CHAPTER 1: Variables & Control Flow

1. **What is the difference between declaring a variable and defining a variable?**

Declaration of a variable in C hints the compiler about the type and size of the variable in compile time. Similarly, declaration of a function hints about type and size of function parameters. No space is reserved in memory for any variable in case of declaration.

Example: int a;
Here variable 'a' is declared of data type 'int'

Defining a variable means declaring it and also allocating space to hold it.
We can say "Definition = Declaration + Space reservation".

Example: int a = 10;
Here variable "a" is described as an int to the compiler and memory is allocated to hold value 10.

2. **What is a static variable?**

A static variable is a special variable that is stored in the data segment unlike the default automatic variable that is stored in stack. A static variable can be initialized by using keyword static before variable name.

Example:
static int a = 5;

A static variable behaves in a different manner depending upon whether it is a global variable or a local variable. A static global variable is same as an ordinary global variable except that it cannot be accessed by other files in the same program / project even with the use of keyword extern. A static local variable is different from local variable. It is initialized only once no matter how many times that function in which it resides is called. It may be used as a count variable.

Example:

```
#include <stdio.h>
//program in file f1.c
void count(void) {
 static int count1 = 0;
 int count2 = 0;
 count1++;
 count2++;
 printf("\nValue of count1 is %d, Value of count2 is %d", count1, count2);
}
/*Main function*/
int main(){
 count();
 count();
 count();
 return 0;
}
```

Output:

```
Value of count1 is 1, Value of count2 is 1
Value of count1 is 2, Value of count2 is 1
```

Value of count1 is 3, Value of count2 is 1

3. What is a register variable?

Register variables are stored in the CPU registers. Its default value is a garbage value. Scope of a register variable is local to the block in which it is defined. Lifetime is till control remains within the block in which the register variable is defined. Variable stored in a CPU register can always be accessed faster than the one that is stored in memory. Therefore, if a variable is used at many places in a program, it is better to declare its storage class as register

Example:
register int x=5;
Variables for loop counters can be declared as register. Note that register keyword may be ignored by some compilers.

4. Where is an auto variables stored?

Main memory and CPU registers are the two memory locations where auto variables are stored. Auto variables are defined under automatic storage class. They are stored in main memory. Memory is allocated to an automatic variable when the block which contains it is called and it is de-allocated at the completion of its block execution.

Auto variables:

Storage	**:**	main memory.
Default value	**:**	garbage value.
Scope	**:**	local to the block in which the variable is defined.
Lifetime	**:**	till the control remains within the block in which the variable is defined.

5. What is scope & storage allocation of extern and global variables?

Extern variables: belong to the External storage class and are stored in the main memory. extern is used when we have to refer a function or variable that is implemented in other file in the same project. The scope of the extern variables is Global.

Example:

```
/***************
Index: f1.c
****************/
#include <stdio.h>
extern int x;
int main() {
 printf("value of x %d", x);
 return 0;
}

Index: f2.c
****************/
int x = 3;
```

Here, the program written in file f1.c has the main function and reference to variable x. The file f2.c has the declaration of variable x. The compiler should know the datatype of x and this is done by *extern* definition.

Global variables: are variables which are declared above the main( ) function. These variables are accessible throughout the program. They can be accessed by all the functions in the program. Their default value is zero.

Example:

```
#include <stdio.h>
int x = 0;

/* Variable x is a global variable.
It can be accessed throughout the program */
void increment(void) {
 x = x + 1;
 printf("\n value of x: %d", x);
}
int main(){
 printf("\n value of x: %d", x);
 increment();
 return 0;
}
```

6. **What is scope & storage allocation of register, static and local variables?**

Register variables: belong to the register storage class and are stored in the CPU registers. The scope of the register variables is local to the block in which the variables are defined. The variables which are used for more number of times in a program are declared as register variables for faster access.
Example: loop counter variables.

```
register int y=6;
```

Static variables: Memory is allocated at the beginning of the program execution and it is reallocated only after the program terminates. The scope of the static variables is local to the block in which the variables are defined.

Example:

```
#include <stdio.h>
void decrement(){
  static int a=5;
  a--;
  printf("Value of a:%d\n", a);
}
int main(){
 decrement();
 return 0;
}
```

Here 'a' is initialized only once. Every time this function is called, 'a' does not get initialized. so output would be 4 3 2 etc.,

Local variables: are variables which are declared within any function or a block. They can be accessed only by function or block in which they are declared. Their default value is a garbage value.

7. **What are storage memory, default value, scope and life of Automatic and Register storage class?**

1. **Automatic storage class:**

Storage	**:**	main memory.
Default value	**:**	garbage value.

Scope : local to the block in which the variable is defined.
Lifetime : till control remains within the block.

2. Register storage class:

Storage : CPU registers.
Default value : garbage value.
Scope : local to the block in which the variable is defined.
Lifetime : till control remains within the block.

8. What are storage memory, default value, scope and life of Static and External storage class?

1. Static storage class:

Storage : main memory.
Default value : zero
Scope : local to the block in which the variable is defined.
Lifetime : till the value of the variable persists between different function calls.

2. External storage class:

Storage : main memory
Default value : zero
Scope : global
Lifetime : as long as the program execution doesn't come to an end.

9. What is the difference between 'break' and 'continue' statements?

Differences between 'break' and 'continue' statements

break	continue
1. break is a keyword used to terminate the loop or exit from the block. The control jumps to next statement after the loop or block.	1. continue is a keyword used for skipping the current iteration and go to next iteration of the loop

2.Syntax:
 	
{
Statement 1;
Statement 2;
Statement n;
break;
}
	2.Syntax:
	
	{
	Statement 1;
	continue;
	Statement 2;
	}

3. break can be used with for, while, do- while, and switch statements. When break is used in nested loops i.e. within the inner most loop then only the innermost loop is terminated.	3. This statement when occurs in a loop does not terminate it but skips the statements after this continue statement. The control goes to the next iteration. continue can be used with for, while and do-while.

4. Example:
 	
i = 1, j = 0;
while(i<=5)
{
i=i+1;
if(i== 2)
	4. Example:
	
	i = 1, j = 0;
	while(i<=5)
	{
	i=i+1;
	if(i== 2)

break;
j=j+1;
}
	continue;
	j=j+1;
	}

10. What is the difference between 'for' and 'while' loops?

for loop: When it is desired to do initialization, condition check and increment/decrement in a single statement of an iterative loop, it is recommended to use 'for' loop.

Syntax:

```
for(initialization;condition;increment/decrement)
{
//block of statements
increment or decrement
}
```

Program: Program to illustrate for loop

```
#include<stdio.h>
int main() {
 int i;
 for (i = 1; i <= 5; i++) {
  //print the number
  printf("\n %d", i);
 }
 return 0;
}
```

Output:

```
1
2
3
4
5
```

Explanation:
The loop repeats for 5 times and prints value of 'i' each time. 'i' increases by 1 for every cycle of loop.

while loop: When it is not necessary to do initialization, condition check and increment/decrement in a single statement of an iterative loop, while loop could be used. In while loop statement, only condition statement is present.

Syntax:

```
#include<stdio.h>
int main() {
 int i = 0, flag = 0;
 int a[10] = { 0, 1, 4, 6, 89, 54, 78, 25, 635, 500 };
 //This loop is repeated until the condition is false.
```

```
 while (flag == 0) {
  if (a[i] == 54) {
   //as element is found, flag = 1,the loop terminates
   flag = 1;
  }
  else {
   i++;
  }
 }
 printf("Element found at %d th location", i);
 return 0;
}
```

Output:
Element found at 5th location

Explanation:
Here flag is initialized to zero. 'while' loop repeats until the value of flag is zero, increments i by 1. 'if' condition checks whether number 54 is found. If found, value of flag is set to 1 and 'while' loop terminates.

CHAPTER 2: Operators, Constants & Structures

1. **Which bitwise operator is suitable for checking whether a particular bit is ON or OFF?**

Bitwise AND operator.

Example: Suppose in byte that has a value 10101101 . We wish to check whether bit number 3 is ON (1) or OFF (0) . Since we want to check the bit number 3, the second operand for AND operation we choose is binary 00001000, which is equal to 8 in decimal.

Explanation:
ANDing operation :

```
 10101101      original bit pattern
 00001000      AND mask
---------
 00001000      resulting bit pattern
---------
```

The resulting value we get in this case is 8, i.e. the value of the second operand. The result turned out to be a 8 since the third bit of operand was ON. Had it been OFF, the bit number 3 in the resulting bit pattern would have evaluated to 0 and complete bit pattern would have been 00000000. Thus depending upon the bit number to be checked in the first operand we decide the second operand, and on ANDing these two operands the result decides whether the bit was ON or OFF.

2. **Which bitwise operator is suitable for turning OFF a particular bit in a number?**

Bitwise AND operator (&), one's complement operator(~)

Example: To unset the 4th bit of byte_data or to turn off a particular bit in a number.

Explanation:
Consider,

char byte_data= 0b00010111;
byte_data= (byte_data)&(~(1<<4));
1 can be represented in binary as 0b00000001 = (1<<4)
<< is a left bit shift operator,
it shifts the bit 1 by 4 places towards left.
(1<<4) becomes 0b00010000
And ~ is the one's complement operator in C language.
So ~(1<<4) = complement of 0b00010000
= 0b11101111
Replacing value of byte_data and ~(1<<4) in
(byte_data)&(~(1<<4));
we get (0b00010111) & (0b11101111)
Perform AND operation to below bytes.

```
 00010111
 11101111
-----------
 00000111
-----------
```

Thus the 4th bit is unset.

3. **What is equivalent of multiplying an unsigned int by 2: left shift of number by 1 or right shift of number by 1?**

Left shifting of an unsigned integer is equivalent to multiplying an unsigned int by 2.

Eg1: 14<<1;
Consider a number 14-----00001110 (8+4+2)is its binary equivalent
left shift it by 1----------00011100(16+8+4) which is 28.

Eg2: 1<<1;
consider the number as 1---00000001(0+0+1).
left shift that by 1----------00000010(0+2+0) which is 2.
left shift by 1 bit of a number=2*number
left shift by 1 bit of 2*number=2*2*number
left shift by n bits of number=(2^n)*number

Program: Program to illustrate left shift and right shift operations.

```
#include<stdio.h>
int main(void)
{
 int x=10,y=10;
 printf("left shift of 10 is %d \n",x<<1);
 printf("right shift of 10 is %d \n",y>>1);
 return 0;
}
```

Output:
left shift of 10 is 20
right shift of 10 is 5

Explanation:
Left shift (by 1 position) multiplies a number by two. Right shift divides a number by 2.

4. **What is an Enumeration Constant?**

Enumeration is a data type. We can create our own data type and define values that the variable can take. This can help in making program more readable. enum definition is similar to that of a structure.

Example: consider light_status as a data type. It can have two possible values - on or off.

```
enum light_status
{
  on, off
};
enum light_status bulb1, bulb2;
/* bulb1, bulb2 are the variables */
```

Declaration of enum has two parts:

a) First part declares the data type and specifies the possible values, called 'enumerators'.
b) Second part declares the variables of this data type.
We can give values to these variables:
bulb1 = on;
bulb2 = off;

5. **What is a structure?**

A structure is a collection of pre-defined data types to create a user-defined data type. Let us say we need to create records of students. Each student has three fields:

```
int roll_number;
char name[30];
int total_marks;
```

This concept would be particularly useful in grouping data types. You could declare a structure student as:

```
struct student {
 int roll_number;
 char name[30];
 int total_marks;
} student1, student2;
```

The above snippet of code would declare a structure by name student and it initializes two objects student1, student2. Now these objects and their fields could be accessed by saying student1.roll_number for accesing roll number field of student1 object, similarly student2.name for accesing name field of student2 object.

6. **What are the differences between a structure and a union?**

Structures and Unions are used to store members of different data types.

STRUCTURE	**UNION**

a)**Declaration:**
 	
struct
{
data type member1;
data type member2;
};
	a)**Declaration:**
	
	union
	{
	data type member1;
	data type member2;
	};

b)Every structure member is allocated memory when a structure variable is defined. **Example:**	b)The memory equivalent to the largest item is allocated commonly for all members. **Example:**

```
struct emp {
 char name[5];
 int age;
 float sal;
};
struct emp e1;
```

Memory allocated for structure is 1+2+4=7 bytes. 1 byte for name, 2 bytes for age and 4 bytes for sal.

c)All structure variables can be initialized at a time

```
struct st {
 int a;
 float b;
};
struct st s = { .a=4, .b=10.5 };
```

Structure is used when all members are to be independently used in a program.

```
union emp1 {
 char name[5];
 int age;
 float sal;
};
union emp1 e2;
```

Memory allocated to a union is equal to size of the largest member. In this case, float is the largest-sized data type. Hence memory allocated to this union is 4 bytes.

c)Only one union member can be initialized at a time

```
union un {
 int a;
 float b;
};
union un un1 = { .a=10 };
```

Union is used when members of it are not required to be accessed at the same time.

7. **What are the advantages of unions?**

Union is a collection of data items of different data types. It can hold data of only one member at a time though it has members of different data types. If a union has two members of different data types, they are allocated the same memory. The memory allocated is equal to maximum size of the members. The data is interpreted in bytes depending on which member is being accessed.

Example:

```
union pen {
 char name;
 float point;
};
```

Here name and point are union members. Out of these two variables, 'point' is larger variable which is of float data type and it would need 4 bytes of memory. Therefore 4 bytes space is allocated for both the variables. Both the variables have the same memory location. They are accessed according to their type. Union is efficient when members of it are not required to be accessed at the same time.

8. **How can typedef be to define a type of structure?**

typedef declaration helps to make source code of a C program more readable. Its purpose is to redefine the name of an existing variable type. It provides a short and meaningful way to call a data type. typedef is useful when the name of the data type is long. Use of typedef can reduce length and complexity of data types.
Note: Usually uppercase letters are used to make it clear that we are dealing with our own data type.

Example:

```
struct employee {
 char name[20];
 int age;
};
```

```
struct employee e;
```

The above declaration of the structure would be easy to use when renamed using typedef as:

```
struct employee {
 char name[20];
 int age;
};
typedef struct employee EMP;
      EMP e1, e2;
```

9. Write a program that returns 3 numbers from a function using a structure.

A function in C can return only one value. If we want the function to return multiple values, we need to create a structure variable, which has three integer members and return this structure.

Program: Program with a function to return 3 values

```
#include<stdio.h>
//sample structure which has three integer variables.
struct sample {
int a, b, c;
};
//this is function which returns three values.
struct sample return3val() {
 struct sample s1;
 s1.a = 10;
 s1.b = 20;
 s1.c = 30;
 //return structure s1, which means return s1.a ,s1.b and s1.c
 return s1;
}

int main() {
 struct sample accept3val;
 //three values returned are accepted by structure accept3val.
 accept3val = return3val();
 //prints the values
 printf(" \n %d", accept3val.a);
 printf("\n %d", accept3val.b);
 printf(" \n %d", accept3val.c);
 return 0;
}
```

Output:

10
20
30.

Explanation:

In this program, we use C structure to return multiple values from a function. Here we have a structure holding three int variables and a function which returns it. 'return3val' is a function which assigns 10, 20, 30 to its integer variables and returns this structure. In this program, 'accept3val' is a structure used to accept the values returned by the function. It accepts those values and shows the output.

10. **In code snippet below:**

```
struct Date {
      int yr;
      int day;
      int month;
} date1,date2;

date1.yr = 2004;
date1.day = 4;
date1.month = 12;
```

Write a function that assigns values to date2. Arguments to the function must be pointers to the structure, Date and integer variables date, month, year.

Date is a structure with three int variables as members. set_date(..) is a function used to assign values to the structure variable.

Program: Program to illustrate a function that assigns value to the structure.

```
#include<stdio.h>
#include<stdlib.h>
//declare structure Date
struct Date {
 int yr;
 int day;
 int month;
} date1, date2;

//declare function to assign date to structure variable
void set_date(struct Date *dte, int dt, int mnt, int year) {
 dte->day = dt;
 dte->yr = year;
 dte->month = mnt;
}

int main(void) {
 date1.yr = 2004;
 date1.day = 4;
 //assigning values one by one
 date1.month = 12;
 //assigning values in a single statement
 set_date(&date2, 05, 12, 2008);
 //prints both dates in date/month/year format
 printf("\n %d %d %d ", date1.day, date1.month, date1.yr);
 printf("\n %d %d %d ", date2.day, date2.month, date2.yr);
 return 0;
}
```

Output:

```
4 12 2004
5 12 2008
```

Explanation:

Two variables of type Date are created and named 'date1', 'date2'. 'date2' is assigned by using the function set_date(..). Address of 'date2' is passed to set_date function.

CHAPTER 3: Functions

1. What is the purpose of main() function?

In C, program execution starts from the main() function. Every C program must contain a main() function. The main function may contain any number of statements. These statements are executed sequentially in the order which they are written.

The main function can in-turn call other functions. When main calls a function, it passes the execution control to that function. The function returns control to main when a return statement is executed or when end of function is reached.

In C, the function prototype of the 'main' is one of the following:
int main(); //main with no arguments
int main(int argc, char *argv[]); //main with arguments

The parameters argc and argv respectively give the number and value of the program's command-line arguments.

Example:

```
#include <stdio.h>
/* program section begins here */
int main() {
// opening brace - program execution starts here
     printf("Welcome to the world of C");
     return 0;
}
// closing brace - program terminates here
```

Output:
Welcome to the world of C

2. Explain command line arguments of main function?

In C, we can supply arguments to 'main' function. The arguments that we pass to main ( ) at command prompt are called command line arguments. These arguments are supplied at the time of invoking the program.
The main ( ) function can take arguments as: main(int argc, char *argv[]) { }
The first argument argc is known as 'argument counter'. It represents the number of arguments in the command line. The second argument argv is known as 'argument vector'. It is an array of char type pointers that points to the command line arguments. Size of this array will be equal to the value of argc.

Example: at the command prompt if we give:
C:\> fruit.exe apple mango
then
argc would contain value 3
argv [0] would contain base address of string " fruit.exe" which is the command name that invokes the program.
argv [1] would contain base address of string "apple"
argv [2] would contain base address of string "mango"
here apple and mango are the arguments passed to the program fruit.exe

Program:

```
#include <stdio.h>
int main(int argc, char *argv[]) {
```

```
 int n;
 printf("Following are the arguments entered in the command line");
 for (n = 0; n < argc; n++) {
  printf("\n %s", argv[n]);
 }
 printf("\n Number of arguments entered are\n %d\n", argc);
 return 0;
}
```

Output:

```
Following are the arguments entered in the command line
C:\testproject.exe
apple
mango
Number of arguments entered are
3
```

3. **What are header files? Are functions declared or defined in header files ?**

Functions and macros are declared in header files. Header files would be included in source files by the compiler at the time of compilation.

Header files are included in source code using #include directive.#include<some.h> includes all the declarations present in the header file 'some.h'.

A header file may contain declarations of sub-routines, functions, macros and also variables which we may want to use in our program. Header files help in reduction of repetitive code.

Syntax of include directive:

```
#include<stdio.h> //includes the header file stdio.h, standard input output header into the source code
```

Functions can be declared as well as defined in header files. But it is recommended only to declare functions and not to define in the header files. When we include a header file in our program we actually are including all the functions, macros and variables declared in it.

In case of pre-defined C standard library header files ex(stdio.h), the functions calls are replaced by equivalent binary code present in the pre-compiled libraries. Code for C standard functions are linked and then the program is executed. Header files with custom names can also be created.

Program: Custom header files example

```
/****************
Index: restaurant.h
****************/
int billAll(int food_cost, int tax, int tip);

/****************
Index: restaurant.c
****************/
#include<stdio.h>
int billAll(int food_cost, int tax, int tip) {
 int result;
 result = food_cost + tax + tip;
 printf("Total bill is %d\n",result);
 return result;
}
```

```
/****************
Index: main.c
****************/
#include<stdio.h>
#include"restaurant.h"
int main() {
 int food_cost, tax, tip;
 food_cost = 50;
 tax = 10;
 tip = 5;
 billAll(food_cost,tax,tip);
 return 0;
}
```

4. **What are the differences between formal arguments and actual arguments of a function?**

Argument: An argument is an expression which is passed to a function by its caller (or macro by its invoker) in order for the function(or macro) to perform its task. It is an expression in the comma-separated list bound by the parentheses in a function call expression.

Actual arguments:
The arguments that are passed in a function call are called actual arguments. These arguments are defined in the calling function.

Formal arguments:
The formal arguments are the parameters/arguments in a function declaration. The scope of formal arguments is local to the function definition in which they are used. Formal arguments belong to the called function. Formal arguments are a copy of the actual arguments. A change in formal arguments would not be reflected in the actual arguments.

Example:

```
#include <stdio.h>
void sum(int i, int j, int k);
/* calling function */
int main() {
      int a = 5;
      // actual arguments
      sum(3, 2 * a, a);
      return 0;
}

/* called function */
/* formal arguments*/
void sum(int i, int j, int k) {
      int s;
      s = i + j + k;
      printf("sum is %d", s);
}
```

Here 3,2*a,a are actual arguments and i,j,k are formal arguments.

5. **What is pass by value in functions?**

Pass by Value: In this method, the value of each of the actual arguments in the calling function is copied into

corresponding formal arguments of the called function. In pass by value, the changes made to formal arguments in the called function have no effect on the values of actual arguments in the calling function.

Example:

```
#include <stdio.h>
void swap(int x, int y) {
 int t;
 t = x;
 x = y;
 y = t;
}
int main() {
 int m = 10, n = 20;
 printf("Before executing swap m=%d n=%d\n", m, n);
 swap(m, n);
 printf("After executing swap m=%d n=%d\n", m, n);
 return 0;
```

Output:

```
Before executing swap m=10 n=20
After executing swap m=10 n=20
```

Explanation:

In the main function, value of variables m, n are not changed though they are passed to function 'swap'. Swap function has a copy of m, n and hence it can not manipulate the actual value of arguments passed to it.

6. What is pass by reference in functions?

Pass by Reference: In this method, the addresses of actual arguments in the calling function are copied into formal arguments of the called function. This means that using these addresses, we would have an access to the actual arguments and hence we would be able to manipulate them. C does not support Call by reference. But it can be simulated using pointers.

Example:

```
#include <stdio.h>
/* function definition */
void swap(int *x, int *y) {
 int t;
 t = *x; /* assign the value at address x to t */
 *x = *y; /* put the value at y into x */
 *y = t; /* put the value at to y */
}
int main() {
 int m = 10, n = 20;
 printf("Before executing swap m=%d n=%d\n", m, n);
 swap(&m, &n);
 printf("After executing swap m=%d n=%d\n", m, n);
 return 0;
}
```

Output:
Before executing swap m=10 n=20
After executing swap m=20 n=10

Explanation:
In the main function, address of variables m, n are sent as arguments to the function 'swap'. As swap function has the access to address of the arguments, manipulation of passed arguments inside swap function would be directly reflected in the values of m, n.

7. What are the differences between getchar() and scanf() functions for reading strings?

Differences between getchar and scanf functions for reading strings:

scanf	getchar
1. Entering of each character should be followed by return key.	1. Need not type return key.
2. Continuous stream of characters cannot be directly supplied using scanf function.	2. Continuous stream of characters can be directly supplied using getchar function

3. Scanf function can be used to provide data at execution time irrespective of its type(int, char, float).
Example:
 	
#include<stdio.h>
int main() {
char a[10];
printf("Enter a: \n");
scanf("%s",a);
return 0;
}
	3. getchar() function is used only with character type.
	Example:
	
	#include<stdio.h>
	int main() {
	char a;
	printf("Enter any character: \n");
	a = getchar();
	printf("Character entered:%c \n",a);
	return 0;
	}

4. scanf() returns the number of items read successfully. A return value 0 indicates that no fields were read. EOF(end of file) is returned in case of an error or if end-of-file/end-of-string character is encountered.	4. getchar() returns the character entered as the value of the function. It returns EOF in case of an error. It is recommeded to use getchar instead of scanf.

8. Out of the functions fgets() and gets(), which one is safer to use and why?

Out of functions fgets( ) and gets( ), fgets( ) is safer to use. gets( ) receives a string from the keyboard and it is terminated only when the enter key is hit. There is no limit for the input string. The string can be too long and may lead to buffer overflow.

Example:
gets(s) /* s is the input string */

Whereas fgets( ) reads string with a specified limit, from a file and displays it on screen.The function fgets( ) takes three arguments.
First argument : address where the string is stored.
Second argument : maximum length of the string.
Third argument : pointer to a FILE.

Example:
fgets(s,20,fp); /* s: address of the string, 20: maximum length of string, fp: pointer to a file */
The second argument limits the length of string to be read. Thereby it avoids overflow of input buffer. Thus fgets( ) is preferable to gets( ).

9. **What is the difference between the functions strdup() and strcpy()?**

strcpy function: copies a source string to a destination defined by user. In strcpy function both source and destination strings are passed as arguments. User should make sure that destination has enough space to accommodate the string to be copied.

'strcpy' sounds like short form of "string copy".

Syntax:
strcpy(char *destination, const char *source);
Source string is the string to be copied and destination string is string into which source string is copied. If successful, strcpy subroutine returns the address of the copied string. Otherwise, a null pointer is returned.

Example Program:
```
#include<stdio.h>
#include<string.h>
int main() {
 char myname[10];
 //copy contents to myname
 strcpy(myname, "interviewmantra.net");
 //print the string
 puts(myname);
 return 0;
}
```

Output:
interviewmantra.net

Explanation:
If the string to be copied has more than 10 letters, strcpy cannot copy this string into the string 'myname'. This is because string 'myname' is declared to be of size 10 characters only.
In the above program, string "nodalo" is copied in myname and is printed on output screen.

strdup function: duplicates a string to a location that will be decided by the function itself. Function will copy the contents of string to certain memory location and returns the address to that location. 'strdup' sounds like short form of "string duplicate"

Syntax:
strdup (const char *s);
strdup returns a pointer to a character or base address of an array. Function returns address of the memory location where the string has been copied. In case free space could not be created then it returns a null pointer. Both strcpy and strdup functions are present in header file <string.h>

Program: Program to illustrate strdup().

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
int main() {
 char myname[] = "interviewmantra.net";
```

```
 //name is pointer variable which can store the address of memory location of string
 char* name;
 //contents of myname are copied in a memory address and are assigned to name
 name = strdup(myname);
 //prints the contents of 'name'
 puts(name);
 //prints the contents of 'myname'
 puts(myname);
 //memory allocated to 'name' is now freed
 free(name);
 return 0;
}
```

Output:
interviewmantra.net
interviewmantra.net

Explanation:
String myname consists of "interviewmantra.net" stored in it. Contents of myname are copied in a memory address and memory is assigned to name. At the end of the program, memory can be freed using free(name);

CHAPTER 4: Pointers

1. **What is a pointer in C?**

A pointer is a special variable in C language meant just to store address of any other variable or function. Pointer variables unlike ordinary variables cannot be operated with all the arithmetic operations such as '*','%' operators. It follows a special arithmetic called as pointer arithmetic.

A pointer is declared as:
int *ap;
int a = 5;

In the above two statements an integer a was declared and initialized to 5. A pointer to an integer with name ap was declared.

Next before ap is used
ap=&a;

This operation would initialize the declared pointer to int. The pointer ap is now said to point to a.

Operations on a pointer:

- **Dereferencing operator ' * ':** This operator gives the value at the address pointed by the pointer . For example after the above C statements if we give

  printf("%d",*ap);

  Actual value of a that is 5 would be printed. That is because ap points to a.

- **Addition operator ' + ':** Pointer arithmetic is different from ordinary arithmetic.

ap=ap+1;

Above expression would not increment the value of ap by one, but would increment it by the number of bytes of the data type it is pointing to. Here ap is pointing to an integer variable hence ap is incremented by 2 or 4 bytes depending upon the compiler.

2. **What are the advantages of using pointers?**

Pointers are special variables which store address of some other variables.

Syntax: datatype *ptr;
Here * indicates that ptr is a pointer variable which represents value stored at a particular address.

Example: int *p;
'p' is a pointer variable pointing to address location where an integer type is stored.

Advantages:

1. Pointers allow us to pass values to functions using *call by reference*. This is useful when large sized arrays are passed as arguments to functions. A function can return more than one value by using *call by reference*.
2. Dynamic allocation of memory is possible with the help of pointers.
3. We can resize data structures. For instance, if an array's memory is fixed, it cannot be resized. But in case of an array whose memory is created out of malloc can be resized.
4. Pointers point to physical memory and allow quicker access to data.

3. **What are the differences between malloc() and calloc()?**

Allocation of memory at the time of execution is called dynamic memory allocation. It is done using the standard library functions malloc() and calloc(). It is defined in "stdlib.h".

malloc(): used to allocate required number of bytes in memory at runtime. It takes one argument, viz. size in bytes to be allocated.

Syntax:
void * malloc(size_t size);

Example:
a = (int*) malloc(4);
4 is the size (in bytes) of memory to be allocated.

calloc(): used to allocate required number of bytes in memory at runtime. It needs *two arguments* viz.,
1. total number of data and
2. size of each data.

Syntax:
void * calloc(size_t nmemb, size_t size);

Example:
a = (int*) calloc(8, sizeof(int));

Here sizeof indicates the size of the data type and 8 indicates that we want to reserve space for storing 8 integers.

Differences between malloc() and calloc() are:

1. Number of arguments differ.
2. By default, memory allocated by malloc() contains garbage values. Whereas memory allocated by calloc() contains all zeros.

4. **How to use realloc() to dynamically increase size of an already allocated array?**

realloc(): This function is used to increase or decrease the size of any dynamic memory which is allocated using malloc() or calloc() functions.

Syntax: void *realloc(void *ptr, size_t newsize);

The first argument 'ptr' is a pointer to the memory previously allocated by the malloc or calloc functions. The second argument 'newsize' is the size in bytes, of a new memory region to be allocated by realloc. This value can be larger or smaller than the previously allocated memory. The realloc function adjusts the old memory region if newsize is smaller than the size of old memory.

If the newsize is larger than the existing memory size, it increases the size by copying the contents of old memory region to new memory region. The function then deallocates the old memory region. realloc function is helpful in managing a dynamic array whose size may change during execution.

Example: a program that reads input from standard input may not know the size of data in advance. In this case, dynamically allocated array can be used so that it is possible allocate the exact amount of memory using realloc function.

5. **What is the equivalent pointer expression for referring an element a[i][j][k][l], in a four dimensional array?**

Consider a multidimensional array a[w][x][y][z].
In this array, a[i] gives address of a[i][0][0][0] and a[i]+j gives the address of a[i][j][0][0]
Similarly, a[i][j] gives address of a[i][j][0][0] and a[i][j]+k gives the address of a[i][j][k][0]
a[i][j][k] gives address of a[i][j][k][0] and a[i][j][k]+l gives address of a[i][j][k][l]

Hence a[i][j][k][l] can be accessed using pointers as *(a[i][j][k]+l)
where * stands for value at address and a[i][j][k]+l gives the address location of a[i][j][k][l].

Program: Example program to illustrate pointer denotation of multi-dimensional arrays.

```
#include<stdio.h>
#include<string.h>
int main() {
 int a[3][3][3][3];
 //it gives address of a[0][0][0][0] .
 printf(" \n address of array a is %u", a);
 printf("\n address of a[2][0][0][0] is %u ,given by a[2], %u given by a+2",
 a[2], a + 2);
 printf("\n address of a[2][2][0][0] is %u ,given by a[2][2], %u given by a[2]+2",
 a[2][2], a[2] + 2);
 printf("\n address of a[2][2][1][0] is %u ,given by a[2][2][1] , %u given by a[2][2]+1",
 a[2][2][1], a[2][2] + 1);
 return 0;
}
```

Output:
address of array a is 65340

address of a[2][0][0][0] is 65448, given by a[2] , 65448 given by a+2
address of a[2][2][0][0] is 65484, given by a[2][2] ,65484 given by a[2]+2
address of a[2][2][1][0] is 65490, given by a[2][2][1] , 65490 given by a[2][2]+1

Explanation:
This output may differ from computer to computer as the address locations are not same for every computer.

6. **Declare an array of three function pointers where each function receives two integers and returns float.**

Declaration:

```
float (*fn[3])(int, int);
```

Program: Illustrates the usage of above declaration

```
#include<stdio.h>
float (*fn[3])(int, int);
float add(int, int);
int main() {
 int x, y, z, j;
  for (j = 0; j < 3; j++){
  fn[j] = &add;
 }
 x = fn[0](10, 20);
 y = fn[1](100, 200);
 z = fn[2](1000, 2000);
 printf("sum1 is: %d \n", x);
 printf("sum2 is: %d \n", y);
 printf("sum3 is: %d \n", z);
 return 0;
}
float add(int x, int y) {
 float f = x + y;
 return f;
}
```

Output:
sum1 is: 30
sum2 is: 300
sum3 is: 3000

Explanation:
Here 'fn[3]' is an array of function pointers. Each element of the array can store the address of function 'float add(int, int)'.

fn[0]=fn[1]=fn[2]=&add
Wherever this address is encountered add(int, int) function is called.

7. **Explain the variable assignment in the declaration**

int *(*p[10])(char *, char *);

It is an array of function pointers that returns an integer pointer. Each function has two arguments which in turn are pointers to character type variable. p[0], p[1],....., p[9] are function pointers.

return type : integer pointer.
p[10] : array of function pointers
char * : arguments passed to the function

Program: Example program to explain function pointers.

```
#include<stdio.h>
#include<stdlib.h>
int *(*p[10])(char *, char *);
//average function which returns pointer to integer whose value is average of ascii value of characters passed by pointers
int *average(char *, char *);
//function which returns pointer to integer whose value is sum of ascii value of characters passed by pointers
int *sum(char *, char *);
int retrn;
int main(void) {
 int i;
 for (i = 0; i < 5; i++) {
  //p[0] to p[4] are pointers to average function.
  p[i] = &(average);
 }
 for (i = 5; i < 10; i++) {
  //p[5] to p[9] are pointers to sum function
  p[i] = &(sum);
 }
 char str[10] = "nodalo.com";
 int *intstr[10];
 for (i = 0; i < 9; i++) {
  //upto p[4] average function is called, from p[5] sum is called.
  intstr[i] = p[i](&str[i], &str[i + 1]);
  if (i < 5) {
   //prints the average of ascii of both characters
   printf(" \n average of %c and %c is %d",
   str[i], str[i + 1],*intstr[i]);
  }
  else {
   //prints the sum of ascii of both characters.
   printf(" \n sum of %c and %c is %d",
   str[i], str[i + 1], *intstr[i]);
  }
 }
 return 0;
}
//function average is defined here
int *average(char *arg1, char *arg2) {
 retrn = (*arg1 + *arg2) / 2;
 return (&retrn);
}
//function sum is defined here
int *sum(char *arg1, char *arg2) {
 retrn = (*arg1 + *arg2);
 return (&retrn);
}
```

Output:
average of n and o is 110
average of o and d is 105
average of d and a is 98 average of d and a is 98
average of a and l is 102
average of l and o is 109
sum of o and . is 157
sum of . and c is 145
sum of c and o is 210
sum of o and m is 220

Explanation:
In this program p[10] is an array of function pointers. First five elements of p[10] point to the function: int *average(char *arg1,char *arg2). Next five elements point to the function int *sum(char *arg1,char *arg2). They return pointer to an integer and accept pointer to char as arguments.

Function average:
int *average(char *arg1,char *arg2) This function finds the average of the two values of the addresses passed to it as arguments and returns address of the average value as an integer pointer.

Function sum:
int *sum(char *arg1,char *arg2) This function finds the sum of the two values of the addresses passed to it as arguments and returns address of the sum value as an integer pointer.

8. **What is the value of**

   sizeof(a) /sizeof(char *)
   in a code snippet:

   char *a[4]={"sridhar","raghava","shashi","srikanth"};

Explanation:
Here a[4] is an array which holds the address of strings. Strings are character arrays themselves.

Memory required to store an address is 4 bits. So memory required to store 4 addresses is equal to 4*4=16 bits.

char *; is a pointer variable which stores the address of a char variable.
So sizeof(char *) is 4 bits. Therefore sizeof(a) /sizeof(char *) = 16/4 = 4 bytes.

9. **(i) What are the differences between the C statements below:**

   char *str = "Hello";
   char arr[] = "Hello";

   (ii) Whether following statements get complied or not? Explain each statement.

   arr++;
   ***(arr + 1) = 's';**
   printf("%s",arr);

(i) char *str="Hello";
"Hello" is an anonymous string present in the memory. 'str' is a pointer variable that holds the address of this string.

char arr[]="Hello";
This statement assigns space for six characters: 'H' 'e' 'l' 'l' 'o' '\0' . 'arr' is the variable name assigned to this

array of characters.

str[4] and arr[4] also have different meanings.
str[4]: adds 4 to the value of 'str' and points to the address same as value of str + 4.
arr[4]: points to the fourth element in array named 'arr'.

(ii) 'arr' is variable name of an array. A variable name can not be incremented or decremented. Hence arr++ is an invalid statement and would result in a compilation error.

*(arr+1)='s';
'arr' is the name of a character array that holds string "Hello". Usually, name of an array points to its base address. Hence value of arr is same as &arr[0].
arr+1 is address of the next element: &arr[1]

Character 's' is assigned to the second element in array 'arr', thereby string changes from "Hello" to "Hsllo".

printf("%s",arr);
This statement prints the string stored in character array 'arr'.

CHAPTER 5: Programs

1. **Write a program to find factorial of the given number.**

Recursion: A function is called 'recursive' if a statement within the body of a function calls the same function. It is also called 'circular definition'. Recursion is thus a process of defining something in terms of itself.

Program: To calculate the factorial value using recursion.

```
#include <stdio.h>
int fact(int n);
int main() {
 int x, i;
 printf("Enter a value for x: \n");
 scanf("%d", &x);
 i = fact(x);
 printf("\nFactorial of %d is %d", x, i);
 return 0;
}
int fact(int n) {
 /* n=0 indicates a terminating condition */
 if (n <= 0) {
  return (1);
 } else {
  /* function calling itself */
  return (n * fact(n - 1));
  /*n*fact(n-1) is a recursive expression */
 }
}
```

Output:
Enter a value for x:
4

Factorial of 4 is 24

Explanation:

fact(n) = n * fact(n-1)
If n=4
fact(4) = 4 * fact(3) there is a call to fact(3)
fact(3) = 3 * fact(2)
fact(2) = 2 * fact(1)
fact(1) = 1 * fact(0)
fact(0) = 1
fact(1) = 1 * 1 = 1
fact(2) = 2 * 1 = 2
fact(3) = 3 * 2 = 6
Thus fact(4) = 4 * 6 = 24

Terminating condition(n <= 0 here;) is a must for a recursive program. Otherwise the program enters into an infinite loop.

2. **Write a program to check whether the given number is even or odd.**

Program:

```
#include <stdio.h>
int main() {
 int a;
 printf("Enter a: \n");
 scanf("%d", &a);
 /* logic */
 if (a % 2 == 0) {
  printf("The given number is EVEN\n");
 }
 else {
  printf("The given number is ODD\n");
 }
 return 0;
}
```

Output:

Enter a: 2
The given number is EVEN

Explanation with examples:

Example 1: If entered number is an even number
Let value of 'a' entered is 4
if(a%2==0) then a is an even number, else odd.
i.e. if(4%2==0) then 4 is an even number, else odd.
To check whether 4 is even or odd, we need to calculate (4%2).
/* % (modulus) implies remainder value. */
/* Therefore if the remainder obtained when 4 is divided by 2 is 0, then 4 is even. */
4%2==0 is true
Thus 4 is an even number.

Example 2: If entered number is an odd number.
Let value of 'a' entered is 7

if(a%2==0) then a is an even number, else odd.
i.e. if(7%2==0) then 4 is an even number, else odd.
To check whether 7 is even or odd, we need to calculate (7%2).
7%2==0 is false /* 7%2==1 condition fails and else part is executed */
Thus 7 is an odd number.

3. **Write a program to swap two numbers using a temporary variable.**

Swapping interchanges the values of two given variables.
Logic:

step1: temp=x;
step2: x=y;
step3: y=temp;

Example:
if x=5 and y=8, consider a temporary variable temp.

step1: temp=x=5;
step2: x=y=8;
step3: y=temp=5;
Thus the values of the variables x and y are interchanged.

Program:

```
#include <stdio.h>

int main() {
 int a, b, temp;
 printf("Enter the value of a and b: \n");
 scanf("%d %d", &a, &b);
 printf("Before swapping a=%d, b=%d \n", a, b);
 /*Swapping logic */
 temp = a;
 a = b;
 b = temp;
 printf("After swapping a=%d, b=%d", a, b);
 return 0;
}
```

Output:
Enter the values of a and b: 2 3
Before swapping a=2, b=3
After swapping a=3, b=2

4. **Write a program to swap two numbers without using a temporary variable.**

Swapping interchanges the values of two given variables.
Logic:

step1: x=x+y;
step2: y=x-y;
step3: x=x-y;

Example:
if x=7 and y=4

step1: x=7+4=11;

step2: y=11-4=7;
step3: x=11-7=4;

Thus the values of the variables x and y are interchanged.

Program:

```
#include <stdio.h>

int main() {
 int a, b;
 printf("Enter values of a and b: \n");
 scanf("%d %d", &a, &b);
 printf("Before swapping a=%d, b=%d\n", a,b);
 /*Swapping logic */
 a = a + b;
 b = a - b;
 a = a - b;
 printf("After swapping a=%d b=%d\n", a, b);
 return 0;
}
```

Output:

```
Enter values of a and b: 2 3
Before swapping a=2, b=3
The values after swapping are a=3 b=2
```

5. **Write a program to swap two numbers using bitwise operators.**

Program:

```
#include <stdio.h>
int main() {
 int i = 65;
 int k = 120;
 printf("\n value of i=%d k=%d before swapping", i, k);
 i = i ^ k;
 k = i ^ k;
 i = i ^ k;
 printf("\n value of i=%d k=%d after swapping", i, k);
 return 0;
}
```

Explanation:

```
i = 65; binary equivalent of 65 is 0100 0001
k = 120; binary equivalent of 120 is 0111 1000
i = i^k;
i...0100 0001
k...0111 1000
---------
val of i = 0011 1001
---------

k = i^k
i...0011 1001
```

k...0111 1000
---
val of k = 0100 0001 binary equivalent of this is 65
---(that is the initial value of i)

i = i^k
i...0011 1001
k...0100 0001
---
val of i = 0111 1000 binary equivalent of this is 120
--- (that is the initial value of k)

6. Write a program to find the greatest of three numbers.

Program:

```
#include <stdio.h>
int main(){
 int a, b, c;
 printf("Enter a,b,c: \n");
 scanf("%d %d %d", &a, &b, &c);
 if (a > b && a > c) {
 printf("a is Greater than b and c");
 }
 else if (b > a && b > c) {
 printf("b is Greater than a and c");
 }
 else if (c > a && c > b) {
 printf("c is Greater than a and b");
 }
 else {
 printf("all are equal or any two values are equal");
 }
 return 0;
}
```

Output:

```
Enter a,b,c: 3 5 8
c is Greater than a and b
```

Explanation with examples:

Consider three numbers a=5,b=4,c=8

if(a>b && a>c) then a is greater than b and c
now check this condition for the three numbers 5,4,8 i.e.
if(5>4 && 5>8) /* 5>4 is true but 5>8 fails */

so the control shifts to else if condition
else if(b>a && b>c) then b is greater than a and c
now checking this condition for 5,4,8 i.e.
else if(4>5 && 4>8) /* both the conditions fail */

now the control shifts to the next else if condition
else if(c>a && c>b) then c is greater than a and b

now checking this condition for 5,4,8 i.e.
else if(8>5 && 8>4) /* both conditions are satisfied */

Thus c is greater than a and b.

7. Write a program to find the greatest among ten numbers.

Program:

```
#include <stdio.h>
int main() {
 int a[10];
 int i;
 int greatest;
 printf("Enter ten values:");
 //Store 10 numbers in an array
 for (i = 0; i < 10; i++) {
  scanf("%d", &a[i]);
 }
 //Assume that a[0] is greatest
 greatest = a[0];
 for (i = 0; i < 10; i++) {
  if (a[i] > greatest) {
   greatest = a[i];
  }
 }
 printf("\nGreatest of ten numbers is %d", greatest);
 return 0;
}
```

Output:

Enter ten values: 2 53 65 3 88 8 14 5 77 64 Greatest of ten numbers is 88

Explanation with example:

Entered values are 2, 53, 65, 3, 88, 8, 14, 5, 77, 64
They are stored in an array of size 10. let a[] be an array holding these values.
/* how the greatest among ten numbers is found */
Let us consider a variable 'greatest'. At the beginning of the loop, variable 'greatest' is assinged with the value of first element in the array greatest=a[0]. Here variable 'greatest' is assigned 2 as a[0]=2.

Below loop is executed until end of the array 'a[]';.

```
 for(i=0; i<10; i++)
     {
       if(a[i]>greatest)
         {
            greatest= a[i];
         }
     }
```

For each value of 'i', value of a[i] is compared with value of variable 'greatest'. If any value greater than the value of 'greatest' is encountered, it would be replaced by a[i]. After completion of 'for' loop, the value of variable 'greatest' holds the greatest number in the array. In this case 88 is the greatest of all the numbers.

8. Write a program to check whether the given number is a prime.

A prime number is a natural number that has only one and itself as factors. Examples: 2, 3, 13 are prime numbers.

Program:

```
#include <stdio.h>
main() {
 int n, i, c = 0;
 printf("Enter any number n: \n");
 scanf("%d", &n);
 /*logic*/
 for (i = 1; i <= n; i++) {
  if (n % i == 0) {
   c++;
  }
 }
 if (c == 2) {
  printf("n is a Prime number");
 }
 else {
  printf("n is not a Prime number");
 }
 return 0;
}
```

Output:

Enter any number n: 7
n is Prime

Explanation with examples:

consider a number n=5

for(i=0;i<=n;i++) /* for loop is executed until the n value equals i */
i.e. for(i=0;i<=5;i++) /* here the for loop is executed until i is equal to n */

1st iteration: i=1;i<=5;i++
here i is incremented i.e. i value for next iteration is 2
now if(n%i==0) then c is incremented
i.e.if(5%1==0)then c is incremented, here 5%1=0 thus c is incremented.
now c=1;

2nd iteration: i=2;i<=5;i++
here i is incremented i.e. i value for next iteration is 3
now if(n%i==0) then c is incremented
i.e.if(5%2==0) then c is incremented, but 5%2!=0 and so c is not incremented, c remains 1
c=1;

3rd iteration: i=3;i<=5;i++
here i is incremented i.e. i value for next iteration is 4
now if(n%i==0) then c is incremented
i.e.if(5%3==0) then c ic incremented, but 5%3!=0 and so c is not incremented, c remains 1
c=1;

4th iteration: i=4;i<=5;i++
here i is incremented i.e. i value for next iteration is 5
now if(n%i==0) then c is incremented
i.e. if(5%4==0) then c is incremented, but 5%4!=0 and so c is not incremented, c remains 1
c=1;

5th iteration: i=5;i<=5;i++
here i is incremented i.e. i value for next iteration is 6
now if(n%i==0) then c is incremented
i.e. if(5%5==0) then c is incremented, 5%5=0 and so c is incremented.
i.e. c=2

6th iteration: i=6;i<=5;i++
here i value is 6 and 6<=5 is false thus the condition fails and control leaves the for loop.

now if(c==2) then n is a prime number
we have c=2 from the 5th iteration and thus n=5 is a Prime number.

9. Write a program to check whether the given number is a palindromic number.

If a number, which when read in both forward and backward way is same, then such a number is called a palindrome number.

Program:

```
#include <stdio.h>
int main() {
 int n, n1, rev = 0, rem;
 printf("Enter any number: \n");
 scanf("%d", &n);
 n1 = n;
 /* logic */
 while (n > 0){
  rem = n % 10;
  rev = rev * 10 + rem;
  n = n / 10;
 }
 if (n1 == rev){
  printf("Given number is a palindromic number");
 }
 else{
  printf("Given number is not a palindromic number");
 }
 return 0;
}
```

Output:
Enter any number: 121
Given number is a palindrome

Explanation with an example:
Consider a number n=121, reverse=0, remainder;
number=121
now the while loop is executed /* the condition (n>0) is satisfied */

```
/* calculate remainder */
    remainder of 121 divided by 10=(121%10)=1;

  now  reverse=(reverse*10)+remainder
            =(0*10)+1     /* we have initialized reverse=0 */
            =1

       number=number/10
            =121/10
            =12

    now the number is 12, greater than 0. The above process is repeated for number=12.
        remainder=12%10=2;
        reverse=(1*10)+2=12;
        number=12/10=1;

   now the number is 1, greater than 0. The above process is repeated for number=1.
        remainder=1%10=1;
        reverse=(12*10)+1=121;
        number=1/10 /* the condition n>0 is not satisfied,control leaves the while loop */
```

Program stops here. The given number=121 equals the reverse of the number. Thus the given number is a palindrome number.

10.Write a program to check whether the given string is a palindrome.

Palindrome is a string, which when read in both forward and backward way is same.
Example: radar, madam, pop, lol, rubber, etc.,

Program:

```
#include <stdio.h>
#include <string.h>
int main() {
 char string1[20];
 int i, length;
 int flag = 0;
 printf("Enter a string: \n");
 scanf("%s", string1);
 length = strlen(string1);
 for(i=0;i < length ;i++){
  if(string1[i] != string1[length-i-1]){
   flag = 1;
   break;
  }
 }
 if (flag) {
  printf("%s is not a palindrome\n", string1);
 }
 else {
  printf("%s is a palindrome\n", string1);
 }
 return 0;
```

}

Output:
Enter a string: radar
"radar" is a palindrome

Explanation with example:
To check if a string is a palindrome or not, a string needs to be compared with the reverse of itself.

Consider a palindrome string: "radar",

---
index: 0 1 2 3 4
value: r a d a r

To compare it with the reverse of itself, the following logic is used:

0th character in the char array, string1 is same as 4th character in the same string.
1st character is same as 3rd character.
2nd character is same as 2nd character.
. . . .
ith character is same as 'length-i-1'th character.

If any one of the above condition fails, flag is set to true(1), which implies that the string is not a palindrome. By default, the value of flag is false(0). Hence, if all the conditions are satisfied, the string is a palindrome.

11. Write a program to generate the Fibonacci series.

Fibonacci series: Any number in the series is obtained by adding the previous two numbers of the series.

Let f(n) be n'th term.
f(0)=0;
f(1)=1;
f(n)=f(n-1)+f(n-2); (for n>=2)
Series is as follows
0
1
1 (1+0)
2 (1+1)
3 (1+2)
5 (2+3)
8 (3+5)
13 (5+8)
21 (8+13)
34 (13+21)
...and so on

Program: to generate Fibonacci Series(10 terms)

```
#include<stdio.h>
int main() {
 //array fib stores numbers of fibonacci series
 int i, fib[25];
 //initialized first element to 0
 fib[0] = 0;
```

```
 //initialized second element to 1
 fib[1] = 1;
 //loop to generate ten elements
 for (i = 2; i < 10; i++) {
  //i'th element of series is equal to the sum of i-1'th element and i-2'th element.
  fib[i] = fib[i - 1] + fib[i - 2];
 }
 printf("The fibonacci series is as follows \n");
 //print all numbers in the series
 for (i = 0; i < 10; i++) {
  printf("%d \n", fib[i]);
 }
 return 0;
}
```

Output:
The fibonacci series is as follows
0
1
1
2
3
5
8
13
21
34

Explanation:
The first two elements are initialized to 0, 1 respectively. Other elements in the series are generated by looping and adding previous two numbes. These numbers are stored in an array and ten elements of the series are printed as output.

12.Write a program to print "Hello World" without using semicolon anywhere in the code.

Generally when we use printf("") statement, we have to use a semicolon at the end. If printf is used inside an if condition, semicolon can be avoided.

Program: Program to print some thing with out using semicolon(;)

```
#include <stdio.h>
int main() {
 //printf returns the length of string being printed
 if (printf("Hello World\n")) //prints Hello World and returns 11
 {
 //do nothing
 }
 return 0;
}
```

Output:
Hello World

Explanation:
The if statement checks for condition whether the return value of printf("Hello World") is greater than 0. printf

function returns the length of the string printed. Hence the statement if (printf("Hello World")) prints the string "Hello World".

13.Write a program to print a semicolon without using a semicolon anywhere in the code.

Generally when use printf("") statement we have to use semicolon at the end.
If we want to print a semicolon, we use the statement: printf(";");
In above statement, we are using two semicolons. The task of printing a semicolon without using semicolon anywhere in the code can be accomplished by using the ascii value of ' ; ' which is equal to 59.

Program: Program to print a semicolon without using semicolon in the code.

```
#include <stdio.h>
int main(void) {
 //prints the character with ascii value 59, i.e., semicolon
 if (printf("%c\n", 59)) {
 //prints semicolon
 }
 return 0;
}
```

Output:

```
;
```

Explanation:
If statement checks whether return value of printf function is greater than zero or not. The return value of function call printf("%c",59) is 1. As printf returns the length of the string printed. printf("%c",59) prints ascii value that corresponds to 59, that is semicolon(;).

14.Write a program to compare two strings without using strcmp() function.

strcmp() function compares two strings lexicographically. strcmp is declared in stdio.h
Case 1: when the strings are equal, it returns zero.
Case 2: when the strings are unequal, it returns the difference between ascii values of the characters that differ.
a) When string1 is greater than string2, it returns positive value.
b) When string1 is lesser than string2, it returns negative value.

Syntax:

```
int strcmp (const char *s1, const char *s2);
```

Program: to compare two strings.

```
#include<stdio.h>
#include<string.h>
int cmpstr(char s1[10], char s2[10]);
int main() {
 char arr1[10] = "Nodalo";
 char arr2[10] = "nodalo";
 printf(" %d", cmpstr(arr1, arr2));
 //cmpstr() is equivalent of strcmp()
 return 0;
}
//s1, s2 are strings to be compared
int cmpstr(char s1[10], char s2[10]) {
 //strlen function returns the length of argument string passed
 int i = strlen(s1);
```

```
 int k = strlen(s2);
 int bigger;

 if (i < k) {
  bigger = k;
 }
 else if (i > k) {
  bigger = i;
 }
 else {
  bigger = i;
 }
 //loops 'bigger' times
 for (i = 0; i < bigger; i++) {
  //if ascii values of characters s1[i], s2[i] are equal do nothing
  if (s1[i] == s2[i]) {
  }
  //else return the ascii difference
  else {
   return (s1[i] - s2[i]);
  }
 }
 //return 0 when both strings are same
 //This statement is executed only when both strings are equal
 return (0);
}
```

Output:
-32

Explanation:
cmpstr() is a function that illustrates C standard function strcmp(). Strings to be compared are sent as arguments to cmpstr().

Each character in string1 is compared to its corresponding character in string2. Once the loop encounters a differing character in the strings, it would return the ascii difference of the differing characters and exit.

15.Write a program to concatenate two strings without using strcat() function.

strcat(string1,string2) is a C standard function declared in the header file string.h
The strcat() function concatenates string2, string1 and returns string1.

Program: Program to concatenate two strings

```
#include<stdio.h>
#include<string.h>
char *strct(char *c1, char *c2);
char *strct(char *c1, char *c2) {
 //strlen function returns length of argument string
 int i = strlen(c1);
 int k = 0;
 //loops until null is encountered and appends string c2 to c1
 while (c2[k] != '\0') {
  c1[i + k] = c2[k];
  k++;
```

```
 }
 return c1;
}

int main() {
 char string1[15] = "first";
 char string2[15] = "second";
 char *finalstr;
 printf("Before concatenation:"
 " \n string1 = %s \n string2 = %s", string1, string2);
 //addresses of string1, string2 are passed to strct()
 finalstr = strct(string1, string2);
 printf("\nAfter concatenation:");
 //prints the contents of string whose address is in finalstr
 printf("\n finalstr = %s", finalstr);
 //prints the contents of string1
 printf("\n string1 = %s", string1);
 //prints the contents of string2
 printf("\n string2 = %s", string2);
 return 0;
}
```

Output:

```
Before concatenation:
string1 = first
string2 = second
After concatenation:
finalstr = firstsecond
string1 = firstsecond
string2 = second
```

Explanation:
string2 is appended at the end of string1 and contents of string2 are unchanged.

In strct() function, using a for loop, all the characters of string 'c2' are copied at the end of c1. return (c1) is equivalent to return &c1[0] and it returns the base address of 'c1'. 'finalstr' stores that address returned by the function strct().

16.Write a program to delete a specified line from a text file.

In this program, user is asked for a filename he needs to change. User is also asked for the line number that is to be deleted. The filename is stored in 'filename'. The file is opened and all the data is transferred to another file except that one line the user specifies to delete.

Program: Program to delete a specific line.

```
#include <stdio.h>
int main() {
 FILE *fp1, *fp2;
 //consider 40 character string to store filename
 char filename[40];
 char c;
 int del_line, temp = 1;
 //asks user for file name
 printf("Enter file name: ");
```

```
 //receives file name from user and stores in 'filename'
 scanf("%s", filename);
 //open file in read mode
 fp1 = fopen(filename, "r");
 c = getc(fp1);
 //until the last character of file is obtained
 while (c != EOF)
 {
 printf("%c", c);
 //print current character and read next character
 c = getc(fp1);
 }
 //rewind
 rewind(fp1);
 printf(" \n Enter line number of the line to be deleted:");
 //accept number from user.
 scanf("%d", &del_line);
 //open new file in write mode
 fp2 = fopen("copy.c", "w");
 c = getc(fp1);
 while (c != EOF) {
 c = getc(fp1);
 if (c == '\n')
 temp++;
 //except the line to be deleted
 if (temp != del_line)
 {
 //copy all lines in file copy.c
 putc(c, fp2);
 }
 }
 //close both the files.
 fclose(fp1);
 fclose(fp2);
 //remove original file
 remove(filename);
 //rename the file copy.c to original name
 rename("copy.c", filename);
 printf("\n The contents of file after being modified are as follows:\n");
 fp1 = fopen(filename, "r");
 c = getc(fp1);
 while (c != EOF) {
 printf("%c", c);
 c = getc(fp1);
 }
 fclose(fp1);
 return 0;
}
```

Output:

```
Enter file name:abc.txt
hi.
hello
```

how are you?
I am fine
hope the same
Enter line number of the line to be deleted:4
The contents of file after being modified are as follows:
hi.
hello
how are you?
hope the same

Explanation:
In this program, user is asked for a filename that needs to be modified. Entered file name is stored in a char array 'filename'. This file is opened in read mode using file pointer 'fp1'. Character 'c' is used to read characters from the file and print them to the output. User is asked for the line number in the file to be deleted. The file pointer is rewinded back and all the lines of the file except for the line to be deleted are copied into another file "copy.c". Now "copy.c" is renamed to the original filename. The original file is opened in read mode and the modified contents of the file are displayed on the screen.

17.Write a program to replace a specified line in a text file.

Program: Program to replace a specified line in a text file.

```
#include <stdio.h>
int main(void) {
 FILE *fp1, *fp2;
 //'filename'is a 40 character string to store filename
 char filename[40];
 char c;
 int del_line, temp = 1;
 //asks user for file name
 printf("Enter file name: ");
 //receives file name from user and stores in 'filename'
 scanf("%s", filename);
 fp1 = fopen(filename, "r");
 //open file in read mode
 c = getc(fp1);
 //print the contents of file .
 while (c != EOF) {
 printf("%c", c);
 c = getc(fp1);
 }
 //ask user for line number to be deleted.
 printf(" \n Enter line number to be deleted and replaced");
 scanf("%d", &del_line);
 //take fp1 to start point.
 rewind(fp1);
 //open copy.c in write mode
 fp2 = fopen("copy.c", "w");
 c = getc(fp1);
 while (c != EOF) {
 if (c == '\n') {
 temp++;
 }
 //till the line to be deleted comes,copy the content from one file to other
 if (temp != del_line){
 putc(c, fp2);
```

```
 }
 else //when the line to be deleted comes
 {
 while ((c = getc(fp1)) != '\n') {
 }
 //read and skip the line ask for new text
 printf("Enter new text");
 //flush the input stream
 fflush(stdin);
 putc('\n', fp2);
 //put '\n' in new file
 while ((c = getchar()) != '\n')
 putc(c, fp2);
 //take the data from user and place it in new file
 fputs("\n", fp2);
 temp++;
 }
 //continue this till EOF is encountered
 c = getc(fp1);
 }
 //close both files
 fclose(fp1);
 fclose(fp2);
 //remove original file
 remove(filename);
 //rename new file with old name opens the file in read mode
 rename("copy.c", filename);
 fp1 = fopen(filename, "r");
 //reads the character from file
 c = getc(fp1);
 //until last character of file is encountered
 while (c != EOF){
 printf("%c", c);
 //all characters are printed
 c = getc(fp1);
 }
 //close the file pointer
 fclose(fp1);
 return 0;
}
```

Output:

```
Enter file name:abc.txt
hi.
hello
how are you?
hope the same
Enter line number of the line to be deleted and replaced:4
Enter new text: sayonara see you soon
hi.
hello
how are you?
sayonara see you soon
```

Explanation:
In this program, the user is asked to type the name of the file. The File by name entered by user is opened in read mode. The line number of the line to be replaced is asked as input. Next the data to be replaced is asked. A new file is opened in write mode named "copy.c". Now the contents of original file are transferred into new file and the line to be modified is deleted. New data is stored in its place and remaining lines of the original file are also transferred. The copied file with modified contents is replaced with the original file's name. Both the file pointers are closed and the original file is again opened in read mode and the contents of the original file is printed as output.

18.Write a program to find the number of lines in a text file.

Number of lines in a file can be determined by counting the number of new line characters present.

Program: Program to count number of lines in a file.

```
#include <stdio.h>
int main()
/* Ask for a filename and count number of lines in the file*/
{
 //a pointer to a FILE structure
 FILE *fp;
 int no_lines = 0;
 //consider 40 character string to store filename
 char filename[40], sample_chr;
 //asks user for file name
 printf("Enter file name: ");
 //receives file name from user and stores in a string named 'filename'
 scanf("%s", filename);
 //open file in read mode
 fp = fopen(filename, "r");
 //get character from file and store in sample_chr
 sample_chr = getc(fp);
 while (sample_chr != EOF) {
 //Count whenever sample_chr is '\n'(new line) is encountered
 if (sample_chr == '\n')
 {
 //increment variable 'no_lines' by 1
 no_lines=no_lines+1;
 }
 //take next character from file.
 sample_chr = getc(fp);
 }
 fclose(fp); //close file.
 printf("There are %d lines in %s \n", no_lines, filename);
 return 0;
}
```

Output:
Enter file name:abc.txt
There are 4 lines in abc.txt

Explanation:
In this program, name of the file to be read is taken as input. A file by the given name is opened in read-mode using a File pointer 'fp'. Characters from the file are read into a char variable 'sample_chr' with the help of getc function. If a new line character('\n') is encountered, the integer variable 'no_lines' is incremented. If the

character read into 'sample_char' is not a new line character, next character is read from the file. This process is continued until the last character of the file(EOF) is encountered. The file pointer is then closed and the total number of lines is shown as output.

19. Write a C program which asks the user for a number between 1 to 9 and shows the number. If the user inputs a number out of the specified range, the program should show an error and prompt the user for a valid input.

Program: Program for accepting a number in a given range.

```
#include<stdio.h>
int getnumber();
int main() {
 int input = 0;
 //call a function to input number from key board
 input = getnumber();
 //when input is not in the range of 1 to 9,print error message
 while (!((input <= 9) && (input >= 1))) {
  printf("[ERROR] The number you entered is out of range");
  //input another number
  input = getnumber();
 }
 //this function is repeated until a valid input is given by user.
 printf("\nThe number you entered is %d", input);
 return 0;
}
//this function returns the number given by user
int getnumber() {
 int number;
 //asks user for a input in given range
 printf("\nEnter a number between 1 to 9 \n");
 scanf("%d", &number);
 return (number);
}
```

Output:

```
Enter a number between 1 to 9
45
[ERROR] The number you entered is out of range
Enter a number between 1 to 9
4
The number you entered is 4
```

Explanation:

getfunction() function accepts input from user. 'while' loop checks whether the number falls within range or not and accordingly either prints the number(If the number falls in desired range) or shows error message(number is out of range).

20. Write a program to display the multiplication table of a given number.

Program: Multiplication table of a given number

```
#include <stdio.h>
int main() {
    int num, i = 1;
    printf("\n Enter any Number:");
```

```
    scanf("%d", &num);
    printf("Multiplication table of %d: \n", num);
    while (i <= 10) {
        printf("\n %d x %d = %d", num, i, num * i);
        i++;
    }
    return 0;
}
```

Output:

```
Enter any Number:5
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

Explanation:
We need to multiply the given number (i.e. the number for which we want the multiplication table) with value of 'i' which increments from 1 to 10.