UNIT–IV:

Functions:: Introduction, Function Definition , Function Declaration, Function calls, Return values and their types, Categories of functions, Recursion, Storage classes, Passing arrays to functions .

Pointers: Pointers and addresses, Pointer expressions and Pointer arithmetic, Pointers and Functions, void pointer, Pointers and arrays, Pointers and strings, Array of pointers, Pointers to pointers.

Dynamic memory allocation: malloc, calloc, realloc, free

1. What is a function? Why we use functions in C language? Give an example.

Ans: Function in C:

A function is a block of code that performs a specific task. It has a name and it is reusable .It can be executed from as many different parts in a program as required, it can also return a value to calling program.

All executable code resides within a **function**. It takes input, does something with it, then give the answer. A C program consists of one or more functions.

A computer program cannot handle all the tasks by itself. It requests other program like entities called functions in C. We pass information to the function called **arguments** which specified when the function is called. A function either can return a value or returns nothing. Function is a subprogram that helps reduce coding.

Simple Example of Function in C

```
#include<stdio.h>
#include <conio.h>
int adition (int, int); //Function Declaration int
addition (int a, int b) //Function Definition
{
int r;
r=a + b;
return (r);
}
int main()
```

```
{
int z;
z= addion(10,3); //Function Call
printf ("The Result is %d", z);
return 0;
}
```

Output: The Result is 13

Why use function:

Basically there are **two reasons** because of which we use functions

1. Writing functions avoids rewriting the same code over and over. For example - if you have a section of code in a program which calculates the area of triangle. Again you want to calculate the area of different triangle then you would not want to write the same code again and again for triangle then you would prefer to jump a "section of code" which calculate the area of the triangle and then jump back to the place where you left off. That section of code is called „function'.

2. Using function it becomes easier to write a program and keep track of what they are doing. If the operation of a program can be divided into separate activities, and each activity placed in a different function, then each could be written and checked more or less independently. Separating the code into modular functions also makes the program easier to design and understand.

2. Distinguish between Library functions and User defined functions in C and Explain with examples.

Ans: Types of Function in C:

(i). Library Functions in C

C provides library functions for performing some operations. These functions are present in the c library and they are predefined.

For example sqrt() is a mathematical library function which is used for finding the square root of any number .The function scanf and printf() are input and output library function similarly we have strcmp() and strlen() for string manipulations. To use a library function we have to include some header file using the preprocessor directive #include.

For example to use input and output function like printf() and scanf() we have to include stdio.h, for math library function we have to include math.h for string library string.h should be included.

(ii). User Defined Functions in C

A user can create their own functions for performing any specific task of program are called user defined functions. To create and use these function we have to know these 3 elements.

1. Function Declaration
2. Function Definition
3. Function Call

1. Function declaration

The program or a function that calls a function is referred to as the calling program or calling function. The calling program should declare any function that is to be used later in the program this is known as the function declaration or function prototype.

2. Function Definition

The function definition consists of the whole description and code of a function. It tells that what the function is doing and what are the input outputs for that. A function is called by simply writing the name of the function followed by the argument list inside the parenthesis. Function definitions have two parts:

Function Header

The first line of code is called Function Header.
int sum(int x, int y)
It has three parts
(i). The name of the function i.e. sum

(ii). The parameters of the function enclosed in parenthesis
(iii). Return value type i.e. int

Function Body
Whatever is written with in { } is the body of the function.

3. Function Call

In order to use the function we need to invoke it at a required place in the program. This is known as the function call.

3. Write some properties and advantages of user defined functions in C?

Ans:

Properties of Functions

- Every function has a unique name. This name is used to call function from “main()” function.
- A function performs a specific task.

- A function returns a value to the calling program.

Advantages of Functions in C

- Functions has top down programming model. In this style of programming, the high level logic of the overall problem is solved first while the details of each lower level functions is solved later.
- A C programmer can use function written by others
- Debugging is easier in function
- It is easier to understand the logic involved in the program
- Testing is easier

4. Explain the various categories of user defined functions in C with examples?

Ans:

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

(i) Functions with no arguments and no return values.

(ii) Functions with arguments and no return values.

(iii) Functions with arguments and return values.

(iv) Functions with no arguments and return values.

(i) Functions with no arguments and no return values:-

When a function has no arguments, it does not return any data from calling function. When a function does not return a value, the calling function does not receive any data from the called function. That is there is no data transfer between the calling function and the called function.

Example

```
#include   <stdio.h>
#include  <conio.h>
void printmsg()
{
printf ("Hello ! I Am A Function .");
}
```

```
int main()
{
printmsg();
return 0;
}
```

Output : Hello ! I Am A Function .

(ii) Functions with arguments and no return values:-

When a function has arguments data is transferred from calling function to called function. The called function receives data from calling function and does not send back any values to calling function. Because it doesn‟t have return value.

Example

```
#include<stdio.h>
#include <conio.h>
void add(int,int);

void main()
{
int a, b;

printf(“enter value”);

scanf(“%d%d”,&a,&b);

add(a,b);
}

void add (intx, inty)
{
int z ;
z=x+y;
```

```
printf ("The sum =%d",z);
}
```

output : enter values 2 3

The sum = 5

(iii) Functions with arguments and return values:-

In this data is transferred between calling and called function. That means called function receives data from calling function and called function also sends the return value to the calling function.

Example

```
#include<stdio.h>
#include <conio.h>
int add(int, int);
main()
{
 int a,b,c;

printf(“enter value”);

scanf(“%d%d”,&a,&b);

c=add(a,b);
printf ("The sum =%d",c);
 }
int add (int x, int y)
{
int z;
z=x+y;

return z;
}
```

output : enter values 2 3

The sum = 5

(iv) Function with no arguments and return type:-

When function has no arguments data cannot be transferred to called function. But the called function can send some return value to the calling function.

Example

```
#include<stdio.h>
#include <conio.h>
int add( );
main()
{
 int c;

c=add();
printf ("The sum =%d",c);
 }
int add ()
{
int x,y,z;

printf(“enter value”);

scanf(“%d%d”,&a,&b);

z=x+y;

return z;
}
```

Output: enter values 2 3

The sum = 5

5. Explain the Parameter Passing Mechanisms in C-Language with examples.

Ans:

Most programming languages have 2 strategies to pass parameters. They are

(i) pass by value

(ii) pass by reference

(i) Pass by value (or) call by value :-

In this method calling function sends a copy of actual values to called function, but the changes in called function does not reflect the original values of calling function.

Example program:

```
#include<stdio.h>

void fun1(int, int);

void main( )

{

int a=10, b=15;

fun1(a,b);

printf("a=%d,b=%d", a,b);

}

void fun1(int x, int y)

{

x=x+10;

y= y+20;

}
```

Output: a=10 b=15

The result clearly shown that the called function does not reflect the original values in main function.

(ii) Pass by reference (or) call by address :-

In this method calling function sends address of actual values as a parameter to called function, called function performs its task and sends the result back to calling function. Thus, the changes in called function reflect the original values of calling function. To return multiple values from called to calling function we use pointer variables.

Calling function needs to pass „&" operator along with actual arguments and called function need to use „*" operator along with formal arguments. Changing data through an address variable is known as indirect access and „*" is represented as indirection operator.

Example program:

```
#include<stdio.h>

void fun1(int,int);

void main( )

{

int a=10, b=15;

fun1(&a,&b);

printf("a=%d,b=%d", a,b);

}

void fun1(int *x, int *y)

{

*x = *x + 10;
```

```
*y = *y + 20;

}
```

Output: a=20 b=35

The result clearly shown that the called function reflect the original values in main function. So that it changes original values.

6. Differentiate actual parameters and formal parameters.

Ans:

Actual parameters	**Formal parameters**
The list of variables in calling function is known as **actual parameters**.	The list of variables in called function is known as **formal parameters**.
Actual parameters are variables that are declared in function call.	Formal parameters are variables that are declared in the header of the function definition.
Actual parameters are passed without using type	Formal parameters have type preceeding with them.
main( ) { …… function_name (actual parameters); ….. }	return_type function_name(formal parameters) { …… function body; ….. }

Formal and actual parameters must match exactly in type, order, and number.

Formal and actual parameters need not match for their names.

7. Explain in detail about nesting of functions with example.

Ans:

Nesting of functions

The process of calling a function within another function is called nesting of function

Syntax:-

```
main()
{
……..
Function1();
…….
}
Function1();
{
 ……
Function2();
……
}
Function2();
{
……
Function3();
……
}
Function3();
{
…….
}
```

main () can call Function 1() where Function1 calls Function2() which calls Function3() and so on

Ex:

```
float ratio (int, int,
int); int difference (int,
int); void main()
{
int a,b,c,d;
printf(" enter three numbers");
scanf("%d%d%d", &a, &b, &c );
d= ratio(a,b,c);
printf ("%f\n" ,d);
}
float ratio(int x,int y,int z)
{
int u ;
u=difference(y,z);
if (u) return(x/(y-
z)); else
return (0.0);
}
int difference (int p, int q)
{
if(p!=q)
```

```
return 1;

else

return 0;

}
```

main reads values a,b,c and calls ratio() to calculate a/(b-c) ratio calls another function difference to test whether (b-c) is zero or not this is called nesting of function

8. What is recursive function? Write syntax for recursive functions.

Ans:

Recursion

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied.

When a function calls itself, a new set of local variables and parameters are allocated storage on the stack, and the function code is executed from the top with these new variables. A recursive call does not make a new copy of the function. Only the values being operated upon are new. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes immediately after the recursive call inside the function.

The main advantage of recursive functions is that we can use them to create clearer and simpler versions of several programs.

Syntax:-

A function is **recursive** if it can call itself; either directly:

```
void f( )
{
f( );
}
```

(or) indirectly:

```
void f( )
{
g( );
}

void g( )
```

```
{
f( );
}
```

Recursion rule 1: Every recursive method must have a **base case** -- a condition under which no recursive call is made -- to prevent infinite recursion.

Recursion rule 2: Every recursive method must make progress toward the base case to prevent infinite recursion

9. Write a program to find factorial of a number using recursion.

Ans:

```
#include<stdio.h>
int fact(int);
main()
{
int n,f;
printf("\n Enter any
number:"); scanf("%d",&n);
f=fact(n);
printf("\n Factorial of %d is %d",n,f);
}
int fact(int n)
{
int f;
if(n==0||n==1) //base case
f=1;
else
f=n*fact(n-1); //recursive case
return f;
}
```

Output:- Enter any number: 5

Factorial of 5 is 120

10. Differentiate between recursion and non-recursion. Ans:

Recursion:- Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied.

When a function calls itself, a new set of local variables and parameters are allocated storage on the stack, and the function code is executed from the top with these new variables. A recursive call does not make a new copy of the function. Only the values being operated upon are new. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes immediately after the recursive call inside the function.

Ex:-

```
void main( )
{
int n=5;
fact( n);
}
int fact( )
{
if(n==0 || n==1)
  return 1;
else
return(n*fact(n-1));
}
```

Non-Recursion:-

Using looping statements we can handle repeated statements in „C". The example of non recursion is given below.

Syntax:-

```
void main( )
{
int n=5;
res = fact(n);
printf("%d",res);
}
int fact( )
{
for(i=1;i<=n;i++)
{
f=f+1;
}
return f;
}
```

Differences:

- Recursive version of a program is slower than iterative version of a program due to overhead of maintaining stack.
- Recursive version of a program uses more memory (for the stack) than iterative version of a program.
- Sometimes, recursive version of a program is simpler to understand than iterative version of a program.

11. Write a program to calculate GCD of two numbers using recursion

Ans:

```
#include<stdio.h>

int gcd(int,int);

main()

{

int a,b;

printf("\n Enter any two

numbers:"); scanf("%d%d",&a,&b);

printf("\nGCD=%d",gcd(a,b));

}

int gcd(int a,int b)

{

if(b==0) //base case

return a;

else

return gcd(b,a%b); //recursive case

}
```

12. Write a program to generate Fibonacci series using recursive functions.

Ans:

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int f,f1,f2,n,i,res;
printf("enter the limit:");
scanf("%d",&n);
printf("The fibonacci series is:");
for(i=0;i<n;i++)
{
res=fib(i);
printf("%d\t",res);
}
}
int fib(int i)
{
if(i==0)
return 0;
else if(i==1)
return 1;
else
return(fib(i-1)+fib(i-2));
}
```

13. How to Pass Array Individual Elements to Functions? Explain with example program.

Ans:

Arrays with functions:

To process arrays in a large program, we need to pass them to functions. We can pass arrays in two ways:

1) pass individual elements

2) pass the whole array.

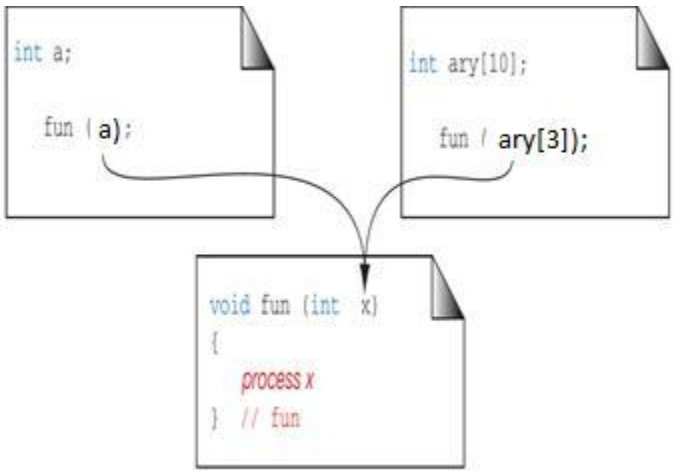
Passing Individual Elements:

One-dimensional Arrays:

We can pass individual elements by either passing their data values or by passing their addresses. We pass data values i.e; individual array elements just like we pass any data value .As long as the array element type matches the function parameter type, it can be passed. The called function cannot tell whether the value it receives comes from an array, a variable or an expression.

Program using call by value

```
void func1( int a);

void main()

{

int a[5]={ 1,2,3,4,5};

func1(a[3]);

}

void func1( int x)

{

printf("%d",x+100);

}
```



Two-dimensional Arrays:

The individual elements of a 2-D array can be passed in the same way as the 1-D array. We can pass 2-D array elements either by value or by address.

Ex: Program using call by value

```
void fun1(int a)

main()

{

int a[2][2]={1,2,3,4};

fun1(a[0][1]);

}

void fun1(int x)

{

printf("%d",x+10);

}
```

14. How can we pass the Whole Array to Functions? Explain with example program. Ans: Passing an entire array to a function:

One-dimensional array:

To pass the whole array we simply use the array name as the actual parameter. In the called function, we declare that the corresponding formal parameter is an array. We do not need to specify the number of elements.

Program :

```
void fun1(int a[])

void main()

{

fun1(a);

}

void fun1( int x[])

{

int  i, sum=0;

for(i=0;i<5;i++)

sum=sum+a[i];

printf("%d",sum);

}
```

Two-dimensional array:

When we pass a 2-D array to a function, we use the array name as the actual parameter just as we did with 1-D arrays. The formal parameter in the called function header, however must indicate that the array has two dimensions.

Rules:

- The function must be called by passing only the array name.
- In the function definition, the formal parameter is a 2-D array with the size of the second dimension specified.

Program:

```
void fun1(int a[][2])

void main()

{
```

```
int a[2][2]={1,2,3,4};
fun1(a);
}
void fun1(int x[][2])
{
int  i,j;
for(i=0;i<2;i++)
for(j=0;j<2;j++)
printf("%d",x[i][j]);
}
```

15. What are different types of storage classes in „C"? (or)

Explain briefly auto and static storage classes with examples?(or)

Explain extern and register storage classes with example programs.

Ans: Storage classes in „C"

Variables in C differ in behavior. The behavior depends on the storage class a variable may assume. From C compiler"s point of view, a variable name identifies some physical location within the computer where the string of bits representing the variable"s value is stored. There are four storage classes in C:

(a) Automatic storage class
(b) Register storage class
(c) Static storage class
(d) External storage class

(a) Automatic Storage Class:-

The features of a variable defined to have an automatic storage class are as under:

Keyword	Auto
Storage	Memory.

Default initial value	An unpredictable value, which is often called a garbage value.
Scope	Local to the block in which the variable is defined.
Life	Till the control remains within the block in which the variable is defined

Following program shows how an automatic storage class variable is declared, and the fact that if the variable is not initialized it contains a garbage value.

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

When you run this program you may get different values, since garbage values are unpredictable. So always make it a point that you initialize the automatic variables properly, otherwise you are likely to get unexpected results. Scope and life of an automatic variable is illustrated in the following program.

```
main( )
{
auto int i = 1 ;
{
auto int i = 2 ;
{
auto int i = 3 ;
printf ( "\n%d ", i ) ;
}
printf ( "%d ", i ) ;
}
printf ( "%d", i ) ;
```

}

Static Storage Class:-

The features of a variable defined to have a **static** storage class are as under:

Keyword	Static
Storage	Memory.
Default initial value	Zero.
Scope	Local to the block in which the variable is defined.
Life	Value of the variable persists between different function calls

The following program demonstrates the details of static storage class:

```
main( )
{
    increment( ) ;
    increment( ) ;
    increment( ) ;
}

increment( )
{
    auto int i = 1 ;
    printf ( "%d\n", i ) ;
    i = i + 1 ;
}
```

```
main( )
{
    increment( ) ;
    increment( ) ;
    increment( ) ;
}

increment( )
{
    static int i = 1 ;
    printf ( "%d\n", i ) ;
    i = i + 1 ;
}
```

The output of the above programs would be:

```
1
1
1
```

```
1
2
3
```

Extern Storage Class:-

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

Keyword	Extern
Storage	Memory
default initial value	Zero
Scope	Global
Life	As long as the program execution does not come to end

External variables differ from those we have already discussed in that their scope is global, not local. External variables are declared outside all functions, yet are available to all functions that care to use them. Here is an example to illustrate this fact.

Ex:

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

Register Storage Class:-

The features of a variable defined to be of **register** storage class are as under:

Keyword	Register
Storage	CPU Registers
default initial value	An unpredictable value, which is often called a garbage value.
Scope	Local to the block in which the variable is defined.

Life	Till the control remains within the block in which the variable is defined.

A value 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. A good example of frequently used variables is loop counters. We can name their storage class as register.

```
main( )

{

register int i ;

for ( i = 1 ; i <= 10 ; i++ )

printf ( "\n%d", i ) ;

}
```

16. Enumerate the scope rules in C

. Ans:

Scope: Scope of a variable is the part of program in which it can be used

Scope rules

The rules are as under:

- Use **static** storage class only if you want the value of a variable to persist between different function calls.
- Use **register** storage class for only those variables that are being used very often in a program. Reason is, there are very few CPU registers at our disposal and many of them might be busy doing something else. Make careful utilization of the scarce resources. A typical application of **register** storage class is loop counters, which get used a number of times in a program.
- Use **extern** storage class for only those variables that are being used by almost all the functions in the program. This would avoid unnecessary passing of these variables as arguments when making a function call. Declaring all the variables as **extern** would amount to a lot of wastage of memory space because these variables would remain active throughout the life of the program.

- If we don‟t have any of the express needs mentioned above, then use the **auto** storage class. In fact most of the times we end up using the **auto** variables, because often it so happens that once we have used the variables in a function we don‟t mind loosing them.

17. What is pointer in c programming? What are its benefits?

Ans:

Pointer is a user defined data type that creates special types of variables which can hold the address of primitive data type like char, int, float, double or user defined data type like function, pointer etc. or derived data type like array, structure, union, enum.

Examples:

int *ptr;

In c programming every variable keeps two types of value.

1. Value of variable.

2. Address of variable where it has stored in the memory.

(1) Meaning of following simple pointer declaration and definition:

```
int a=5;
int* ptr;
ptr=&a;
```

Explanation:

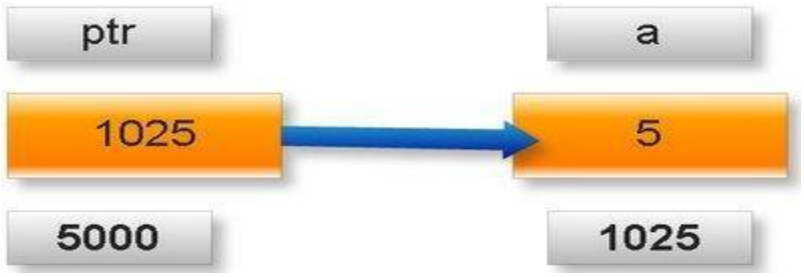
About variable ―a‖ :

1. Name of variable: a
2. Value of variable which it keeps: 5
3. Address where it has stored in memory: 1025 (assume)

About variable ―ptr‖ :

4. Name of variable: ptr
5. Value of variable which it keeps: 1025
6. Address where it has stored in memory: 5000 (assume)

Pictorial representation:



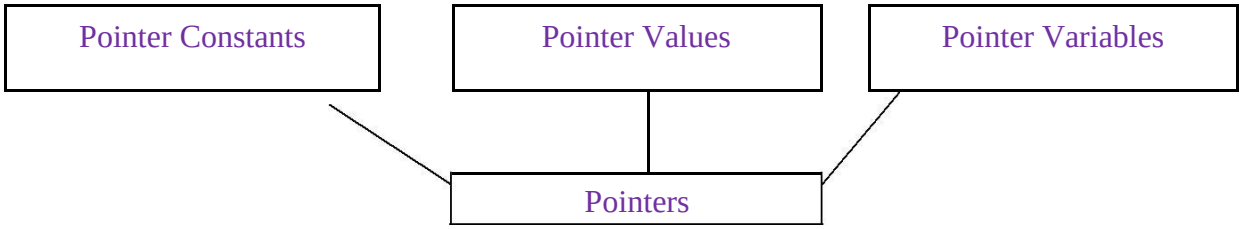
Note: A variable where it will be stored in memory is decided by operating system. We cannot guess at which location a particular variable will be stored in memory.

Pointers are built on three underlying concepts which are illustrated below:-

Memory addresses within a computer are referred to as ***pointer constants***. We cannot change them. We can only use them to store data values. They are like house numbers.

We cannot save the value of a memory address directly. We can only obtain the value through the variable stored there using the address operator (&). The value thus obtained is known as ***pointer value***. The pointer value (i.e. the address of a variable) may change from one run of the program to another.

Once we have a pointer value, it can be stored into another variable. The variable that contains a pointer value is called a ***pointer variable.***



Benefits of using pointers are:-

1) Pointers are more efficient in handling arrays and data tables.
2) Pointers can be used to return multiple values from a function via function arguments.
3) The use of pointer arrays to character strings results in saving of data storage space in memory.
4) Pointers allow C to support dynamic memory management.
5) Pointers provide an efficient tool for manipulating dynamic data structures such as structures , linked lists , queues , stacks and trees.
6) Pointers reduce length and complexity of programs.
7) They increase the execution speed and thus reduce the program execution time.

18. Explain the process of declaring and initializing pointers. Give an example

Ans:

In C, every variable must be declared for its type. Since pointer variable contain addresses that belong to a separate data type ,they must be declared as pointers before we use them.

a) Declaration of a pointer variable:

The declaration of a pointer variable takes the following form:

data_type *pt_name;

This tells the compiler three things about the variable pt_name:

1) The * tells that the variable pt_name is a pointer variable

2) pt_name needs a memory location

3) pt_name points to a variable of type data_type

Ex: int *p;

Declares the variable p as a pointer variable that points to an integer data type.

b) Initialization of pointer variables:

The process of assigning the address of a variable to a pointer variable is known as ***initialization.*** Once a pointer variable has been declared we can use assignment operator to initialize the variable.

Ex:

int quantity ;

int *p; //declaration

p=&quantity; //initialization

We can also combine the initialization with the declaration:

int *p=&quantity;

Always ensure that a pointer variable points to the corresponding type of data.

It is also possible to combine the declaration of data variable, the declaration of pointer variable and the initialization of the pointer variable in one step.

int x, *p=&x;

19. Explain in brief about & and * operators

Explain how to access the value of a variable using pointer

Explain how to access the address of a variable.

Ans: The actual location of a variable in the memory is system dependent and therefore the address of a variable is not known to us immediately. In order to determine the address of a variable we use & operator in c. This is also called as the **address operator.** The operator & immediately preceding a variable returns the address of the variable associated with it.

P=&x; would assign the address 5000 to the variable p.

The & operator can be remembered as **address of.**

Example program:

```
main( )
{
int a = 5 ;
printf ( "\nAddress of a = %u", &a );
printf ( "\nValue of a = %d", a );
}
```

Output: The output of the above program would be:

```
address of a=1444
value of a=5
```

The expression &a returns the address of the variable a, which in this case happens to be 1444 .Hence it is printed out using %u, which is a format specified for printing an unsigned integer.

Accessing a variable through its pointer:

Once a pointer has been assigned the address of a variable, to access the value of the variable using pointer we use the operator ***'*', called 'value at address' operator***. It gives the value stored at a particular address. The „value at address" operator is also called ***'indirection'*** **operator (or dereferencing operator)**.

Ex:

```
main()
{
int a = 5 ;
printf ( "\nAddress of a = %u", &a );
printf ( "\nValue of a = %d", a ) ;
printf ( "\nValue of a = %d", *( &a ) );

}
```

Output: The output of the above program would be:

```
Address of a = 1444
Value of a = 5
Value of a = 5
```

20. Explain how pointers are used as function arguments.

Ans: When we pass addresses to a function, the parameter receiving the addresses should be pointers. The process of calling a function using pointers to pass the address of variables is known as "call by reference". The function which is called by reference can change the value of the variable used in the call.

Example :-

```
void main()
{
int x;
x=50;
change(&x);       /* call by reference or address */
printf("%d\n",x);
}
change(int *p)
{
  *p=*p+10;
}
```

When the function change () is called, the address of the variable x, not its value, is passed into the function change (). Inside change (), the variable **p** is declared as a pointer and therefore **p** is the address of the variable x. The statement,

*p = *p + 10;

Means ―add 10 to the value stored at the address **p‖.** Since **p** represents the address of **x**, the value of x is changed from 20 to 30. Thus the output of the program will be 30, not 20.

Thus, call by reference provides a mechanism by which the function can change the stored values in the calling function.

21. Write a _C' function using pointers to exchange the values stored in two locations in the memory. (Or)

Write a C program for exchanging of two numbers using call by reference mechanism.

Ans:

Using pointers to exchange the values

- Pointers can be used to pass addresses of variables to called functions, thus allowing the called function to alter the values stored there.
- Passing only the copy of values to the called function is known as **"call by value".**

- Instead of passing the values of the variables to the called function, we pass their addresses, so that the called function can change the values stored in the calling routine. This is known as **"call by reference",** since we are referencing the variables.
- Here the addresses of actual arguments in the calling function are copied into formal arguments of the called function. Here the formal parameters should be declared as pointer variables to store the address.

program

```
#include<stdio.h>

void swap (int, int);

void main( )

{

int a=10, b=15;

swap(&a, &b);

printf("a=%d,b=%d", a,b);

}

void swap(int *x, int *y)

{

int temp;

temp = *x;

*x = *y;

*y = temp;

}
```

Output: **a = 15 b=10**

22. Explain how functions return pointer with an example program.

Ans:

Functions return multiple values using pointers. The return type of function can be a pointer of type int , float ,char etc.

Example :

```
#include<stdio.h>
int * smallest(int * , int *);
void main()
{
int a,b,*s;
printf("Enter a,b values ");
scanf("%d%d",&a,&b);
s=smallest(&a,&b);
printf("smallest no. is %d",*s);
}
int * smallest(int *a, int *b)
{

if(*a<*b)
return a;
else
return b;
}
```

In this example, "return a" implies the address of "a" is returned, not value .So in order to hold the address, the function return type should be pointer.

23. Explain about various arithmetic operations that can be performed on pointers.

Ans:

- Like normal variables, pointer variables can be used in expressions.
  Ex: x= (*p1) + (*p2);
- C language allows us to add integers to pointers and to subtract integers from pointers

  Ex: If p1, p2 are two pointer variables then operations such as p1+4, p2 - 2, p1 - p2 can be performed.
- Pointers can also be compared using relational operators.

Ex: p1>p2, p1==p2, p1! =p2 are valid operations.

- We should not use pointer constants in division or multiplication. Also, two pointers cannot be added. p1/p2, p1*p2, p1/3, p1+p2 are invalid operations.

- Pointer increments and scale factor:-

Let us assume the address of p1 is 1002. After using p1=p1+1, the value becomes 1004 but not 1003.

Thus when we increment a pointer, its values is increased by length of data type that points to. This length is called scale factor.

24. Explain in detail how to access a one dimensional array using pointers with an example program?

Ans: Pointers and Arrays :-

When an array is declared the complier allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations. The base address is the location of the first element (index 0) of the array .The complier also defines the array name as a constant pointer to the first element.

Ex:- static int x[5]= {1,2,3,4,5};

Suppose the base address of x is 1000 and assuming that each integer requires two bytes.
The five elements will be stored as follows.

elements	x[0]	x[1]	x[2]	x[3]	x[4]
value	1	2	3	4	5
address	1000	1002	1004	1006	1008

the name x is defined as a constant pointer pointing to the first element, x[0], and therefore the value x is 1000, the location where x[0] is stored. That is

x=&x[0]=1000;

If we declare p as an integer pointer, then we can make the pointer p to the array x by the following assignment

p=x;

which is equivalent to p=&x[0];

Now we can access every value of x using p++ to move from one element to another. The relationship between p and x is shown below

p=&x[0]=1000

p+1=&x[1]=1002

p+2=&x[2]=1004

p+3=&x[3]=1006

p+4=&x[4]=1008

Note:- address of an element in an array is calculated by its index and scale factor of the datatype

Address of x[n] = base address + (n*scale factor of type of x).

Eg:- int x[5]; x=1000;

Address of x[3]= base address of x+ (3*scale factor of int)

= 1000+(3*2)

= 1000+6

=1006

Exg:- float avg[20]; avg=2000;

Address of avg [6]=2000+(6*scale factor of

float) =2000+6*4 =2000+24

=2024.

Ex:- char str [20]; str =2050;

Address of str[10]=2050+(10*1)

=2050+10

=2060.

Note2:- when handling arrays, of using array indexing we can use pointers to access elements.

Like *(p+3) given the value of x[3]

The pointer accessing method is faster than the array indexing.

Accessing elements of an array:-

```
/*Program to access elements of a one dimensional array*/
#include<stdio.h>
#include<conio.h>
void main()
{
int arr[5]={10,20,30,40,50};
int p=0;
printf("\n value@ arr[i] is arr[p] | *(arr+p)| *(p+arr) | p[arr] | located @address
\n"); for(p=0;p<5;p++)
 {
printf("\n value of arr[%d] is:",p);
printf(" %d   | ",arr[p]);
printf(" %d   | ",*(arr+p));
printf(" %d   | ",*(p+arr));
printf(" %d   | ",p[arr]);
printf("address of arr[%d]=%u\n",p,arr[p]);
  }
  }
```

output:

```
value@ arr[i] is arr[p] | *(arr+p)| *(p+arr) | p[arr] | located @address
value of arr[0] is: 10    |  10    |  10    |  10    | address of arr[0]=10
value of arr[1] is: 20    |  20    |  20    |  20    | address of arr[1]=20
value of arr[2] is: 30    |  30    |  30    |  30    | address of arr[2]=30
value of arr[3] is: 40    |  40    |  40    |  40    | address of arr[3]=40
value of arr[4] is: 50    |  50    |  50    |  50    | address of arr[4]=50
-
```

25. Explain in detail how to access a two dimensional array using pointers with an example program?

Ans: Pointers to two-dimensional arrays:-

Pointer can also be used to manipulate two-dimensional arrays.

Just like how x[i] is represented by

*(x+i) or *(p+i), similar, any element in a 2-d array can be represented by the pointer as follows.

((a+i)+j) or *(*(p+i)+j)

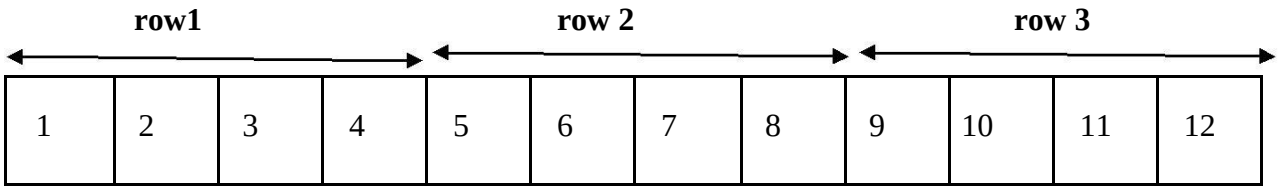
The base address of the array a is &a[0][0] and starting from this address the complier allocates contiguous space for all the element row – wise .

i,e the first element of the second row is placed immediately after the last element of the first row and so on

Ex:- suppose we declare an array as follows.

int a[3][4]={{1,2,3,,4},{5,6,7,8},{9,10,11,12}};

The elements of a will be stored as shown below.



Base address =&a[0][0]

If we declare p as int pointer with the intial address &a[0][0] then a[i][j] is equivalent to *(p+4 x i+j). You may notice if we increment „i‟ by 1, the p is incremented by 4, the size of each row. Then the element a[2][3] is given by *(p+2 x 4+3) =*(p+11).

This is the reason why when a two –dimensional array is declared we must specify the size of the each row so that the complier can determine the correct storage at mapping

Program:

```
/* accessing elements of 2 d array using pointer*/

#include<stdio.h>

void main()

{

int z[3][3]={{1,2,3},{4,5,6},{7,8,9}};

int i, j;

int *p;

p=&z[0][0];

for(i=0;i<3;i++)

{

for(j=0;j<3;j++)

 {

printf("\n %d\ti=%d j=%d\t%d",*(p+i*3+j),i,j,*(*(z+i)+j));

   }

   }

for(i=0;i<9;i++)

printf("\n\t%d",*(p+i));

   }
```

output:

```
1       i=0 j=0         1
2       i=0 j=1         2
3       i=0 j=2         3
4       i=1 j=0         4
5       i=1 j=1         5
6       i=1 j=2         6
7       i=2 j=0         7
8       i=2 j=1         8
9       i=2 j=2         9
        1
        2
        3
        4
        5
        6
        7
        8
        9_
```

26. Explain in detail pointers to character strings with an example program?

Ans:

String is an array of characters terminated with a null character. We can also use pointer to access the individual characters in a string .this is illustrated as below

Note :- In `c` a constant character string always represents a pointer to that string and the following statement is valid

```
char *name;

name = "delhi";
```

these statements will declare name as a pointer to a character and assign to name the constant character string "Delhi"

This type of declarations is not valid for character string

Like:- char name [20];

name ="delhi" ;**//invalid**

Program:

/* pointers and characters strings*/

```
//length of string using pointers
#include<stdio.h>
#include<conio.h>
void main()
{
int length=0; char
*name,*cptr;
name="sandhya STGY";
cptr=name; //assigning one pointer to another
printf("\n ADDRESS OF POINTERS: name=%u\t
cptr=%u",name,cptr); puts("\n entered string is:");
puts(name);
cptr=name;
while(*cptr!='\0') //is true untill end of str is reached
 {
printf("\n\t%c is @ %u",*cptr,cptr);
cptr++; //when while loop ends cptr has the address of '\0' in it
 }
length=cptr-name; //end of str minus strat of string gives no of chars in
 between //i.e. length of str;
printf("\n  Length of string is:%d",length);
 }
```

Output:

```
ADDRESS OF POINTERS: name=170   cptr=170
  entered string is:
sandhya STGY

        s is @ 170
        a is @ 171
        n is @ 172
        d is @ 173
        h is @ 174
        y is @ 175
        a is @ 176
          is @ 177
        S is @ 178
        T is @ 179
        G is @ 180
        Y is @ 181
  Length of string is:12
```

Pointer to table of string:-

One important use of pointers is in handling of a table of strings.

Consider the following array string :

char name [3][25];

Here name containing 3 names, each with a maximum length of 25 characters and total storage requirement for the name table is 75 bytes

We know that rarely the individual string will be of equal length instead of making each row a fixed number of characters we can make it a pointer to a string of varying length.

Eg:- char *name[3]={“apple”

“banana”

“pine apple”};

This declaration allocates only 21 bytes sufficient to hold all the character

- To print all the 3 names use the following statement ,

for (i=0; i<=2; i++)

printf("%s/n", name [i]);

- To access the jth character in the ith name we may write ,

*(name [i]+j)

Note:- The character arrays with the rows of varying length are called "ragged arrays" and are better handled pointers

27. What is an array of pointer? How it can be declared? Explain with an example?

Ans: Array of Pointers(Pointer arrays):-

We have studied array of different primitive data types such as int, float, char etc. Similarly C supports array of pointers i.e. collection of addresses.

Example :-

```
void main()
{
int *ap[3];
int al[3]={10,20,30};
int k;
for(k=0;k<3;k++)
ap[k]=al+k;
printf("\n address element\n");
for(k=0;k<3;k++)
{
printf("\t %u",ap[k]);
printf("\t %7d\n",*(ap[k]));
}
}
```

Output:

Address	Element
4060	10
4062	20
4064	30

In the above program , the addresses of elements are stored in an array and thus it represents array of pointers.

A two-dimensional array can be represented using pointer to an array. But, a two-dimensional array can be expressed in terms of array of pointers also. The conventional array definition is,

data_type array name [exp1] [exp2];

Using array of pointers, a two-dimensional array can be defined as,

data_type *array_name [exp1];

Where,

- data_type refers to the data type of the array.
- array_name is the name of the array.
- exp1 is the maximum number of elements in the row.

Note that exp2 is not used while defining array of pointers. Consider a two-dimensional vector initialized with 3 rows and 4 columns as shown below,

int p[3][4]={{10,20,30,40},{50,60,70,80},{25,35,45,55}};

The elements of the matrix p are stored in memory row-wise (can be stored column-wise also) as shown in Fig.

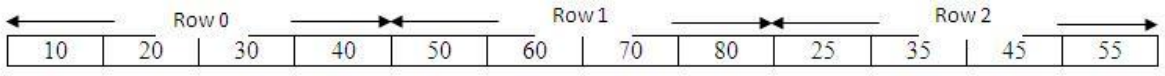


Fig: Two-Dimensional Array

Using array of pointers we can declare p as,

int *p[3];

Here, p is an array of pointers. p[0] gives the address of the first row, p[1] gives the address of the second row and p[2] gives the address of the third row. Now, p[0]+0 gives the address of the

element in 0^{th} row and 0^{th} column, p[0]+1 gives the address of the elements in 0^{th} row and 1^{st} column and so on. In general,

- Address of i^{th} row is given by a[i].
- Address of an item in i^{th} row and j^{th} column is given by, p[i]+j.
- The element in i^{th} row and j^{th} column can be accessed using the indirection operator * by specifying, *(p[i]+j)

28. Write in detail about pointers to functions? Explain with example program.

Ans: Pointers to functions:-

A function, like a variable has a type and address location in the memory. It is therefore possible to declare a pointer to a function, which can then be used as an argument in another function.

A pointer to a function can be declared as follows.

type (*fptr)();

This tells the complier that fptr is a pointer to a function which returns type value the parentheses around *fptr is necessary.

Because type *gptr(); would declare gptr as a function returning a pointer to type.

We can make a function pointer to point to a specific function by simply assigning the name of the function to the pointer.

Eg: double mul(int,int);

double (*p1)();

p1=mul();

It declares p1 as a pointer to a function and mul as a function and then make p1 to point to the function mul.

To call the function mul we now use the pointer p1 with the list of parameters.

i,e (*p1)(x,y); //function call equivalent to mul(x,y);

Program:

```
double mul(int ,int);
void main()
{
int x,y;
double (*p1)();
double res;
p1=mul;
printf("\n enter two numbers:");
scanf("%d %d",&x,&y);
res=(*p1)(x,y);
printf("\n The Product of X=%d and Y=%d is res=%lf",x,y,res);
 }
double mul(int a,int b)
 {
double val;
val=a*b;
return(val);
  }
```

Output:

```
using pointers to function
 enter two numbers:22 7

The Product of X=22 and Y=7 is res=154.000000_
```

29. What is a pointer to pointer? Write syntax and explain with example program.

Ans: It is possible to make a pointer to point to another pointer variable. But the pointer must be of a type that allows it to point to a pointer. A variable which contains the address of a pointer

variable is known as pointer to pointer. Its major application is in referring the elements of the two dimensional array.

Syntax for declaring pointer to pointer,

int *p; // declaration of a pointer

int **q; // declaration of pointer to pointer

This declaration tells compiler to allocate a memory for the variable „q‟ in which address of a pointer variable which points to value of type data type can be stored.

Syntax for initialization

q = & p;

This initialization tells the compiler that now „q‟ points to the address of a pointer variable „p‟.

Accessing the element value we specify, ** q;

Example:-

```
void main( )
{
int a=10;
int *p, **ptr;
 p = &a;
ptr = &p;
printf(“ \n the value of a =%d “, a);
printf(“ \n Through pointer p, the value of a =%d  “, *p);
printf(“ \n Through pointer to pointer q, the value of a =%d  “, **ptr);
}
```

Output:-

The value of a = 10;

Through pointer p, the value of a = 10;

Through pointer to pointer q, the value of a = 10;

30. Write a program for pointer to void?

Ans: We cann‟t assign a pointer of one type to other type. But, a pointer of any type can be assigned to pointer to void type and a pointer to void type can be assigned to a pointer of any referenced type. We can use Pointer to void to point either an integer or a real number.

```
void *p;
```

Program for Pointer to void:-

```
#include<stdio.h>

int main(void)

{

void *p;

int i=7;

float f=23.5;

p = &i;

printf(“ i contains %d \n”,*((int *)p));

p=&f;

printf(“f contains %f \n”,*((float *)p);

}
```

Output: i contains: 7

f contains:23.500000

31. Differentiate between static and dynamic memory allocation.

Explain about static and dynamic memory allocation techniques.

Ans: Memory can be reserved for the variables either during the compilation time or during execution time. Memory can be allocated for variables using two different techniques:

1. Static allocation

2. Dynamic allocation

1) S**tatic allocation**: If the memory is allocated during compilation time itself, the allocated memory space cannot be expanded to accommodate more data or cannot be reduced to accommodate less data.

In this technique once the size of the memory is allocated it is fixed. It cannot be altered even during execution time .This method of allocating memory during compilation time is called static memory allocation.

2) **Dynamic allocation**: Dynamic memory allocation is the process of allocating memory during execution time. This allocation technique uses predefined functions to allocate and release memory for data during execution time.

Static memory allocation	Dynamic memory allocation
It is the process of allocating memory at compile time.	It is the process of allocating memory during execution of program.
Fixed number of bytes will be allocated.	Memory is allocated as and when it is needed.
The memory is allocated in memory stack.	The memory is allocated from free memory pool (heap).
Execution is faster	Execution slow
Memory is allocated either in stack area or data area	Memory is allocated only in heap area
Ex: Arrays	Ex: Dynamic arrays, Linked List, Trees

32. Explain about malloc( ) and calloc( ) dynamic memory management functions with an example.

Ans: The function malloc( ) allocates a block of **size** bytes from the free memory pool (heap).

It allows a program to allocate an exact amount of memory explicitly, as and when needed.

```
Ptr=(cast_type *)malloc (byte_ size);
```

Ptr is a pointer of type **cast_type**.The malloc returns a pointer to an area of memory with size **byte_size**.The parameter passed to malloc() is of the type **byte_size**. This type is declared in the header file **alloc.h**. **byte_size** is equivalent to the unsigned int data type. Thus, in compilers

where an int is 16 bits in size, malloc() can allocate a maximum of 64KB at a time, since the maximum value of an unsigned int is 65535.

Return value:

- ✓ On success, i.e., if free memory is available, malloc() returns a pointer to the newly allocated memory. Usually, it is generic pointer. Hence, it should be typecast to appropriate data type before using it to access the memory allocate.
- ✓ On failure, i.e., if enough free memory does not exist for block, malloc() returns NULL. The constant NULL is defined in stdio.h to have a value zero. Hence, it is safe to check the return value.

Ex: 1) malloc(30); allocates 30 bytes of memory and returns the address of byte0.

2) malloc(sizeof(float)); allocates 4 bytes of memory and returns the address of byte0.

2) calloc( ) — allocates multiple blocks of memory

The function calloc( ) has the following prototype:

```
Ptr= (cast_type *)calloc(n,ele_ size);
```

calloc( ) provides access to the C memory heap, which is available for dynamic allocation of variable-sized blocks of memory.

Unlike malloc(), the function calloc( ) accepts two arguments: **n** and **ele_size**. The parameter **n** specifies the number of items to allocate and **ele_size** specifies the size of each item.

Return value:

- ✓ On success, i.e., if free memory is available, calloc( ) returns a pointer to the newly allocated memory. Usually, it is generic pointer. Hence, it should be typecast to appropriate data type before using it to access the memory allocated.
- ✓ On failure, i.e., if enough free memory does not exist for block, calloc( ) returns NULL. The constant NULL is defined in stdio.h to have a value zero. Hence, it is safe to verify the return value before using it.

Ex: 1) calloc(3,5); allocates 15 bytes of memory and returns the address of byte0.

2) malloc(6,sizeof(float)); allocates 24 bytes of memory and returns the address of byte0.

33. Explain about free( ) and realloc( ) allocation functions with an example?

Ans: i) realloc( ) — grows or shrinks allocated memory

The function realloc( ) has the following prototype:

```
Ptr= realloc(ptr, newsize);
```

The function realloc() allocates new memory space of size **newsize** to the pointer variable ptr and returns a pointer to the first byte of the new memoty block**.**

ptr is the pointer to the memory block that is previously obtained by calling malloc(), calloc() or realloc(). If **ptr** is NULL pointer, realloc() works just like malloc().

Return value:

- ✓ On success, this function returns the address of the reallocated block, which might be different from the address of the original block.
- ✓ On failure, i.e., if the block can‟t be reallocated or if the size passed is 0, the function returns NULL.

The function realloc() is more useful when the maximum size of allocated block cann‟t be decided in advance.

Ex: int *a;

a=(int *) malloc(30); //first 30 bytes of memory is allocated.

a=(int *) realloc(a,15); //later the allocated memory is shrink to 15 bytes.

ii) free( ) — de-allocates memory

The function free( ) has the following prototype:

```
free(ptr);
```

The function free( ) de-allocates a memory block pointed by **ptr.**

ptr is the pointer that is points to allocated memory by malloc( ), calloc( ) or realloc(). Passing an uninitialized pointer, or a pointer to a variable not allocated by malloc( ), calloc() or realloc() could be dangerous and disastrous.

Ex: int *a;

a=(int *) malloc(30); //first 30 bytes of memory is allocated.

free(a); //de-allocates 30 bytes of memory.