q);
    for(i=0;i<p;i++)
        for(j=0;j<q;j++)
            scanf("%d",&b[i][j]);
    for(i=0;i<m;i++)
        for(j=0;j<q;j++)
            c[i][j]=0;
    printf("A Matrix is:\n");
    display(a,m,n);
    printf("B Matrix is:\n");
    display(b,p,q);
    multiply(a,b,m,n,q);
    printf("C Matrix is:\n");
    display(c,m,q);
    return 0;
}
void display(int a[10][10],int m,int n)
{
    int i,j;
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
            printf("%d\t",a[i][j]);
        printf("\n");
    }
}
void multiply(int a[10][10],int b[10][10],int m,int n,int q)
{
    int i,j,k;
    for(i=0;i<m;i++)
        for(j=0;j<n;j++)
            for(k=0;k<q;k++)
```

```
c[i][j]=c[i][j]+a[i][k]*b[k][j];
```

```
}
```

Output:

Matrix A Size

2 2

Matrix B Size

2 2

Enter 4 elements in to matrix A

1 2 3 4

Enter 4 elements in to matrix B

1 0 0 1

A Matrix is:

1 2

3 4

B Matrix is:

1 0

0 1

C Matrix is:

1 2

3 4

5. What is String? Explain different string handling functions in C?

String:

- A string is a collection of characters.
- A string is also called as an array of characters.
- A String must access by %s access specifier in c and c++.
- A string is always terminated with \0 (Null) character.
- Example of string: "SVCE"
- A string always recognized in double quotes.
- A string also considers space as a character.

Declaration and initialization:

- The following declaration and initialization create a string consisting of the word "Hello".
- To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

- If you follow the rule of array initialization then you can write the above statement as follows :

```
char greeting[] = "Hello";
```

- Following is the memory presentation of the above defined string in C/C++:

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0

String handling functions:-

- C supports a wide range of functions that manipulate null-terminated strings:

S.No.	Function	Purpose
1	strcpy(s1, s2);	Copies string s2 into string s1.
2	strcat(s1, s2);	Concatenates string s2 onto the end of string s1.

3	strlen(s1);	Returns the length of string s1.
4	strcmp(s1, s2);	Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	strchr(s1, ch);	Returns a pointer to the first occurrence of character ch in string s1.
6	strstr(s1, s2);	Returns a pointer to the first occurrence of string s2 in string s1.
7	strupr(s1)	Converts string into uppercase
8	strlwr (s1)	Converts string into lowercase
9	strrev(s1)	Reversing the string

- The following example uses some of the above-mentioned functions:

```
#include <stdio.h>
#include <string.h>
int main ()
{
    char str1[12] = "Hello";
    char str2[12] = "World";
    char str3[12];
    int len ;

    /* copy str1 into str3 */
    strcpy(str3, str1);
    printf("strcpy( str3, str1) : %s\n", str3 );

    /* concatenates str1 and str2 */
    strcat( str1, str2);
    printf("strcat( str1, str2): %s\n", str1 );

    /* total length of str1 after concatenation */
    len = strlen(str1);
    printf("strlen(str1) : %d\n", len );

    return 0;
}
```

Output:

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

6. Write a C program to print the prime numbers up to n.

```
#include<stdio.h>
void main()
{
    int n,i,fact,j;
    printf("Enter the Number");
    scanf("%d",&n);
    printf("Prime Numbers are: \n");
    for(i=1; i<=n; i++)
    {
        fact=0;
```



```

    for(j=1; j<=n; j++)
    {
        if(i%j==0)
            fact++;
    }
    if(fact==2)
        printf("%d ",i);
}
getch();
}

```

OUTPUT:

Enter the Number50

Prime Numbers are:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47

7. Write a C program to find whether the given number is prime or not.

```

#include <stdio.h>
int main()
{
    int n, i, flag = 0;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    for(i = 2; i <= n/2; ++i)
    {
        // condition for nonprime number
        if(n%i == 0)
        {
            flag = 1;
            break;
        }
    }
    if (n == 1)
    {
        printf("1 is neither a prime nor a composite number.");
    }
    else
    {
        if (flag == 0)
            printf("%d is a prime number.", n);
        else
            printf("%d is not a prime number.", n);
    }
    return 0;
}

```

Output

Enter a positive integer: 29

29 is a prime number.

8. Write a C program to reverse the given number.

```

#include <stdio.h>
int main()
{
    int n, reverse = 0;

```

```

printf("Enter a number to reverse\n");
scanf("%d", &n);
while (n != 0)
{
    reverse = reverse * 10;
    reverse = reverse + n%10;
    n = n/10;
}
printf("Reverse of entered number is = %d\n", reverse);
return 0;
}

```

Output:

Enter a number to reverse
12341
Reverse of entered number is = 14321

9. Write a C program to find the given number is palindrome or not

```

#include <stdio.h>
int main()
{
    int n, reverse = 0, t;
    printf("Enter a number to check if it is a palindrome or not\n");
    scanf("%d", &n);
    t = n;
    while (t != 0)
    {
        reverse = reverse * 10;
        reverse = reverse + t%10;
        t = t/10;
    }
    if (n == reverse)
        printf("%d is a palindrome number.\n", n);
    else
        printf("%d isn't a palindrome number.\n", n);
    return 0;
}

```

Output:

Enter a number to check if it is a palindrome or not
151
151 is a palindrome number.

10. Write a C program to find the given number is Armstrong or not

```

#include <stdio.h>
#include <math.h>
void main()
{
    int number, sum = 0, rem = 0, cube = 0, temp;
    printf("enter a number");
    scanf("%d", &number);
    temp = number;
    while (number != 0)
    {
        rem = number % 10;

```

```

    cube = pow(rem, 3);
    sum = sum + cube;
    number = number / 10;
}
if (sum == temp)
    printf ("The given number is armstrong number");
else
    printf ("The given numbe is not a armstrong number");
}

```

Output:

enter a number

370

The given number is armstrong number

11. Write a C program to find sum of digits of given number.

```
#include <stdio.h>
```

```

int main()
{
    int n, t, sum = 0, remainder;
    printf("Enter an integer\n");
    scanf("%d", &n);
    t = n;
    while (t != 0)
    {
        remainder = t % 10;
        sum = sum + remainder;
        t = t / 10;
    }
    printf("Sum of digits of %d = %d\n", n, sum);
    return 0;
}

```

Output:

Enter an integer

123

Sum of digits of 123 = 6

UNIT-3

1. What is a function? Explain different types of functions?

Functions:

- A function is a group of statements that together perform a task.
- Every C program has at least one function, which is main(), and all the most programs can define additional functions.

Advantage of functions:

1. Reduce the source code
2. Easy to maintain and modify
3. It can be called anywhere in the program.

Basic types of functions:

- Functions are basically two types:
 1. **Built-in / Library functions:**
 - Created by the developers and perform specific task.
 - Example: printf(), scanf(), getch(), exit(), etc.
 2. **User defined functions:**
 - Functions defined by the users according to their requirements are called user-defined functions.
 - These functions are used to break down a large program into small functions.

- Syntax:

```
return_type  function_name( parameter list )
{
    body of the function
}
```

- A function definition consists of a function header and a function body. Here are all the parts of a function –

1. **Return Type:**

- A function may return a value.
- The return_type is the data type of the value the function returns.
- Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

2. **Function Name :**

- This is the actual name of the function

3. **Parameters:**

- We may pass zero or more parameters to the functions.
- Parameters are optional; that is, a function may contain no parameters.
- The form of argument list is:

`(type1 arg1, type2 arg2,..., typen argn)`

4. **Function Body:**

- The function body contains a collection of statements that define what the function does.

Function types based on return values and arguments

- Functions are 4 types based on return values and arguments
 1. Function without return value and without argument.
 2. Function without return value and with argument.
 3. Function with return value and with argument.
 4. Function with return value and without argument.

1. Function without return value and without argument:

- In this type, no data transfers takes place between the calling function and the called function. They read data values and print result in the same block.

- **Example:**

```
#include<stdio.h>
void f_mult();//Function declaration
void f_mult( )//function definition
{
    int x,y,z;
    printf("Enter 2 numbers");
    scanf("%d%d", &x,&y);
    z=x* y;
    printf("The result is %d", z);
}
void main()
{
    f_mult();
}
```

Output:

```
Enter 2 numbers
4      5
The result is 20
```

2. Function without return value and with argument:

- In this type, data is transferred from the calling function to called function. The called function receives some data from the calling function and does not send back any values to the calling functions.

- **Example:**

```
#include<stdio.h>
void f_mult(int x, int y); //Function declaration
void f_mult(int x, int y)//function definition
{
    int z;
    z=x* y;
    printf("The result is %d", z);
}
void main()
{
    int c, d;
    printf("Enter any two number");
    scanf("%d%d", &c, &d);
    f_mult(c, d); //Function call
}
```

Output:

```
Enter any two numbers
4      5
The result is 20
```

3. Function with return value and with argument:

- In this prototype, the data is transferred between the calling function and called function. The called function receives some data from the calling function and sends back a value return to the calling function.

- **Example:**

```
#include<stdio.h>
int f_mult(int x, int y); //Function declaration
int f_mult(int x, int y) //function definition
{
    int z;
    z=x* y;
    printf("The result is %d", z);
    return(z);
}
void main()
{
    int a,b,c;
    printf("Enter any two value:");
    scanf("%d%d", &a, &b);
    c=f_mult(a, b); //Function call
}
```

Output:

Enter any two numbers

4 5

The result is 20

4. Function with return value and without argument:

- The calling function cannot pass any arguments to the called function but the called function may send some return value to the calling function.

- **Example:**

```
#include<stdio.h>
int f_mult(); //Function declaration
int f_mult() //function definition
{
    int x,y,z;
    printf("Enter any two number");

    scanf("%d%d", &x,&y);
    z=x* y;
    printf("The result is %d", z);
    return(z);
}
void main()
{
    int c;
    c=f_mult(); //Function call
    getch();
}
```

Output:

Enter any two numbers

4 5

The result is 20

2. Write short note on function declaration, function definition and function calling?

1. Function declaration (Prototype):

- A function **declaration** tells the compiler about a function's name, return type, and parameters.
- It also tells that how to call the function
- A function declaration has the following parts:

```
return_type function_name( parameter list );
```

- It's defines the function before it can be used.
- Function prototype need to be written at the beginning of the program.

2. Function definition:

- Function definition consists of actual function body.

Syntax:

```
return_type function_name( parameter list )
{
    body of the function
}
```

- A function definition consists of a function header and a function body. Here are all the parts of a function.

1. Return Type:

- A function may return a value.
- The return_type is the data type of the value the function returns.
- Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

2. Function Name :

- This is the actual name of the function

3. Parameters:

- We may pass zero or more parameters to the functions.
- Parameters are optional; that is, a function may contain no parameters.
- The form of argument list is:

```
(type1 arg1, type2 arg2,..., typen argn)
```

4. Function Body:

- The function body contains a collection of statements that define what the function does.

3. Function calling:

- To use a function, you will have to call that function to perform the defined task.

```
function_name( parameter list );
```

- When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.
- To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value.

3. Write short note on actual and formal parameters, local and global variables?

Actual and Formal parameters:

- **Actual parameters:**
 - These are the parameters transferred from the calling function to the called function.
- **Formal parameters:**
 - These are the arguments which are used in function definition.

Example:

```
#include<stdio.h>
int f_mult(int x, int y); //Function declaration
int f_mult(int x, int y) //function definition, Formal parameters, Called function
{
    int z;
    z=x* y;
    printf("The result is %d", z);
    return(z);
}
void main()
{
    int a,b,c;
    printf("Enter any two value:");
    scanf("%d%d", &a, &b);
    c=f_mult(a, b); //Function call, Actual parameters, Calling function
}
```

Output:

Enter any two numbers

4 5

The result is 20

Local and Global variables

- **Global variables:**
 - Global variable are that variable which is define at the top of the programmer (after header files)
 - We can access from anywhere.
 - There **initial values** are
 - int=0
 - float=0.000000
 - char- "blank space(non-printable char)".
 - **Scope[availability]** is through out of the program.
 - **Life-time[active in memory]** is until total program is executed.
 - They are **stored in data segment area**, which is public area. Hence they are also called as public variables.
- **Local variables:**
 - The variables that are declared inside a function are called local variables.
 - They are **stored in stack**
 - There **initial values** are garbage.
 - **Scope [availability]** is until that function or block is executed.
 - **Life-time [active in memory]** is until that function is executed.

Example:

```
#include <stdio.h>
/* global variable declaration */
int g;
int main ()
{
    /* local variable declaration */
    int a;
    /* actual initialization */
    a = 10;
    g=20;
    printf ("a = %d\n g = %d\n", a, g);
    return 0;
}
```

Output:

```
a= 10
g= 20
```

4. Explain in detail about call by value and call by reference?**Call by value (or) Pass by Value:**

- When the value is passed directly to the function it is called call by value.
- In call by value only a copy of the variable is only passed so any changes made to the formal arguments (called function) does not reflect in the actual arguments (calling function).

Example:

```
#include<stdio.h>
void swap(int,int);
void main()
{
    int x,y;
    printf("Enter two numbers:");
    scanf("%d %d",&x,&y);
    printf("\nBefore swapping : x=%d y=%d",x,y);
    swap(x,y);
    printf("\nAfter swapping :x=%d y=%d",a,b);
}
swap(int a, int b)
{
    int t;
    t=a;
    a=b;
    b=t;
}
```

Output:

```
Enter two numbers:
12    34
Before swapping:
12    34
After swapping:
12    34
```

Call by reference (or) Pass by Reference:

- When the address of the value is passed to the function it is called call by reference.
- In call by reference since the address of the value is passed any changes made formal arguments (called function) that reflect in the actual arguments (calling function).

Example:

```
#include<stdio.h>
swap(int *, int *);
void main()
{
    int x,y;
    printf("Enter two numbers:");
    scanf("%d %d",&x,&y);
    printf("\nBefore swapping:x=%d y=%d",x,y);
    swap(&x,&y);
    printf("\nAfter swapping :x=%d y=%d",x,y);
}
swap(int *a, int *b)
{
    int t;
    t=*a;
    *a=*b;
    *b=t;
}
```

Output:

```
Enter two numbers:
12    34
Before swapping:
12    34
After swapping:
34    12
```

5. Write short note on return statement.

return statement:

- It is used to return the information from the function to the calling portion of the program.
- **Syntax:**
 return;
 return();
 return(constant);
 return(variable);
 return(exp);
 return(condn_exp);
- By default, all the functions return int data type.

6. Define recursive function. Write a C program to find factorial of a number using recursion?

Recursion:

- Recursion is calling function by itself again and again until some specified condition has been satisfied.
- **Syntax:**

```
int f_name (int x)
{
    local variables;
    f_name(y); // this is recursion
    statements;
}
```

Factorial using Recursion

```
#include<stdio.h>
int factorial(int );
void main()
{
    int res,x;
    printf("\n Enter the value:");
    scanf("%d", &x);
    res=factorial(x);
    printf("The factorial of %d is %d",x, res);
}
int factorial(int n)
{
    int fact;
    if (n==1)
        return(1);
    else
        return (n*factorial(n-1));
}
```

Output:

```
Enter the value: 5
The factorial of 5 is 120
```

7. Define recursive function. Write a C program to find Fibonacci series using recursion?

Recursion:

- Recursion is calling function by itself again and again until some specified condition has been satisfied.
- **Syntax:**

```
int f_name (int x)
{
    local variables;
    f_name(y); // this is recursion
    statements;
}
```

Fibonacci using Recursion

```
#include<stdio.h>
int Fibonacci(int);
void main()
{
    int n, i = 0, c;
    printf("Enter a number ");
    scanf("%d",&n);
    printf("Fibonacci series\n");
    for ( c = 1 ; c <= n ; c++ )
    {
        printf("%d\n", Fibonacci(i));
        i++;
    }
}
int Fibonacci(int n)
{
    if ( n == 0 )
        return 0;
    else if ( n == 1 )
        return 1;
    else
        return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

Output:

```
Enter a number 5
Fibonacci series
0
1
1
2
3
```

8. List and Explain storage classes in C?

- A variable's storage class tells us:
 - (a) Where the variable would be stored.
 - (b) What will be the initial value of the variable, if initial value is not specifically assigned.(i.e. the default initial value).
 - (c) What is the scope of the variable; i.e. in which functions the value of the variable would be available.
 - (d) What is the life of the variable; i.e. how long would the variable exist.
- There are four storage classes in C:
 1. Automatic storage class
 2. Register storage class
 3. Static storage class
 4. External storage class

1. Automatic storage class

- The features of a variable defined to have an automatic storage class are as under:
Storage: Memory.
Default initial value: A garbage value.
Scope: Local to the block in which the variable is defined.
Life time: Till the control remains within the block in which the variable is defined.

- **Example:**

```
#include<stdio.h>
main( )
{
    auto int i, j ;
    printf ( "%d %d", i, j ) ;
}
```

Output:

1211 221

Where, 1211 and 221 are garbage values of i and j.

2. Register storage class

- The features of a variable defined to be of register storage class are as under:
Storage - CPU registers.
Default initial value - Garbage value.
Scope - Local to the block in which the variable is defined.
Life time- Till the control remains within the block in which the variable is defined.

- **Example:**

```
#include<stdio.h>
main( )
{
    register int i ;
    printf ( "%d", i ) ;
}
```

Output:

1211

Where, 1211 are garbage value of i.

3. Static storage class

- The features of a variable defined to have a static storage class are as under:
Storage – Memory.
Default initial value – Zero.
Scope – Local to the block in which the variable is defined.
Life time– Value of the variable persists between different function calls.

- **Example:**

```
#include<stdio.h>
main( )
{
    static int i ;
    printf ( "%d", i ) ;
}
```

Output:

0

4. External storage class

- The features of a variable whose storage class has been defined as external are as follows:

Storage – Memory.

Default initial value – Zero.

Scope – Global.

Life time– As long as the program's execution doesn't come to an end.

- **Example:**

```
#include<stdio.h>
int a=3;
void func()
{
    extern int b;
    printf("%d\t%d\n",a,b);
}
int b=5;
void main()
{
    printf("%d \n",a);
    func();
}
```

Output:

```
3
3    5
```

9. Define a pointer? How to declare and initialize a pointer, Explain with an example?

Pointer:

- C Pointer is a variable that stores/points the address of another variable.
- C Pointer is used to allocate memory dynamically i.e. at run time.

Pointer Declaration:

- The variable might be any of the data type such as int, float, char, double, short etc.
- **Syntax :**

data_type *var_name;

- **Example :**

```
int *p;
char *p;
```

- Where, * is used to denote that "p" is pointer variable and not a normal variable.

Pointer –Initialization:

- Assigning value to pointer:

- It is not necessary to assign value to pointer.
- Only zero (0) and NULL can be assigned to a pointer no other number can be assigned to a pointer.
- Consider the following examples;

```
int *p=0;
int *p=NULL;
```

- The above two assignments are valid.

int *p=1000; This statement is invalid.

- **Assigning variable to a pointer:**

```
int x; *p;
```

```
p = &x;
```

- This is nothing but a pointer variable p is assigned the address of the variable x.
- The address of the variables will be different every time the program is executed.

- **Reading value through pointer:**

```
int x=123; *p;
```

```
p = &x;
```

- Here the pointer variable p is assigned the address of variable x.

Pointer Assignments:

- We can use a pointer on the right-hand side of an assignment to assign its value to another variable.
- **Example:**

```
int var=50;
```

```
int *p1, *p2;
```

```
p1=&var;
```

```
p2=p1;
```

Example program:

```
#include<stdio.h>
```

```
int main( )
```

```
{
```

```
    int x=123; *p;
```

```
    p = &x;
```

```
    printf("**p=%d", *p); //will display value of x 123. This is reading value
                          //through pointer
```

```
    printf("p=%d", p); //will display the address of the variable x.
```

```
    printf("&p=%d", &p); //will display the address of the pointer variable p.
```

```
    printf("x=%d",x); //will display the value of x 123.
```

```
    printf("&x=%d", &x); //will display the address of the variable x.
```

```
    return 0;
```

```
}
```

10. Explain about pointer arithmetic?

Pointer Expression & Pointer Arithmetic:

- C allows pointer to perform the following arithmetic operations:
- A pointer can be incremented / decremented.
- Any integer can be added to or subtracted from the pointer.
- A pointer can be incremented / decremented.
- In 16 bit machine, size of all types[data type] of pointer always 2 bytes.

- **Example:**

```
int a;
```

```
int *p;
```

```
p++;
```

- Each time that a pointer p is incremented, the pointer p will points to the memory location of the next element of its base type.

- Each time that a pointer p is decremented, the pointer p will point to the memory location of the previous element of its base type.

```
int a,*p1,*p2,*p3;
p1=&a;
p2=p1++;
p3=++p1;
printf("Address of p where it points to %u", p1);// prints 1000
printf("After incrementing Address of p where it points to %u", p1);//prints 1002
printf("After assigning and incrementing p %u", p2);// prints 1000
printf("After incrementing and assigning p %u", p3);// prints 1002
```

- In 32 bit machine, size of all types of pointer is always 4 bytes.
- The pointer variable p refers to the base address of the variable a..
- We can increment the pointer variable,

$$p++ \quad \text{or} \quad ++p$$
- This statement moves the pointer to the next memory address. let p be an integer pointer with a current value of 2,000 (that is, it contains the address 2,000).
- Assuming 32-bit integers, after the expression

$$p++;$$
the contents of p will be 2,004, not 2,001! Each time p is incremented, it will point to the next integer.
- The same is true of decrements. For example,

$$p--;$$
will cause p to have the value 1,996, assuming that it previously was 2,000.
- Here is why: Each time that a pointer is incremented, it will point to the memory location of the next element of its base type.
- Each time it is decremented, it will point to the location of the previous element of its base type.

Any integer can be added to or subtracted from the pointer:

- Like other variables pointer variables can be used in expressions. For example if p1 and p2 are properly declared and initialized pointers, then the following statements are valid.

```
y=*p1**p2;
sum=sum+*p1;
z= 5* - *p2/p1;
*p2= *p2 + 10;
```

- C allows us to add integers to or subtract integers from pointers as well as to subtract one pointer from the other.
- We can also use short hand operators with the pointers p1+=; sum+=*p2; etc., we can also compare pointers by using relational operators the expressions such as

p1 > p2 ,

p1==p2 and
p1!=p2 are allowed.

/*Program to illustrate the pointer expression and pointer arithmetic*/

```
#include<conio.h>
```

```
void main()
```

```
{
```

```
int ptr1,ptr2;
```

```
int a,b,x,y,z;
```

```
a=30;b=6;
```

```
ptr1=&a;
```



```

ptr2=&b;
x=*ptr1+ *ptr2 -6;
y=6*- *ptr1/ *ptr2 +30;
printf("\nAddress of a + %u",ptr1);
printf("\nAddress of b %u", ptr2);
printf("\na=%d, b=%d", a, b);
printf("\nx=%d,y=%d", x, y);
ptr1=ptr1 + Computer_Programming_Lecture_Notes70;
ptr2= ptr2;
printf("\na=%d, b=%d", a, b);
}

```

11. Explain about dynamic memory allocation functions?

Dynamic memory allocation:

- The process of allocating memory during program execution is called dynamic memory allocation.

Dynamic memory allocation functions:

S. No	Function	Syntax	Purpose
1	malloc()	ptr=(cast-type*)malloc(byte-size);	Allocates requested size of bytes and returns a pointer first byte of Allocated space.
2	calloc()	ptr=(cast-type*)calloc(n,element-size);	Allocates space for an array Elements, initializes to zero and then returns a pointer to memory.
3	free()	free(ptr);	De-allocate the previously allocated space.
4	realloc()	ptr=realloc(ptr,newsize);	Change the size of previously allocated space.

Example #1: Using malloc() and free()

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.

```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int num, i, *ptr, sum = 0;

    printf("Enter number of elements: ");
    scanf("%d", &num);

    ptr = (int*) malloc(num * sizeof(int)); //memory allocated using malloc
    if(ptr == NULL)
    {
        printf("Error! memory not allocated.");
    }
}

```

```

        exit(0);
    }

    printf("Enter elements of array: ");
    for(i = 0; i < num; ++i)
    {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }

    printf("Sum = %d", sum);
    free(ptr);
    return 0;
}

```

Example #2: Using calloc() and free()

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.

```

#include <stdio.h>
#include <stdlib.h>
int main()
{
    int num, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &num);
    ptr = (int*) calloc(num, sizeof(int));
    if(ptr == NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i = 0; i < num; ++i)
    {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }
    printf("Sum = %d", sum);
    free(ptr);
    return 0;
}

```

Example #3: Using realloc()

```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *ptr, i, n1, n2;
    printf("Enter size of array: ");
    scanf("%d", &n1);
    ptr = (int*) malloc(n1 * sizeof(int));
    printf("Address of previously allocated memory: ");
    for(i = 0; i < n1; ++i)

```

```
        printf("%u\t", ptr + i);  
    printf("\nEnter new size of array: ");  
    scanf("%d", &n2);  
    ptr = realloc(ptr, n2 * sizeof(int));  
    for(i = 0; i < n2; ++i)  
        printf("%u\t", ptr + i);  
    return 0;  
}
```

UNIT-4

1. Define structure? Explain declaration of structures and accessing of structure members?

Structure:

- Structure is a collection of elements of dissimilar data types.

(Or)

- A collection of one or more variables typically of different types, grouped together under a single name.

General Format of declaration:

```
struct tag
{
    datatype variable1;
    datatype variable2;
    .....
};
```

- Structure declaration always starts with “struct” keyword. Tag refers to the name of the structure.

Ex:-

```
struct book
{
    char b_name [30];
    char author[30];
    int pages;
    float price;
};
```

- The variables which declared and defined in the structure are known as members of the structures.

struct book a, b, c;

- The above statement declares and lets storage for three variables of each of the type struct book.

Struct book *p;

- It declares a pointer to an object of type struct book.

Accessing Members of A Structure:-

- We can access the members of a structure by using structure variables.
- Structure variables are normal variables and pointer variables.
- Dot (.) operator is used for normal variables and arrow (->) operator is used for pointers.

Initialization of structures:

Ex:-

```
struct book
{
    char name [30];
    int pages;
    float price;
};
```

```
struct book m= {"c language", 250, 300};
```

(Or)

```
m.name= "c language";
```

```
m.pages=250;
```

```
m.price=300;
```

Example program:

```
//Write a "c" program that use structures and display book details.
```

```
# include < stdio.h>
```

```
void main ()
```

```
{
```

```
    struct book
```

```
    {
```

```
        char b_name[20];
```

```
        char b_author [20];
```

```
        int pages;
```

```
        float price;
```

```
    };
```

```
    struct book b1, b2;
```

```
    printf ("enter details of first book \n");
```

```
    scanf ("%s %s %d %f", &b1.b_name, &b1.b_author,
        &b1.pages,&b1.price);
```

```
    printf ("enter the details of second book \n");
```

```
    scanf ("%s %s %d %f", &b2.b_name, &b2.b_author,
        &b2.pages,&b2.price);
```

```
    printf ("First book details \n");
```

```
    printf ("\t name = %s \n\t author =%s \n\t pages =%d\n\t price =%f\n",
        b1.b_name, b1.b_author, b1.pages, b1.price);
```

```
    printf ("Second book details \n");
```

```
    printf ("\t name = %s \n\t author =%s \n\t pages =%d\n\t price =%f\n",
        b2.b_name, b2.b_author, b2.pages, b2.price);
```

```
}
```

2. Write the differences between array and structure?

BASIS FOR COMPARISON	ARRAY	STRUCTURE
Basic	An array is a collection of variables of same data type.	A structure is a collection of variables of different data type.
Syntax	type array_name[size];	struct struct_name{ type element1; type element1; } variable1, variable2, . . .;
Memory	Array elements are stored in contiguous memory location.	Structure elements may not be stored in a contiguous memory location.
Access	Array elements are accessed by their index number.	Structure elements are accessed by their names.

Operator	Array declaration and element accessing operator is "[]" (square bracket).	Structure element accessing operator is "." (Dot operator).
Pointer	Array name points to the first element in that array so, array name is a pointer.	Structure name does not point to the first element in that structure so, structure name is not a pointer.
Objects	Objects (instances) of an array can not be created.	Structure objects (instance or structure variable) can be created.
Size	Every element in array is of same size.	Every element in a structure is of different data type.
Bit field	Bit field cannot be defined in an array.	Bit field can be defined in a structure.
Keyword	There is no keyword to declare an array.	"struct" is a keyword used to declare the structure.
User-defined	Arrays are not user-defined they are directly declared.	Structure is a user-defined data type.
Accessing	Accessing array element requires less time.	Accessing a structure elements require comparatively more time.
Searching	Searching an array element takes less time.	Searching a structure element takes comparatively more time than an array element.

3. Write the differences between structure and union?

	STRUCTURE	UNION
Keyword	The keyword struct is used to define a structure	The keyword union is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is greater than or equal to the sum of sizes of its members.	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of union is equal to the size of largest member.
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

4. Write a C program to demonstrate the use of array of structures?

```
#include<stdio.h>
#include<conio.h>
void main()
{
    struct book
    {
        char bname[];
        char aname[];
        int pages;
        float price;
```

```

    }b1[20];
    int i;
    printf("enter 10 book details");
    for(i=0;i<10;i++)
    scanf("%s %s %d %f",&b1[i].bname,&b1[i].aname,&b[i].pages,&b[i].price);
    printf("entered book details are");
    for(i=0;i<10;i++)
    {
        printf("bookname=%s\n author=%s\n pages=%d\n price=%f\n", b1[i].bname,
            b1[i].aname, b1[i].pages, b1[i].price);
    }
    getch();
}

```

5. Explain structure with in structure with an example?

Structure with in Structure (Nested Structures):

- When a structure contains another structure, it is called nested structure.
- For example, we have two structures named Address and Employee. To make Address nested to Employee, we have to define Address structure before and outside Employee structure and create an object of Address structure inside Employee structure.

- **Syntax:**

```

struct structure1
{
    -----
    -----
};

struct structure2
{
    -----
    -----
    struct structure1 obj;
};

```

- **Example for structure within structure or nested structure**

```

#include<stdio.h>
struct Address
{
    char HouseNo[25];
    char City[25];
    char PinCode[25];
};
struct Employee
{
    int Id;
    char Name[25];
    float Salary;
    struct Address Add;
};
void main()
{
    int i;
    struct Employee E;
    printf("\n\tEnter Employee Id : ");
}

```

```

scanf("%d",&E.Id);
printf("\n\tEnter Employee Name : ");
scanf("%s",&E.Name);
printf("\n\tEnter Employee Salary : ");
scanf("%f",&E.Salary);
printf("\n\tEnter Employee House No : ");
scanf("%s",&E.Add.HouseNo);
printf("\n\tEnter Employee City : ");
scanf("%s",&E.Add.City);
printf("\n\tEnter Employee House No : ");
scanf("%s",&E.Add.PinCode);
printf("\nDetails of Employees");
printf("\n\tEmployee Id : %d",E.Id);
printf("\n\tEmployee Name : %s",E.Name);
printf("\n\tEmployee Salary : %f",E.Salary);
printf("\n\tEmployee House No : %s",E.Add.HouseNo);
printf("\n\tEmployee City : %s",E.Add.City);
printf("\n\tEmployee House No : %s",E.Add.PinCode);
}

```

Output:

```

Enter Employee Id: 101
Enter Employee Name: Suresh
Enter Employee Salary: 45000
Enter Employee House No: 4598/D
Enter Employee City: Delhi
Enter Employee Pin Code: 110056

```

```

Details of Employees
Employee Id: 101
Employee Name: Suresh
Employee Salary: 45000
Employee House No: 4598/D
Employee City: Delhi
Employee Pin Code: 110056

```

6. Explain self referential structure with an example

- Self Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.
- In other words, structures pointing to the same type of structures are self-referential in nature.
- Example:

```

struct node
{
    int data1;
    char data2;
    struct node* link;
};
int main()
{
    struct node ob;
    return 0;
}

```


- In the above example 'link' is a pointer to a structure of type 'node'. Hence, the structure 'node' is a self-referential structure with 'link' as the referencing pointer.

7. What is union? Write a C program to store information in a union and display it.

Union:

- Both structure and union are collection of different data type.
- They are used to group number of variables of different type in a single unit.

Union Declaration:

- Declaration of union must start with the keyword union followed by the union name and union's member variables are declared within braces.
- **Syntax:**

```
union union-name
{
    datatype var1;
    datatype var2;
    -----
    -----
    datatype varN;
};
```

Accessing the union members:

- We have to create an object of union to access its members.
- Object is a variable of type union. Union members are accessed using the dot operator (.) between union's object and union's member name.

Syntax for creating object of union:

union union-name obj;

Example for creating object & accessing union members

```
#include<stdio.h>
union Employee
{
    int Id;
    char Name[25];
    int Age;
    long Salary;
};
void main()
{
    union Employee E;
    printf("\nEnter Employee Id : ");
    scanf("%d",&E.Id);
    printf("\nEnter Employee Name : ");
    scanf("%s",&E.Name);
    printf("\nEnter Employee Age : ");
    scanf("%d",&E.Age);
    printf("\nEnter Employee Salary : ");
    scanf("%ld",&E.Salary);
    printf("\n\nEmployee Id : %d",E.Id);
    printf("\nEmployee Name : %s",E.Name);
    printf("\nEmployee Age : %d",E.Age);
    printf("\nEmployee Salary : %ld",E.Salary);
}
```

Output:

Enter Employee Id: 1
Enter Employee Name: Kumar
Enter Employee Age: 29
Enter Employee Salary: 45000

Employee Id: 1
Employee Name: Kumar
Employee Age: 29
Employee Salary: 45000

8. Differentiate between structure and unions.

	STRUCTURE	UNION
Keyword	The keyword struct is used to define a structure	The keyword union is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is greater than or equal to the sum of sizes of its members.	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of union is equal to the size of largest member.
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

9. How to pass the structures to functions as an argument? Explain with a suitable example.

- Like ordinary variables a structure variable can also be passed to a function.
- We may either pass individual structure members or pass whole structure variable at one time.

Example:

// passing whole structure variable at once to a function

```
#include<stdio.h>
```

```
#include<conio.h>
```

```
struct book
```

```
{
```

```
    char bname[20];
```

```
    char aname[20];
```

```
    int pages;
```

```
    float price;
```

```
};
```

```
void show(struct book);
```

```
void main()
```

```
{
```

```
    struct book k={"java","steve","1000","450.00"};
```

```
    show(k);
```

```
    getch();
```

```
}
```

```
void show(struct book)
{
    printf("%s %s %d %f", k.bname, k.aname, k.pages, k.price);
}
```

Output:

Java Steve 1000 450.000000

10. Write short note on pointer to structure.

Pointer to structure:

- Pointer is used to store the address of another variable. That variable may be of any type. As we declare pointer for int, float, char, in the same way we can declare for structures also. In this case, the starting address of the member variable can be accessed. This type of pointers is called as pointer to structures.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    struct book
    {
        char bname[];
        char aname[];
        int pages;
        float price;
    };
    struct book b={"cds","kamthane",500,399.00};
    struct book *p;
    p=&b;
    clrscr();
    printf("%s %s %d %f", b.bname, b.aname, b.pages, b.price);
    printf("%s %s %d %f", p->bname, p->aname, p->pages, p->price);
    getch();
}
```

Output:-

CDS kamthane 500 399.000000

CDS kamthane 500 399.000000

11. Write short note on typedef.

typedef:

- The purpose of typedef is to assign alternative names to existing types.
- Type def is a keyword in "C"
- A typedef declaration does not reserve storage.
- The names you define using typedef are not new data types, but synonyms for the data types or combination of data types they represent.

Example:

```
typedef int LENGTH;
LENGTH length,weight,height;
```

This is same as

```
int length,weight,height;
```

Usage of typedef in structures:

```
struct new
{
    int var1;
    int var2;
    float var3;
};
```

- To create a variable of struct new type we need struct key word as struct new p;
- A typedef can be used to eliminate the need for struct keyword in "c"

Example:

```
typedef struct new TYPE;
```

We can now create a variable of this type with
TYPE p;

Advantages of type def:

- It provides a mean to make a program more portable. Instead of having to change a type everywhere it appears throughout the programs source files, only a single typedef statement needs to be changed.
- A typedef can make a complex declaration easier to understand

12. Write short note on Enumerated data types.

Enumerated data types (enum):-

- An enumerated data type is a data type consisting of a set of named values called elements, members or enumerators of the type.
- enum is the keyword used for declaration of enumerated data types.
- The enumerator names are usually identifiers that behave as constants in the language

Example:

- red, green, black, blue belongs to colour enum
- club, diamond, heart, spade belongs to play card enum
- married, single, divorced, widowed belongs to marital status enum
- The following declaration declares the data type and specifies its possible values. These values are called "enumerators".

```
enum cardsuit { CLUBS, DIAMONDS, HEARTS, SPADES, };
```
- The following statement declares variables of type enum card suit

```
enum cardsuit play1, play2;
```
- Now we can give values to these variables

```
play1=CLUBS;
play2=SPADES;
```
- Internally the compiler treats the enumerator as integers. Each value on the list of permissible values corresponds to an integer, starting with 0.
- Thus in the above example CLUBS=0, DIAMONDS=1, HEARTS=2, SPADES=3
- This way of assigning numbers can be overridden by the programmer by initializing the enumerators to different integer values as shown below

```
enum cardsuit
{
    CLUBS=10, DIAMONDS=12, HEARTS=13, SPADES=15
};
```

13. Write short note on bit fields.

Bit fields:-

- A bitfield is a common idiom used in computer programming to compactly store a value as a short series of bits.
- A bitfield is most commonly used to represent integral types of known fixed bit-width

Example:

- If we want to store an employee's information, each employee can have the following data
 - Be a male or female
 - Be single, married , divorced or widowed
 - Have one of the eight different hobbies
- According to the above data we need one bit to store gender, two bit to store marital status, three for hobby. We need six bits to pack all the data.
- To do this using bit fields, we declare the following structure.

```
struct emp
{
    unsigned gender : 1 ;
    unsigned mar_stat : 2;
    unsigned hobby : 3;
};
```

- The colon (:) in the above declaration tells the compiler that we are talking about bit fields and the number after it tells how many bits are to be allotted for that field.

```
#include<stdio.h>
#define male 0
#define female 1
#define single 0
#define married 1
#define divorced 2
#define widowed 3
main()
{
    struct emp
    {
        unsigned gender:1 ;
        unsigned mar_stat:2;
        unsigned hobby:3;
    };
    struct emp d;
    d.gender = male;
    d.mar_stat = divorced;
    d.hobby = 5;
    printf("gender=%d\n", d.gender);
    printf("marital status=%d\n", d.mar_stat);
    printf("bytes occupied by d=%d\n", sizeof(d));
    getch();
}
```

Output:

Gender=0

Marital status=2

Bytes occupied by d=2

UNIT-5

1. Explain FILE structure in detail.

Files:

Definition:

- A file is a collection of bytes stored on a secondary storage device, which is generally a disk of some kind.
- A file is a collection of related data stored on a hard disk or secondary storage device. Files can be accessed using library functions.
- The files accessed through the library functions are called “stream oriented files” and the files accessed with system calls are known as “system oriented files”.

Stream:

- Stream means reading and writing of data.
- Stream is classified into 3 types and those are defined in io.h header file.
 1. **Stdin** : reads input from the keyboard
 2. **stdout** : send output to the screen.
 3. **stderr** : print errors to an error device.

File types:

- Based on **internal storage representation**, files are two types:-
 1. Text files
 2. Binary files

1. Text File: (COLLECTION OF CHARACTER STORES IN A FILE)

- The I/O stream can be a text stream or binary stream.
- A text stream is a sequence of text. In a text stream, each line consists of zero or more characters, terminated with newline character.

2. Binary File: (collection of ASCII values stores in a file)

- A binary stream is a sequential of bytes. In a binary stream, it represents raw data without any conventions.

- Based on **accessing files** are two types:-
 1. Sequential file
 2. Random access file.

1. Sequential File:

- In this type of files records are kept sequential. If we want to read the last record of the file we need to read all the records before that record.

2. Random Access File:

- In this type of files data can be read and modified randomly. In this type if we want to read the last records of the file, we can read it directly.

FILE:

- In C, FILE is a structure that holds the description of a file and is defined in stdio.h.
- It contains the current status of the file. FILE is a data type and a region of storage.

File pointer:

- Declaration of file pointer is as follows,

FILE *variable-name;

Where variable-name refers to the name of the pointer variable.

Example:

```
FILE *fp1,*fp2;  
FILE *fptr;
```

- A file pointer is a pointer to FILE data type.
- File pointer is also called as a stream pointer or buffer pointer.
- A file pointer points to the block of information of the stream that has just been opened.

File Operations:-

- a) Creation of a new file.
- b) Opening an existing file.
- c) Reading from a file.
- d) Writing to a file.
- e) Moving to a specific location. (Seeking)
- f) Closing a file.

2. Write short note on file opening modes.**File opening modes:**

MODE	DESCRIPTION
" r "	<ul style="list-style-type: none">• Reading from the file.• Searches the file if the file is opened successfully fopen() loads it into memory and sets up a pointer which points to the first character in it.• If the file cannot be opened fopen() returns NULL.
" w "	<ul style="list-style-type: none">• Writing to the file.• Searches the file, if the file exist its contents are overwritten.• If the file doesn't exist a new file is created.• If the file cannot be opened fopen() returns NULL.
" a "	<ul style="list-style-type: none">• Appends new content at the end of file.• Searches the file if the file is opened successfully fopen() loads it into the memory and sets up a pointer that points to the last character in it.• If the doesn't exist a new file is created.• If the file cannot be opened fopen() returns NULL.
" r+ "	<ul style="list-style-type: none">• Reading existing contents, creating new contents, modifying existing contents of the file.• Searches file, if it is opened successfully.• Fopen() loads it into memory and sets up a pointer which points to the first byte character in it.• If the file cannot be opened fopen() returns NULL.
" w+ "	<ul style="list-style-type: none">• Reading existing contents, writing new contents. modifying existing contents of the file.• Searches file, if it is opened successfully its contents are overwritten.• If the file doesn't exist a new file is created.• If the file cannot be opened fopen() returns NULL.
" a+ "	<ul style="list-style-type: none">• Reading existing contents, appending new contents to the end of the file.• Cannot modify existing contents of the file.• Searches file, if it is opened successfully fopen() loads it into memory and sets up a pointer which points to the first byte character in it.• If the file doesn't exist a new file is created.• If the file cannot be opened fopen() returns NULL.

3. Describe the following file functions.

(a) fopen().

(b) fclose().

File Open/Close:**fopen():**

- fopen() performs three important tasks when you open the file in "r" mode.
 1. Firstly it searches on the hard disk the file to be opened.
 2. Then it loads the file from the disk into a place in memory called buffer.
 3. It sets up a pointer that points to the first character of the buffer.

Syntax:

fopen("file name","mode");

fclose():

- When we no longer need an opened file, we should close it to free system resources, such as buffer space.
- A file is closed using the close function fclose().

Syntax:

fclose(file_ptr);

Example:

//Write a program to read the file and display its contents on the screen.

```
#include <stdio.h>
```

```
int main( )
```

```
{
```

```
    FILE *fp;
```

```
    char ch;
```

```
    fp=fopen("factorial.c","r");
```

```
    while(1)
```

```
    {
```

```
        ch=fgetc(fp);
```

```
        if(ch==EOF)
```

```
            break;
```

```
        else
```

```
            printf("%c",ch);
```

```
    }
```

```
    fclose(fp);
```

```
}
```

4. Describe formatted input output statements.

Formatted I/O:

1. fprintf():

syntax:

```
printf("control string", variable-list);
```

```
fprintf(file-pointer, "control string", variable list);
```

2. fscanf():

syntax:

```
scanf("control string",variable-list);
```

```
fscanf(file-pointer, "control string", variable list);
```

//program that shows the example of fprintf() and fscanf().

```
#include<stdio.h>
```

```
#include<io.h>
```

```
#inlucde<conio.h>
```

```
int main( )
```

```
{
```

```
    FILE *fp;
```

```
    char s[80];
```

```
    int t;
```

```
    clrscr( );
```

```
    if(( fp=fopen("test.txt","w"))==NULL)
```

```
    {
```

```
        printf("file cannot open\n");
```

```
        exit(1);
```



```

    }
    printf("enter a string and a numbers\n");
    fscanf(stdin,"%s%d",s,&t);
    fprintf(fp,"%s%d",s,t);
    fclose(fp);
    if((fp==fopen("test.txt","r"))=NULL)
    {
        printf("cannot open file");
        exit(1);
    }
    fscanf(fp,"%s%d",s,&t);
    fprintf(stdout,"%s%d",s,t);
    return 0;
    getch();
}

```

5. Describe the following file functions.

(a) **fread()**.

(b) **fwrite()**

fread() :

Syntax:

*fread (void *pinarea, int elements size, int count, file-ptr);*

- Here "**pinarea**" refers to the input area in memory. Here generic pointer is used.
- This allows any pointer type to be passed to the function.
- File read expects a pointer to the input area, which is usually a structure. This is because binary files are most often used to store structure.
- The "**elements size**" and "count" are multiplied to determine how much data are to be transferred. The size is normally specified using the sizeof() operator.
- **Count** is no of records in the structure.
- The **last parameter** is the file pointer.

fwrite() :

Syntax:

*fwrite (void *poutarea, int elementsize, int count, file-ptr);*

- fwrite() copies "elementsiz*count" bytes from the address specified by "poutarea" to the file.
- Here "**poutarea**" refers to the output area in memory. Here generic pointer is used.
- This allows any pointer type to be passed to the function.
- The **size** is normally specified using the sizeof() operator.
- **Count** is no of records in the structure.
- The last parameter is the file pointer.

//Write a program that shows the example of fread and fwrite functions and read and write an array of structures.

```
#include<stdio.h>
```

```
#include<conio.h>
```

```
int main (void)
```

```
{
```

```
    FILE *fp;
```

```
    struct product
```

```
    {
```

```
        int cat_num;
```

```
        float cost;
```

```

};
struct product p[3]={ { 3,30.2},{4,50.76},{5,70.9}};
struct product *k=&p;
fp=fopen("sample.txt","w");
fwrite(p, sizeof(product),3,fp);
rewind(fp);
for(i=0;i<3,i++)
    fread (k,sizeof(product),1,fp);
printf("product %d;cat_num=%d,cost=%f\n",i,k->cat_num,k->cost);
}
getch( );
return 0;
}

```

6. Describe the following file functions.

(a) **fputc().**

(b) **fgetc().**

fputc():

- fputc() is used to write a single character at a time to a given file.
- It writes the given character at the position denoted by the file pointer and then advances the file pointer.
- This function returns the character that is written in case of successful write operation else in case of error EOF is returned.

Syntax:

```
fputc (int ch, file pointer);
```

fgetc():

- fgetc() is used to obtain input from a file single character at a time.
- This function returns the number of characters read by the function.
- It returns the character present at position indicated by file pointer.
- After reading the character, the file pointer is advanced to next character.
- If pointer is at end of file or if an error occurs EOF file is returned by this function.

Syntax:

```
fgetc (file pointer);
```

//Write a program to write some characters on file and read them back.

```

#include<stdio.h>
main( )
{
FILE *fptr;
char ch;
clrscr( );
fptr=fopen("new.txt","w");
printf("enter some characters on file");
while((ch=getchar( ))!=EOF)
fputc(ch,fptr);
fclose(fptr);
fptr=fopen("new.txt","r");
while((ch=getchar( ))!=EOF)
ch=fgetc(fptr);
fclose(fptr);
getch( );
}

```

7. Write short note on file status functions.

1. Test end of file (feof()):

Syntax:

feof(file-ptr);

- feof function is used to check if the end of the file has been reached.
- If used is at the end that is, if all the data have read the function returns nonzero (true).
- If the end of the file has not been reached zero (false) is returned.

2. Test error (ferror()):

Syntax:

ferror(file-ptr);

- This function is used to check the error status file.
- Errors can be created for memory reasons.
- Ranging bad physical media to illogical operations, such as to read a file in the write state.
- The ferror() returns true(nonzero) if an error has occurred. It returns false (zero) if no error has occurred.

//Write a program that illustrates error handling in file operations.

```
#include<stdio.h>
```

```
main( )
```

```
{
```

```
    char *filename;
```

```
    FILE *fp1, *fp2;
```

```
    int l,n;
```

```
    fp1=fopen("sample.txt","w");
```

```
    for( i=0; i<=100; i=i+100)
```

```
    putw(l,fp1);
```

```
    fclose(fp1);
```

```
    printf("enter file name");
```

```
    open-file;
```

```
    scanf("%s", filename);
```

```
    if((fp2=fopen (filename,"r"))= NULL)
```

```
    {
```

```
        printf("cannot open the file\n");
```

```
        printf("type file name again\n");
```

```
        goto open file;
```

```
    }
```

```
    else
```

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

```
        {
```

```
            n=getw(fp2);
```

```
            if(feof(fp2))
```

```
            {
```

```
                printf("end of file\n");
```

```
                break;
```

```
            }
```

```
            else
```

```
                rintf("%d",n);
```

```
        }
```

```
    fclose(fp2);
```

```
}
```

8. Write about file positioning functions.

Positioning Functions:

1. Rewind file(rewind):

Syntax

rewind (file-ptr);

This function simply sets the file pointer to the beginning of the file.

2. Current position (ftell):

Syntax:

ftell(file-ptr);

This function returns the current position of the file pointer in the file relative to the beginning of the file.

Example:

```
#include<stdio.h>
main( )
{
    FILE *fptr;
    long size;
    clrscr( );
    fptr = fopen("ABC.txt","r");
    if (fptr == NULL)
        perror ("enter opening file");
    else
    {
        fseek(fptr,0,SEEK_END);
        size=ftell(fptr);
        fclose(fptr);
        fclose(fptr);
        printf("size of ABC.txt=%ld bytes", size);
    }
}
```

9. Write about random accessing a file.

Random Accessing a File:

fseek():

- The fseek() positions the file pointer to a specific position in a file.

Syntax:

fseek (file ptr, long offset, int whereform);

- The first parameter is a position to opened file.
- The second parameter is assigned integer that specifies the number of bytes the pointer must move absolutely.
- The third parameter "whereform" is three types.

SEEK_SET	0	Beginning of file
SEEK_CUR	1	Current position of file pointer
SEEK_END	2	End of file

- This function returns zero value on success and nonzero value on failure.

Example:-

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

```

FILE *fp;
fp=fopen("example.txt","w");
fputs("this is an apple",fp);
fseek(fp,9,SEEK_SET)
fputs("sam",fp)
fclose(fp);
return 0;
getch( );
}

```

Operations of fseek() Function:-

Statement	Meaning
fseek(fp,0,0)	Go to the beginning.
fseek(fp,0,1)	Stay at the current position.
fseek(fp,0,2)	Go to the end of file past the last character of the file.
fseek(fp,m,0)	More to(m+1) the bytes in the file.
fseek(fp,m,1)	Go forward by m bytes.
fseek(fp,-m,1)	Go backward by m bytes from the current position.
fseek (fp,-m, 2)	Go backward by m bytes from the end.(postion the file to the file m th character from the end).

10. Write about command line arguments in C?

Command Line Arguments:

- All the programs we have been coded with no parameters for main but main is a function and as a function, it may have parameters when main has parameters they are known as command line arguments.
- These arguments allow the user to specify additional information when the program is invoked.
- Command line arguments are two types. One an integer and the other an array of pointers to char (strings) that represents user-determined values to be passed to main.

int main(int argc, char *argv[])

argc - argument count

argv - argument vector

- The first arguments the number of elements in the array identify in the second arguments.
- The value for the arguments is not entered using the keyboard; the system determines it from the arguments the user types.
- The value of argc is determined from the user-typed values for argv.

// Program that demonstrate the use of command line arguments.

```

int main (int argc, char *argv)
{
int i;
printf("the no of arguments: %d\n", argc);
printf("the name of the program: %s\n", argv [0]);
for (i=1; i<argc; i++)
printf ("user values no: %d, %s\n", i, argv[i]);
return 0;
}

```

11. Explain about Macros with an example.

Preprocessor directives (Macros)

The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. We'll refer to the C Preprocessor as CPP.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives –

S.No.	Directive	Description
1	#define	Substitutes a preprocessor macro.
2	#include	Inserts a particular header from another file.
3	#undef	Undefines a preprocessor macro.
4	#ifdef	Returns true if this macro is defined.
5	#ifndef	Returns true if this macro is not defined.
6	#if	Tests if a compile time condition is true.
7	#else	The alternative for #if.
8	#elif	#else and #if in one statement.
9	#endif	Ends preprocessor conditional.
10	#error	Prints error message on stderr.
11	#pragma	Issues special commands to the compiler, using a standardized method.

Preprocessors Examples

Analyze the following examples to understand various directives.

```
#define MAX_ARRAY_LENGTH 20
```

This directive tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20. Use *#define* for constants to increase readability.

```
#include <stdio.h>
```

```
#include "myheader.h"
```

These directives tell the CPP to get stdio.h from **System Libraries** and add the text to the current source file. The next line tells CPP to get **myheader.h** from the local directory and add the content to the current source file.

```
#undef FILE_SIZE
```

```
#define FILE_SIZE 42
```

It tells the CPP to undefine existing FILE_SIZE and define it as 42.

```
#ifndef MESSAGE
```

```
#define MESSAGE "You wish!"
```

```
#endif
```

It tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

```
#ifdef DEBUG
```

```
/* Your debugging statements here */
```

```
#endif
```

It tells the CPP to process the statements enclosed if DEBUG is defined. This is useful if you pass the *-DDEBUG* flag to the gcc compiler at the time of compilation. This will define DEBUG, so you can turn debugging on and off on the fly during compilation.

Predefined Macros

ANSI C defines a number of macros. Although each one is available for use in programming, the predefined macros should not be directly modified.

S.No.	Macro	Description
1	<code>__DATE__</code>	The current date as a character literal in "MMM DD YYYY" format.
2	<code>__TIME__</code>	The current time as a character literal in "HH:MM:SS" format.
3	<code>__FILE__</code>	This contains the current filename as a string literal.
4	<code>__LINE__</code>	This contains the current line number as a decimal constant.
5	<code>__STDC__</code>	Defined as 1 when the compiler complies with the ANSI standard.

Let's try the following example –

```
#include <stdio.h>
int main()
{
    printf("File :%s\n", __FILE__ );
    printf("Date :%s\n", __DATE__ );
    printf("Time :%s\n", __TIME__ );
    printf("Line :%d\n", __LINE__ );
    printf("ANSI :%d\n", __STDC__ );
}
```

Output:

```
File :test.c
Date :Jun 2 2012
Time :03:36:24
Line :8
ANSI :1
```

