# UNIT - III

## Functions:

A function is a self contained block of code that performs a particular task. Once a function has been designed, it can be treated as a blackbox that takes some data from the main program and returns a value.

### Function declaration:

A function consists of four parts.

* function type (return type)
* function name
* parameter list
* Terminating semicolon

Syntax:

function-type function-name (parameter list); [Prototype of function]

Ex: int mul(int m, int n); /* function prototype */

### Elements of user defined functions:

In order to make use of a user-defined function, we need to establish - three elements that are related to function.

1. function definition
2. function call
3. function declaration.

- The function definition is an independent program module that is specially written to implement the requirements of the function.
- In order to use this function we need to invoke it

at a required place in the program. This is known as function call.

- The program that calls the function is referred to as the calling program or calling function.

function implementation include following elements

1. function name 2. function type 3. list of parameters
4. local variable declaration 5. function statements 6. return statement

All the six elements are grouped into two parts

1. function Header
2. function body.

A general format of function definition

```
function_type function_name (parameter_list)
{
    local variable declaration;
    executable statement;
    ....
    ....
    return statement ;
}
```

function type and function name is known as function header and the statements within the opening and closing braces constitute the function body.

The function header (prototype) tells the compiler the name of the function, the return type & the type of arguments sending.

## Return values and their types:

- The "return" statement send back any value to the calling function.
- The return statement can take one of the following forms.

```
return;
   or
return(expression);
```

* - the plain "return" does not return any value. When a return is encountered the control is immediately passed back to the calling function.

* The return with an expression returns a value of the expression for example:

```
int mul(int x, int y)
{
  int p;
  p = x*y;
  return(p);
}
```

It returns the value of p which is the product of x and y.

* All functions by default return 'int' type data.

## Function calls:

A function can be called by using the function name followed by actual parameters.

Example:

```
main()
{
  int y;
  y = mul(10,5); /*function call*/
  printf("%d\n",y); }
```

When a compiler encounters a function call, the control is transferred to the function mul(), this function is then executed line by line as described and a value is returned when a return statement is encountered. This value is assigned to y.

## Category of functions:-

A function, depending on whether arguments are present or not and whether a value is returned or not, it may belong to one of the following categories.

Category 1: functions with no arguments and no return values.

Category 2: functions with arguments and no return values.

Category 3: functions with arguments and one return value.

Category 4: Functions with no argument but return a value

~~Category 5~~: ~~Functions return~~ multiple values.

1. In this category, the function has no arguments. It doesn't receive any data from the calling function. Similarly, it doesn't return any value. The calling function doesn't receive any data from the called function. So, there is no communication between calling and called function. Example program

2. In this category, function has some arguments. It receives data from the calling function, but it doesn't return a value from the calling function. The calling function doesn't receive any data from the called function. So it is one way data communication bln called and calling functions.

Example:

No arguments and no return value.

* To check whether the given number is prime number or not.

Program:

```
#include <stdio.h>
void check();
int main()
{
   check();
   return 0;
}
void check()
{
   int n, i, flag=0;
   Printf("Enter a positive number");
   scanf("%d", &n);
   for(i=2; i<=n/2; ++i)
   {
      if (n%i==0)
      {
         flag=1;
      }
   }
   if (flag==1)
      printf("%d is not a prime number.", n);
   else
      printf("%d is not is a prime number.", n);
}
```

Eg: Printing n Natural Numbers.

```
#include <stdio.h>
#include <conio.h>
void nat(int);
void main()
{
    int n;
    clrscr()
    Printf("\n Enter n value: ");
    Scanf("%d", &n);
    nat(n);
    getch();
}
void nat(int n);
{
    int i;
    for(i=1; i<=n; i++)
    Printf("%d\t", i);
}
```

Output:

Enter n value: 5

1 2 3 4 5

Category 3: In this category, functions has some arguments and it receives data from the calling function. Similarly, it returns value to the calling function. The calling funct receives data from the called function. So it is two-way data communication between calling and called function.

Eg: Write a 'c' program to print factorial of a given number using functions.

```
#include <stdio.h>
#include <conio.h>
int fact(int)
void main()
{
   int n;
   clrscr();
   printf("\n Enter n: ");
   scanf("%d", &n);
   printf("\n Factorial of the number: %d", fact(n));
   getch();
}
int fact(int n)
{
   int i,f;
   for(i=1, f=1; i<=n; i++)
   f = f*i;
   return(f);
}
```

Output:

Enter n: 5

Factorial of the number: 120.

Category: 4: In this category, the functions has no arguments, i.e ~~and~~ it doesn't receive any data from the calling function, but it returns value to the calling function. the calling function receives data from the called function. So, it is one way data communication b/n called and calling function.

Eg: To print sum of 'n' elements.

```
#include <stdio.h>
#include <conio.h>
int sum();
void main()
{
  int s;
  clrscr();
  Printf("\n Enter number of elements to be added : ");
  s= sum();
  Printf("\n sum of the elements : %d", p);
  getch();
}
int sum()
{
  int a[20], i, s=0, n;
  scanf("%d", &n);
  Printf(" Enter the elements ");
  for (i=0; i<n; i++)
   scanf("%d", &a[i]);
   for (i=0; i<n; i++)
     s= s+a[i];
     return s;
}
```

## Nesting of functions :-

'C' permits nesting of functions freely. main can call function 1, which calls function 2, which calls function 3 and so on.

## Scope and Extent :-

* The scope of any variable is actually a subset of lifetime.
* A variable may be in the memory but maynot be accessible though.
* So the area of our program where we can actually access our entity or variable is the scope of that variable.

The scope of a variable categorized into three categories.

* **Global scope:** When variable is defined outside all functions, it is then available to all the functions of the program and all the blocks program contains.
* **Local scope:** When a variable is defined inside a function or a block, then it is locally accessible within the block and hence it is a local variable.
* **Function scope:** When a variable is passed as formal arguments it is said to have function scope.

**Example:**

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

O/P :→ 10 20

Notes on storage class

## Storage classes :-

A storage class defines the scope and life-time of variables and functions within a 'c' program.

We have four different storage classes in c

1. auto 2. register 3. static 4. extern.

### The 'auto' storage class:

This is the default storage class for all local variables.

```
Ex: {
      int count;
      auto int month;
    }
```

The example defines two variables with in the same storage class. auto can only be used within the functions i.e., local variables.

### * Register variables (register storage class).

Register inform the compiler that a variable should be kept in one of the machines registers.

We should use register storage class only for those variables that are used in our program very oftenly.

* Syntax: register int count;

### * static :-

The value of static variable persists until the end of the program. A variable can be declared static using the keyword static like,

```
static int x;
static float y;
```

The static storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying each time.

Example:

```
#include <stdio.h>
void func(void)   /* function declaration */.
static int count=5; /* global variable */
main()
{
  while (count--)
  {
    func();
  }
  return 0;
}
/* function definition */
void func(void) {
static int i=5;  /*local static variable */
 i++;
 Printf ("i is %d and count is %d \n", i, count);
}
```

Output:

i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0

* The extern storage class:-

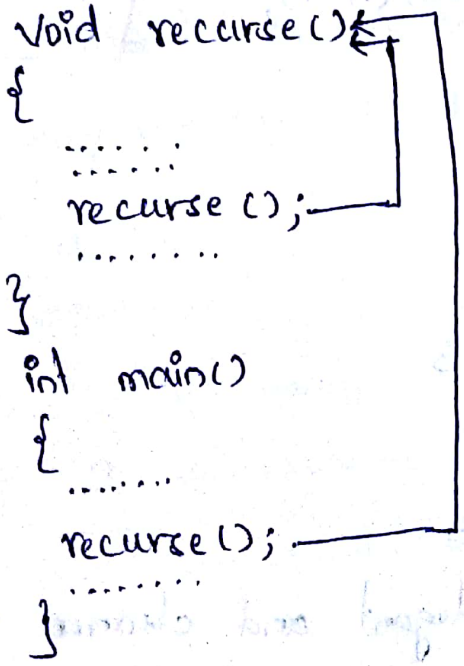
The extern storage class is used to give a reference of a global variable that is visible to all program files. When we use extern the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

## Recursive functions:-

When a called function in turn calls another function a process of chaining occurs. Recursion is the special case of this process, where a function calls itself.

A function that calls itself is known as recursive funct and this technique is known as recursion.

## How recursion works:




* To prevent infinite recursion, if...else condition can be used.

Example: Sum of Natural Numbers using Recursion.

```
#include <stdio.h>
int sum (int n);
int main()
{
    int number, result;
    Printf ("Enter a positive number: ");
    scanf ("%d", &number);
    result = sum(number);
    Printf ("sum=%d", result);
}
int sum (int num)
{
    if (num != 0)
        return num + sum (n-1);
    else
        return num;
}.
```

Output:

Enter a positive number: 3

sum = 6.

Advantages:

Recursion makes program elegant and cleaner. All algorithms can be defined recursively which makes it easier to visualize and prove.

## pointers:

→ A pointer is a derived datatype in C.
→ pointers contain memory addresses of their values.
→ pointers are variables that hold address of another variable of same data type.
→ It provides power and flexibility to the language.

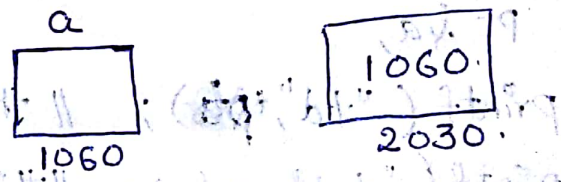
## Benefits of using pointers:

→ pointers are more efficient in handling array and structures.
→ pointers allows references to function & thereby helps in passing of function as arguments to other functions.
→ It reduces the length & the program execution time.
→ It allows C to support dynamic memory management.

## Concept of pointers:

Whenever a variable declared, System will allocate a location to that variable in the memory, to hold value. This location will have its own address number.

Ex: int a = 10;
int *p;
p = &a;




10 ← value.
a ← Name of location/variable
1060 ← address of variable

**Pointer variable:** the variable that holds the address of other variable is called pointer variable.

## Declaring a pointer variable:

Syntax: datatype *pointer_name;

Note: Datatype of pointer must be same as the variable, which the pointer is pointing.

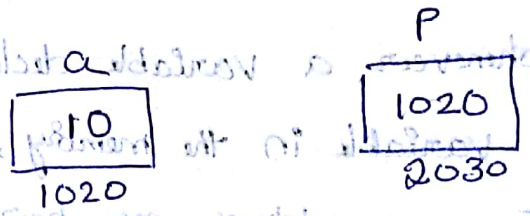
## Initialization of pointer variable:

It is the process of assigning address of a variable to pointer variable.

→ pointer variable contains address of variable of same data type.

→ The address operator '&' is used to determine the address of a variable.

Ex: int a = 10;
int *ptr;
ptr = &a;




Ex: int a, *P;
a = 10;
P = &a;
printf("%d", *P); // this will print the value of 'a'
printf("%d", *&a); // this will print the value of 'a'
printf("%u", &a); // this will print the address of 'a'
printf("%u", P); // this will also print the address of 'a'.
printf("%u", &P); // this will print the address of P.

value in P pointed address

*P = *(1020)

↓ value in P

*P = *&a = *(1020)

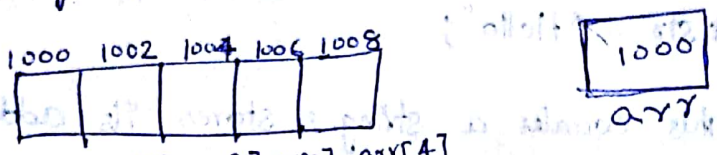
value in 1020 address

## Pointer & Arrays:

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array.
→ Base address which gives location of the first element is also allocated by the compiler.

Ex: `int arr[5] = {1,2,3,4,5};`

Assume that base address of arr is '1000' & each integer requires 2 bytes, the 5 elements will be stored as follows:




| element | arr[0] | arr[1] | arr[2] | arr[3] | arr[4] |
|---|---|---|---|---|---|
| address | 1000 | 1002 | 1004 | 1006 | 1008 |

Ex: `int *p;`
`p = arr;` (or) `p = &arr[0];`

## Pointer to Array:

```
int i;
int a[5] = {1,2,3,4,5};
int *p = a;
for(i=0; i<5; i++)
{
printf("%d", *p);
p++;
}
```

Note: we can increment a pointer variable, but we can't decrease once we increments.

```
printf("%d", a[i]);   // print the array, by incrementing index.
printf("%d", i[a]);   // this will also print elements of array.
printf("%d", a+i);    // this will print address of all the array elements.
```

printf ("%d", *(a+i)); // it will print value of array element.
printf ("%d", *a); // will print value of a[0] only.

## Pointer to Multidimensional Array:

a[i][j] → *(*(ptr+i)+j)

## pointer and character strings:

→ pointer variables of 'char' type are treating as string.

```
char *str = "Hello";
```

this creates a string & stores its address in the pointer variable 'str'. The pointer 'str' now points to the first character of the string "Hello".

// it will assign at runtime.

```
char *str;
str = "hello";
```

## Array of pointers:

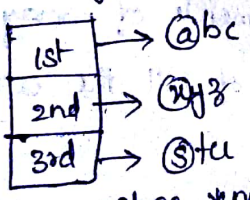
We can also have array of pointers. pointers are very helpful in handling character array without rows of varying length.

```
char *name[3] = { "Adam", "xyz", "Stu" };
```
abc

// the same array without using pointer

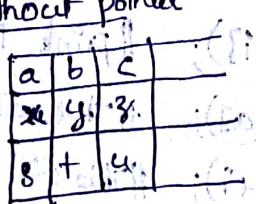
```
char name[3][20] = { "abc", "xyz", "stu" };
```

using pointer




char *name[3] → allocates upto 3 char

without pointer




char name[3][20] → extends till 20 memory location

## pointers As a function arguments:

Here we can pass the address of a variable as an argument to a function in the normal function.

when we pass address of a function, the parameters receiving the addresses should be pointers.

The process of calling a function using pointers to pass the addresses of variables is known as "call by reference".

Note: The process of passing the actual value of variable is known as "call by value".

The function which is called by 'reference' can change the value of the variable used in the call.

Ex:
```
main()
{
  int x;
  x = 10;
  change (&x);   // call by reference (or) address
  printf ("%d", x);
}
change (int *p)
{
  *p = *p+10;   // It increments the value of x where it is stored. Pointer passed to a pointer variable p through a function.
}
```

/* program to swap by using function through pointers

```
void exchange (int *, int *);
main()
{ int x, y;
  x = 100;
```

```
    y=200;
    printf("Before exchange : x=%d y=%d \n", x, y);
    exchange(&x, &y);          /* calling a function */
    printf("After exchange : x=%d, y=%d \n", x, y);
}
exchange(int *a, int *b)
{
    int s;
    s = *a;
    *a = *b;
    *b = s;
}
```

pointers to functions:

Like a variable for function also having a type and an address location in the memory. so we can declare a pointer to a function. which can then be used as an argument in another function.

→ declaration of pointer to a function:

[~~type (*fptr)();~~]

Ex: double mul(int, int);
double (*P1)();
P1 = mul; // ref a ptr variable P1 to a function 'mul()'

→ Here 'mul' is a function
'*P1' is a pointer for ref to a function

Now for calling a function 'mul' we can use a pointer variable P1

Ex: (*P1)(x, y) it is equivalent to mul(x, y)

## Compatibility:

The possibility of assigning a pointer variable to any ~~of datatype~~ variable with any datatype is called "compatibility."

To achieve this property we should declare a pointer variable with 'void' type.

Ex: void *p;

→ A void pointer also known as generic pointer, is a special type of pointer that can be pointed at objects of any data type.

Ex:
```
#include <stdio.h>
void main()
{
int a=10;
float b=25.10;
void *ptr; // declaring a void pointer
ptr = &a; // assigning address of integer to void pointer.
printf("The value of integer variable = %d", *((int*)ptr));
// Here it uses type casting
ptr = &b; // assigning address of float to void pointer.
printf("The value of float variable = %f", *((float*)ptr));
}
```

## Memory allocation functions:

Dynamic data structures provide flexibility in adding, deleting (or) rearranging data items at run time.

→ The process of allocating memory at run time is known as "dynamic memory allocation".

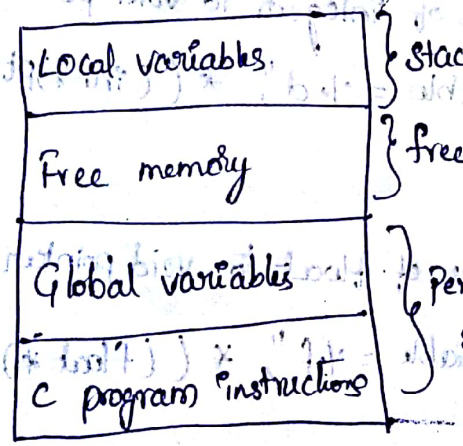
malloc : It allocates request size of bytes and returns a pointer to the first byte of the allocated space.

calloc : Allocates space for an array of elements, initializes them to zero & then returns a pointer to the memory.

free : Frees previously allocated space.

realloc : Modifies the size of previously allocated space.

## memory allocating process:




Permanent storage area:
→ It stores program instructions global & static variables.

Local stack: It stores the local variables

heap: It is a free memory region. & is available for dynamic allocation during execution of a program.

Note: The size of heap keeps changing when program is executed due to creation & death of variables that are local to functions & blocks.

→ There is a possibility of "overflow" during dynamic allocation process.

**malloc**: Allocating a block of memory:

This function reserves a block of memory of specified size and returns a pointer of type 'void'.

Syntax: ptr = (cast-type *) malloc (byte-size);

Ex: x = (int *) malloc (100 * sizeof (int));

Here it allocates 100 times of size of int & the address of first byte of memory is assigned to pointer 'x' of type 'int'.

Ex: cptr = (char *) malloc (10);

It allocates 10 bytes of space for the pointer 'cptr' of type 'char'

All the memory locations contains garbage values

**Calloc**: Allocating multiple blocks of memory:

It allocates multiple blocks of storage & it is normally used for allocating requested memory space at runtime for storing derived data types such as arrays & structures.

Syntax: ptr = (cast-type *) calloc (n, elem-size);

This statement allocates contiguous space for 'n' blocks.

st_ptr = (record *) calloc (class_size, sizeof (record));

**free**: Releasing the used space:

Programs on malloc & calloc

1. What is the output of this C code?

```
1.     #include <stdio.h>
2.     void main()
3.     {
4.         m();
5.         void m()
6.         {
7.             printf("hi");
8.         }
9.     }
```

a) hi  b) Compile time error  c) Nothing  d) Varies

2. What is the output of this C code?

```
1.     #include <stdio.h>
2.     void main()
3.     {
4.         m();
5.     }
6.     void m()
7.     {
8.         printf("hi");
9.         m();
10.     }
```

a)Compile time error  b) hi  c) Infinite hi  d) Nothing

3. What is the output of this C code?

```
1.     #include <stdio.h>
2.     void main()
3.     {
4.         static int x = 3;
5.         x++;
6.         if (x <= 5)
7.         {
8.             printf("hi");
9.             main();
10.         }
11.     }
```

a) Run time error  b) hi  c) Infinite hi  d) hi hi

4. Which of the following is a correct format for declaration of function?

a) return-type function-name(argument type);  b) return-type function-name(argument type){}

c) return-type (argument type)function-name;  d) all of the mentioned

5. Which of the following function declaration is illegal?

a) int 1bhk(int);  b) int 1bhk(int a);  c) int 2bhk(int*, int []);  d) all of the mentioned

6. Which function definition will run correctly?

a) int sum(int a, int b)
   return (a + b);

b) int sum(int a, int b)
   {return (a + b);}

c) int sum(a, b)
   return (a + b);

d) none of the mentioned

7. Can we use a function as a parameter of another function? [ Eg: void wow(int func()) ].

a) Yes, and we can use the function value conveniently

b) Yes, but we call the function again to get the value, not as convenient as in using variable

c) No, C does not support it

d) This case is compiler dependent

8. The value obtained in the function is given back to main by using _____ keyword?

a) return

b) static

c) new

d) volatile

9. What is the return-type of the function sqrt()

a) int

b) float

c) double

d) depends on the data type of the parameter

10. Which of the following function declaration is illegal?

a) double func();
   int main(){}
   double func(){}

b) double func(){};
   int main(){}

c) int main()
   {
   double func();
   }
   double func(){//statements}

d) None of the mentioned

11. What is the output of this code having void return-type function?

```
1.     #include <stdio.h>
2.     void foo()
3.     {
4.         return 1;
5.     }
6.     void main()
7.     {
8.         int x = 0;
9.         x = foo();
10.         printf("%d", x);
11.     }
```

a) 1 b) 0 c) Runtime error d) Compile time error

12. What will be the data type returned for the following function?

```
1.     #include <stdio.h>
2.     int func()
3.     {
4.         return (double)(char)5.0;
5.     }
```

a) char b) int c) double d) multiple type-casting in return is illegal

13. What is the problem in the following declarations?

int func(int);

double func(int);

int func(float);

a) A function with same name cannot have different signatures

b) A function with same name cannot have different return types

c) A function with same name cannot have different number of parameters

d) All of the mentioned

14.Which of the following storage class supports char data type?

**A.** register **B.** static **C.** auto **D.** All of the mentioned

15.. The output of the code below is

```
1.     #include <stdio.h>
2.     void main()
3.     {
4.         int k = m();
5.         printf("%d", k);
6.     }
7.     void m()
8.     {
9.         printf("hello");
```

```
10.     }
```

a) hello 5  b) Error  c) Nothing  d) Junk value

16. The output of the code below is

```
1.     #include <stdio.h>
2.     int *m()
3.     {
4.         int *p = 5;
5.         return p;
6.     }
7.     void main()
8.     {
9.         int *k = m();
10.         printf("%d", k);
11.     }
```

a) 5  b) Junk value  c) 0  d) Error

17. The output of the code below is

```
1.     #include <stdio.h>
2.     int *m();
3.     void main()
4.     {
5.         int *k = m();
6.         printf("hello ");
7.         printf("%d", k[0]);
8.     }
9.     int *m()
10.     {
11.         int a[2] = {5, 8};
12.         return a;
13.     }
```

a) hello 5 8  b) hello 5  c) hello followed by garbage value  d) Compilation error

18. Which of the following is true for static variable?

**A.** It can be called from another function.

**B.** It exists even after the function ends.

**C.** It can be modified in another function by sending it as a parameter.

**D.** All of the mentioned

19.Functions have static qualifier for its declaration by default.

**A.** true  **B.** false  **C.** Depends on the compiler  **D.** Depends on the standard

20.Automatic variables are initialized to?

**A.** 0 **B.** Junk value **C.** Nothing **D.** Both (a) & (b)

21.The variable declaration with no storage class specified is by default:

**A.** auto **B.** extern **C.** static **D.** register

22.What linkage does automatic variables have?

**A.** Internal linkage **B.** External linkage **C.** No linkage **D.** None of the mentioned

23.Automatic variables are variables that are?

**A.** Declared within the scope of a block, usually a function

**B.** Declared outside all functions

**C.** Declared with auto keyword

**D.** Declared within the keyword extern

24.Is initialization mandatory for local static variables?

**A.** YES **B.** NO **C.** Depends on the compiler **D.** Depends on the standard

25. What is the default return type if it is not specified in function definition?

**A.** void **B.** int **C.** double **D.** short int

29.Functions in C are ALWAYS:

**A.** Internal **B.** External **C.** Both Internal and External **D.** External and Internal are not valid terms for functions

30.Global variables are:

**A.** Internal **B.** External **C.** Both (a) and (b) **D.** None of the mentioned.

31.What will be the output?

```
double var = 8;
int main()
{
    int var = 5;
    printf("%d", var);
}
```

**A.** 5

**B.** 8

**C.** Compile time error due to wrong format identifier for double

**D.** Compile time error due to redeclaration of variable with same name

32.What is the output of this C code?

```
double i;
int main()
```

```
{
   printf("%g\n",i);
   return 0;
}
```

**A.** 0 **B.** 0.000000 **C.** Garbage value **D.** Depends on the compiler

33.Can variable i be accessed by functions in another source file?

```
int i;
int main()
{
   printf("%d\n", i);
}
```

**A.** 0 **B.** false **C.** Only if static keyword is used **D.** Depends on the type of the variable

34.The scope of an automatic variable is:

**A.** Within the block it appears

**B.** Within the blocks of the block it appears

**C.** Until the end of program

**D.** Both (a) and (b)

35.Automatic variables are allocated space in the form of a:

**A.** stack **B.** queue **C.** priority queue **D.** random

36.Automatic variables are stored in:

**A.** stack **B.** data segment **C.** register **D.** heap

37. What is the output of this C code?

```
int main()
{
int i = 10;
void *p = &i;
printf("%d\n", (int)*p);
return 0;
}
```

**A.** Compile time error

**B.** Segmentation fault/runtime crash

**C.** 10

**D.** Undefined behavior

38.Comment on the following?

```
const int *ptr;
```

**A.** You cannot change the value pointed by ptr

**B.** You cannot change the pointer ptr itself

**C.** Both (a) and (b)

**D.** You can change the pointer as well as the value pointed by it

39.Which is an indirection operator among the following?

- **A.** & **B.** * **C.** -> **D.** .

40.What is the output of this C code?

```
void main()
{
int x = 0;
int *ptr = &5;
printf("%p\n", ptr);
}
```

**A.** 5

**B.** Address of 5

**C.** Nothing

**D.** Compile time error

41.What is the output of this C code?

```
void main()
{
int x = 0;
int *ptr = &x;
printf("%d\n", *ptr);
}
```

- **A.** Address of x **B.** Junk value **C.** 0 **D.** Run time error

42.What is the output of this C code?

```
void main()
{
int k = 5;
int *p = &k;
int **m = &p;
printf("%d%d%d\n", k, *p, **m);
}
```

**A.** 5 5 5

**B.** 5 5 junk value

**C.** 5 junk value

**D.** Run time error

43.How many number of pointer (*) does C have against a pointer variable declaration?

**A.** 7 **B.** 127 **C.** 255 **D.** No limits

44.Which of the following declaration throw run-time error?

**A.** int **c = &c;

**B.** int **c = &*c;

**C.** int **c = **c;

**D.** None of the mentioned.

45.The maximum number of arguments that can be passed in a single function are_____
a) 127 b) 253 c) 361 d) No limits in number of arguments