



# **SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.**

**18CSS101J –Programming for Problem Solving**  
**Unit III**



# SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

## Unit III: Content

**String Basics** – String Declaration and Initialization – String Functions: gets(), puts(), getchar(), putchar(), printf() - String Functions: atoi, strlen, strcat strcmp – String Functions: sprintf, sscanf, strrev, strcpy, strstr, strtok – Arithmetic characters on strings.



# SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

## STRING BASICS

- **Strings in C are represented by arrays of characters.**
- String is nothing but the collection of the individual array elements or characters stored at contiguous memory locations
- i) Character array – 'P','P','S'*
- ii) Double quotes - “PPS” is a example of String.*
  - If string contains the double quote as part of string then we can use escape character to keep double quote as a part of string.
  - “PP\S” is a example of String

### *iii) Null Character*

- The end of the string is marked with a special character, the *null character*, which is a character all of whose bits are zero i.e., a NULL.
- String always Terminated with NULL Character ('/0')

***char name[10] = {'P','P','S','\0'}***

- NULL Character is having ASCII value 0
- ASCII Value of '\0' = 0
- As String is nothing but an array , so it is Possible to Access Individual Character

***name[10] = "PPS";***

- It is possible to access individual character

***name[0] = 'P';***

***name[1] = 'P';***

***name[2] = 'S';***

***name[3] = '\0';***

#### *iv) MEMORY:*

Each Character Occupy 1 byte of Memory

Size of "PPS" = Size of 'P' +  
                  = Size of 'P' +  
                  = Size of 'S' ;  
Size of "PPS" is 3 BYTES

Each Character is stored in consecutive memory location.

Address of 'P' = 2000

Address of 'P' = 2001

Address of 'S' =  
2002

## ***STRING DECLARATION AND INITIALIZATION***

- ***String Declaration:***

- String data type is not supported in C Programming. String means Collection of Characters to form particular word. String is useful whenever we accept name of the person, Address of the person, some descriptive information. We cannot declare string using String Data Type, instead of we use array of type character to create String.
- Character Array is Called as ‘String’.
- Character Array is Declared Before Using it in Program.

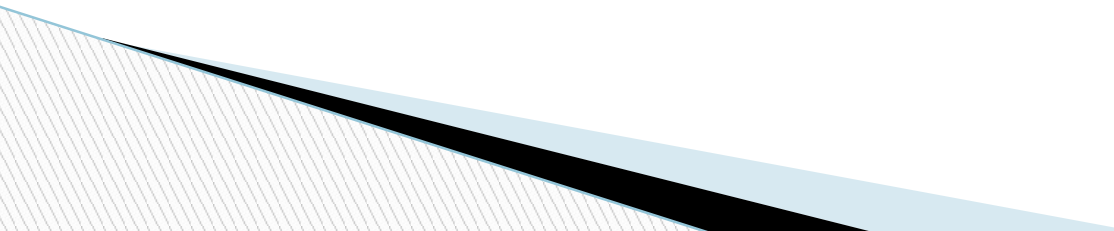
***char String\_Variable\_name [ SIZE ] ;***

***Eg: char city[30];***

## Point

## Explanation

Significance	- We have declared array of character[i.e String]
Size of string	- 30 Bytes
Bound checking	- C Does not Support Bound Checking i.e if we store City with size greater than 30 then C will not give you any error
Data type	- char
Maximum size	- 30



## *Precautions to be taken while declaring Character Variable :*

- String / Character Array Variable name should be legal C Identifier.
- String Variable must have Size specified.
  - **char city[];**
- Above Statement will cause compile time error.
  - Do not use String as data type because String data type is included in later languages such as C++ / Java. C does not support String data type
  - **String city;**
- When you are using string for other purpose than accepting and printing data then you must include following header file in your code –
  - **#include<string.h>**



## 2. *Initializing String [Character Array] :*

- Whenever we declare a String then it will contain garbage values inside it. We have to initialize String or Character array before using it. Process of Assigning some legal default data to String is Called Initialization of String. There are different ways of initializing String in C Programming –
  - ❑ Initializing Unsized Array of character
  - ❑ Initializing String Directly
  - ❑ Initializing String Using Character Pointer

## *Way 1 : Unsize Array and Character*

- ***Unsize Array***: Array Length is not specified while initializing character array using this approach
- Array length is Automatically calculated by Compiler
- Individual Characters are written inside Single Quotes , Separated by comma to form a list of characters. Complete list is wrapped inside Pair of Curly braces
- ***NULL Character*** should be written in the list because it is ending or terminating character in the String/Character Array
- `char name [] = {'P','P','S','\0'};`

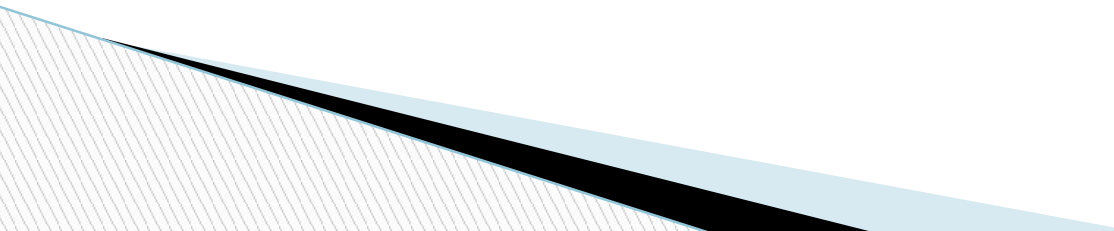
## *Way 2 : Directly initialize String Variable*

- In this method we are directly assigning String to variable by writing text in double quotes.
- In this type of initialization , we don't need to put NULL or Ending / Terminating character at the end of string. It is appended automatically by the compiler.
- `char name [ ] = "PPS";`

### *Way 3 : Character Pointer Variable*

- Declare Character variable of pointer type so that it can hold the base address of “String”
- Base address means address of first array element i.e (address of name[0] )
- NULL Character is appended Automatically
- `char *name = "PPS";`

## ***STRING FUNCTIONS***

- puts( )
  - gets( )
  - Getchar( )
  - Putchar( )
  - Printf( )
  - Atoi( )
  - Strlen( )
  - Strcat ( )
  - Strcmp( )
  - Sprintf( )
  - Sscanf( )
  - Strrev( )
  - Strcpy( )
  - Strstr()
  - strtok()
- 

## ***GETS():***

Syntax for Accepting String :

**char \* gets ( char \* str );    OR   gets( <variable-name> )**

**Example :** #include<stdio.h> void

main()

{

char name[20]; printf("\nEnter the Name : ");

gets(name); }

**Output:**

**Enter the name: programming in c**





## SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

- ❑ Whenever gets() statement encounters then characters entered by user (the string with spaces) will be copied into the variable.
- ❑ If user start accepting characters , and if new line character appears then the newline character will not be copied into the string variable(i.e name).
- ❑ A terminating null character is automatically appended after the characters copied to string variable (i.e name)
- ❑ gets() uses stdin (Standard Input Output) as source, but it does not include the ending newline character in the resulting string and does not allow to specify a maximum size for string variable (which can lead to buffer overflows).



# SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

## *Some Rules and Facts*

### **A. %s is not Required :**

Like scanf statement %s is not necessary while accepting string. scanf("%s",name);  
and here is gets() syntax which is simpler than scanf() –

gets(name);

### **B. Spaces are allowed in gets()**

: gets(name);

Whenever the above line encounters then interrupt will wait for user to enter some text on the screen. When user starts typing the characters then all characters will be copied to string and when user enters newline character then process of accepting string will be stopped.

### **Sample Input Accepted by Above Statement :**

- Value Accepted : Problem solving\n
- Value Stored : Problem solving(\n Neglected)



## 2. *PUTS()*:

### Way 1 :Messaging

- `puts(" Type your Message / Instruction ");`
- Like Printf Statement `puts()` can be used to display message.

### Way 2 : Display String

- `puts(string_Variable_name) ;`

### Notes or Facts :

- `puts` is included in header file “`stdio.h`”
- As name suggest it used for Printing or Displaying Messages or Instructions.

### Example :

```
#include< stdio.h>
#include< conio.h>
void main()
{
    char string[] = "This is an example string\n";
    puts(string);    // String is variable Here
    puts("String");  // String is in Double Quotes
    getch();
}
```

### Output :

***String is : This is an example  
string String is : String***





# SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

## GETCHAR():

- **Getchar()** function is also one of the function which is used to accept the single character from the user.
- The characters accepted by `getchar()` are buffered until **RETURN** is hit means `getchar()` does not see the characters until the user presses return. (i.e Enter Key)

## Syntax for Accepting String and Working :

```
/* getchar accepts character & stores in ch */  
char ch = getchar();
```

- When control is on above line then `getchar()` function will accept the single character. After accepting character control remains on the same line. When user presses the enter key then `getchar()` function will read the character and that character is assigned to the variable 'ch'.

### **Parameter Explanation**

Header File	-	stdio.h
Return Type	-	int (ASCII Value of the character)
Parameter	-	Void
Use	-	Accepting the Character

## Example 1 :

In the following example we are just accepting the single character and printing it on the console –

```
main()
```

```
{
```

```
    char ch;
```

```
    ch = getchar();
```

```
    printf("Accepted Character : %c",ch);
```

```
}
```

Output :

Accepted Character : A





# SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

## Example 2 : Accepting String (Use One of the Loop)

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

```
void main()
```

```
{
```

```
int i = 0;
```

```
char name[20];
```

```
printf("\nEnter the Name : ");
```

```
while((name[i] = getchar())!='\n')
```

*i++ ; /\*while loop will accept the one character at a time and check it with newline character. Whenever user enters newline character then control comes out of the loop.\*/*

```
getch(); }
```

## PUTCHAR():

### Displaying String in C Programming

Syntax :

```
int putchar(int c);
```

Way 1 : Taking Character as

Parameter `putchar('a');` // Displays :

a

- Individual Character is Given as parameter to this function.
- We have to explicitly mention Character.

## Way 2 : Taking Variable as Parameter

`putchar(a);`

// Display Character Stored in a

- ❑ Input Parameter is Variable of Type “Character”.
- ❑ This type of putchar() displays character stored in variable.

## Way3:Displaying Particular Character from Array

`putchar(a[0]) ;`

// Display a[0] th element from array

- ❑ Character Array or String consists of collection of characters.
- ❑ Like accessing individual array element , characters can be displayed one by one using putchar().



### Example:

```
#include< stdio.h>
#include< conio.h>
int main()
{
    char string[] = "C
programming\n"; int i=0;
    while(string[i]!='\0')
    {
        putchar(string[i]);
        i++;
    }
    return 0;
}
```

Output:

C programming



# SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

## **PRINTF():**

Syntax :

Way 1 : Messaging

```
printf ( " Type your Message / Instruction " ) ;
```

Way 2 : Display String

```
printf ( "Name of Person is %s ", name ) ;
```

**Notes or Facts :**

printf is included in header file “**stdio.h**”

As name suggest it used for **Printing or Displaying Messages or**

**Instructions Uses :**

- Printing Message
- Ask user for entering the data ( Labels . Instructions )
- Printing Results



# SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

## **atoi FUNCTION:**

- Atoi = A to I = Alphabet to Integer
- **Convert String of number into Integer**

**Syntax: num = atoi(String);**

num - Integer Variable

String - String of Numbers

Example :

```
char a[10] = "100";  
int value = atoi(a);  
printf("Value = %d\n",  
value); return 0;
```

**Output :**

**Value : 100**

## Significance :

- Can Convert any String of Number into Integer Value that can Perform the arithmetic Operations like integer
- Header File : **stdlib.h**

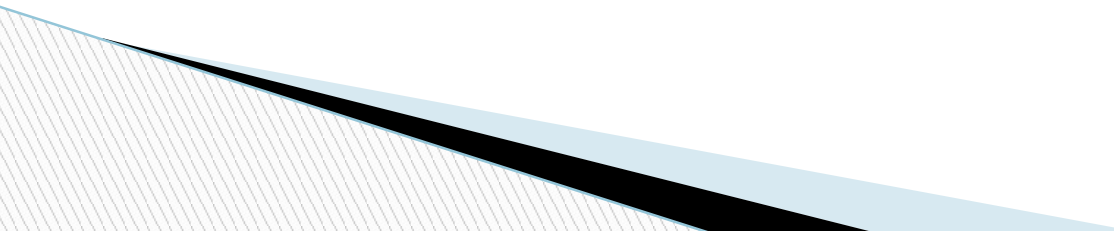
## Ways of Using Atoi Function :

### Way 1 : Passing Variable in Atoi Function

```
int num;  
char marks[3] = "98";  
num = atoi(marks);  
printf("\nMarks : %d",num);
```

### Way 2 : Passing Direct String in Atoi Function

```
int num;  
num = atoi("98");  
printf("\nMarks : %d",num);
```



## ***OTHER INBUILT TYPECAST FUNCTIONS IN C PROGRAMMING LANGUAGE:***

- ❑ Typecasting functions in C language performs data type conversion from one type to another.
- ❑ Click on each function name below for description and example programs.
  - `atof()` Converts string to float
  - `atoi()` Converts string to int
  - `atol()` Converts string to long
  - `itoa()` Converts int to string
  - `ltoa()` Converts long to string



# SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

## **STRLEN FUNCTION:**

- Finding length of string

### **Point Explanation**

No of	-	1
Parameters	-	Character Array Or String
Parameter Taken	-	Integer
Return Type	-	Compute the Length of the String
Description	-	string.h
Header file		

## *Different Ways of Using strlen() :*

- There are different ways of using strlen function. We can pass different parameters to strlen() function.

### *Way 1 : Taking String Variable as Parameter*

```
char str[20];  
int length ;  
printf("\nEnter the String : ");  
gets(str);  
length = strlen(str);  
printf("\nLength of String : %d ",  
length);
```

**Output:**

**Enter the String : hello**

**Length of String : 5**

Way 2 : Taking String Variable which is Already Initialized using

Pointer `char *str = "priteshtaral";`

`int length ;`

`length = strlen(str);`

`printf("\nLength of String : %d ", length);`

Way 3 : Taking Direct String

`int length ;`

`length = strlen("pritesh");`

`printf("\nLength of String :  
%d",length);`

Way 4 : Writing Function in printf

Statement `char *str = "pritesh";`

`printf("\nLength of String : %d", strlen(str));`



## ***STRCAT FUNCTION:***

### **What strcat Actually does ?**

- Function takes 2 Strings / Character Array as Parameter
- Appends second string at the end of First String. Parameter Taken - 2 Character Arrays / Strings Return Type - Character Array / String
- **Syntax :**

**`char* strlen ( char * s1, char * s2);`**



## ***Ways of Using Strcat Function :***

### **Way 1 : Taking String Variable as Parameter**

```
char str1[20] = "Don" , str2[20] =  
"Bosqo"; strcat(str1,str2);  
puts(str1);
```

### **Way 2 : Taking String Variable which is Already Initialized using Pointer**

```
char *str1 = "Ind",*str2 = "ia";  
strcat(str1,str2);// Result stored in str1 puts(str1); // Result : India
```

### **Way 3 : Writing Function in printf**

```
Statement printf("nString: ",  
strcat("Ind","ia"));
```

## **STRCMP FUNCTION:**

What strcmp Actually Does ?

- Function takes two Strings as parameter.
- It returns integer .

Syntax :

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

### **Return Type**

-ve Value

+ve Value

0 Value

-

-

-

### **Condition**

String1 < String2

String1 > String2

String1 = String2

## Example 1 : Two strings are Equal

```
char s1[10] = "SAM",s2[10]="SAM" ;  
int len;  
len = strcmp (s1,s2);
```

Output

0

*/\*So the output will be 0. if u want to print the string then give condition like\*/*

```
char s1[10] = "SAM",s2[10]="SAM" ;  
int len;  
len = strcmp (s1,s2);  
if (len == 0)  
printf ("Two Strings are Equal");
```

Output:

Two Strings are Equal

## Example 2 : String1 is Greater than String2

```
char s1[10] = "SAM",s2[10]="sam" ;  
int len;  
len = strcmp (s1,s2);  
printf ("%d",len); //-ve value
```

Output:

-32

Reason:

ASCII value of "SAM" is smaller than  
"sam"

ASCII value of 'S' is smaller than 's'



### Example 3 : String1 is Smaller than String1

```
char s1[10] = "sam",s2[10]="SAM" ;  
int len;  
len = strcmp (s1,s2);  
printf ("%d",len); //+ve value
```

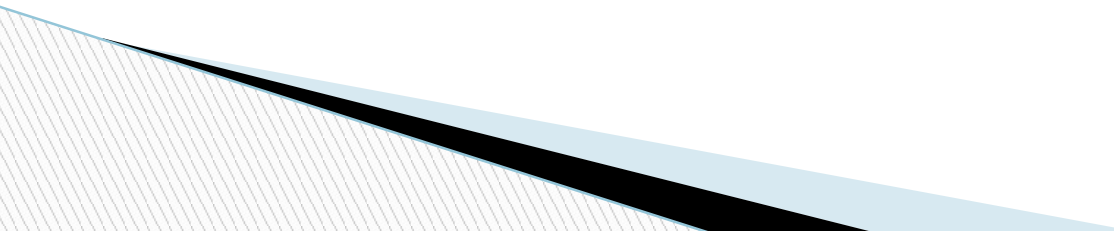
Output:

85

Reason:

ASCII value of “SAM” is greater than  
“sam”

ASCII value of ‘S’ is greater than ‘s’



## ***SPRINTF FUNCTION:***

- **sends formatted output to String.**

### **Features :**

- Output is Written into String instead of Displaying it on the Output Devices.
- Return value is integer ( i.e Number of characters actually placed in array / length of string ).
- String is terminated by '\0'.
- Main Purpose : Sending Formatted output to String.
- Header File : **Stdio.h**

### **Syntax :**

**int sprintf(char \*buf,char format,arg\_list);**



### Example :

```
int age = 23 ;  
char str[100];  
sprintf( str , "My age is  
%d",age); puts(str);
```

### Output:

My age is 23

### Analysis of Source Code: Just keep in mind that

- ❑ Assume that we are using printf then we get output “My age is 23”
- ❑ What does printf does ? — Just Print the Result on the Screen
- ❑ Similarly Sprintf stores result “My age is 23” into string str instead of printing it.





# SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

## ***SSCANF FUNCTION:***

**Syntax :**

**int sscanf(const char \*buffer, const char \*format[, address, ...]);**

What it actually does ?

- Data is read from array Pointed to by buffer rather than stdin.
- Return Type is Integer
- Return value is nothing but number of fields that were actually assigned a value

## Example

```
#include <stdio.h>
int main ()
{
    char buffer[30]="Fresh2refresh 5 ";
    char name [20];
    int age;
    sscanf (buffer,"%s %d",name,&age);
    printf ("Name : %s \n Age : %d
\n",name,age); return 0;
}
```

## Output:

Name : Fresh2refresh Age : 5



### 3. ***STRSTR FUNCTION:***

- Finds first occurrence of sub-string in other string

#### Features :

- Finds the first occurrence of a sub string in another string
- Main Purpose : **Finding Substring**
- Header File : **String.h**
- Checks whether s2 is present in s1 or not
- On success, strstr returns a pointer to the element in s1 where s2 begins (points to s2 in s1).
- On error (if s2 does not occur in s1), strstr returns null.

## Syntax :

**char \*strstr(const char \*s1, const char \*s2);**

```
#include <stdio.h>
#include <string.h>
int main ()
{
    char string[55] = "This is a test
    string; char *p;
    p = strstr (string, "test");
    if(p)
    {
        printf("string found\n");
    }
```

```
printf("First string \"test\" in \"%s\" to  \"%\" \"%s \"" ,string,  
p);  
}  
else  
    printf("string not found\n" );  
return 0;  
}
```

**Output:**

string found

First string “test” in “This is a test string” to “test string”.



## Example 2

### Parameters as String initialized using Pointers

```
#include<stdio.h>
#include<string.h>
int main(void)
{
    char *str1 = "c4learn.blogspot.com", *str2 = "spot",
    *ptr; ptr = strstr(str1, str2);
    printf("The substring is: %sn", ptr);
    return 0;
}
```

### Output:

The substring is: spot.com



## Example 3

### Passing Direct Strings

```
#include<stdio.h>
#include<string.h>
void main()
{
    char *ptr;
    ptr =
    strstr("c4learn.blogspot.com","spot");
    printf("The substring is: %sn", ptr);
}
```

Output:

The substring is: spot.com



## **STRREV():**

- reverses a given string in C language. Syntax for strrev( ) function is given below.

**char \*strrev(char \*string);**

- strrev() function is nonstandard function which may not be available in standard library in C.

### **Algorithm to Reverse String in C :**

- Start
- Take 2 Subscript Variables 'i','j'
- 'j' is Positioned on Last Character
- 'i' is positioned on first character
- str[i] is interchanged with str[j]
- Increment 'i'
- Decrement 'j'
- If 'i' > 'j' then goto step 3
- Stop



## Example

```
#include<stdio.h>
#include<string.h>
int main()
{
    char name[30] = "Hello";
    printf("String before strrev() :%s\n",name);
    printf("String after strrev(%s",
    strrev(name)); return 0;
}
```

## Output:

String before strrev() : Hello

String after strrev() :

olleH

## ***STRCPY FUNCTION:***

**Copy second string into First**

What strcpy Actually Does ?

- ❑ Function takes two Strings as parameter.
- ❑ Header File : String.h
- ❑ It returns string.
- ❑ Purpose : Copies String2 into String1.
- ❑ Original contents of String1 will be lost.
- ❑ Original contents of String2 will remains as it is.

**Syntax :**

**char \* strcpy ( char \*string1, char \*string2 ) ;**



- strcpy ( str1, str2) – It copies contents of str2 into str1.
- strcpy ( str2, str1) – It copies contents of str1 into str2.
- If destination string length is **less than source string**, **entire source string value won't be copied into destination** string.
- For example, consider destination string length is 20 and source string length is 30. Then, only 20 characters from source string will be copied into destination string and remaining 10 characters won't be copied and will be truncated.

## Example 1

```
char s1[10] = "SAM" ;
```

```
char s2[10] = "MIKE" ;
```

```
strcpy (s1,s2);
```

```
puts (s1) ; // Prints : MIKE
```

```
puts (s2) ; // Prints : MIKE
```

Output:

MIKE MIKE



## Example 2

```
#include <stdio.h>
#include <string.h>
int main( )
{
    char source[ ] = "hihello" ;
    char target[20]= "" ;
    printf ( "\nsource string = %s", source ) ;
    printf ( "\ntarget string = %s", target ) ;
    strcpy ( target, source ) ;
    printf("target string after strcpy()=%s",target) ;
    return 0; }
```

## Output

source string = hihello

target string =

target string after strcpy( ) = hihello

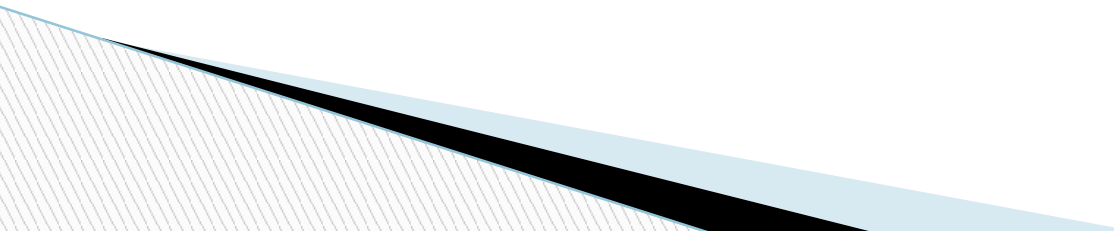
### 3.11.6 STRTOK FUNCTION

- tokenizes/parses the given string using delimiter.

#### Syntax

**char \* strtok ( char \* str, const char \* delimiters );**

*For example, we have a comma separated list of items from a file and we want individual items in an array.*

- *Splits str[] according to given delimiters and returns next token.*
  - *It needs to be called in a loop to get all tokens.*
  - *It returns NULL when there are no more tokens.*
- 

## Example

```
include <stdio.h>
#include <string.h>
int main()
{
char str[] = "Problem_Solving_in_c";//Returns first token
char* token = strtok(str, "_"); //Keep printing tokens while one of the delimiters present in str[].
    while (token != NULL) {
        printf("%s\n", token);
        token = strtok(NULL, "_");
    }
return 0;
}
```

**Output:**

**Problem Solving in C**



# ARITHMETIC CHARACTERS ON STRING

- C Programming Allows you to Manipulate on String
- Whenever the Character is variable is used in the expression then it is automatically Converted into Integer Value called ASCII value.
- All Characters can be Manipulated with that Integer Value.(Addition,Subtraction)

**Examples :**

**ASCII value of : 'a' is 97    ASCII value of :  
'z' is 121**





# Possible Ways of Manipulation :

Way 1:Displays ASCII value[ Note that %d in Printf]

```
char x = 'a';
```

```
printf("%d",x); // Display Result = 97
```

Way 2 :Displays Character value[Note that %c in Printf] char x = 'a';

```
printf("%c",x); // Display Result = a
```

Way 3 : Displays Next ASCII value[ Note that %d in Printf ] char x = 'a' + 1 ;

```
printf("%d",x); //Display Result = 98 (ascii of 'b' )
```



#### Way 4 Displays Next Character value[Note that %c in Printf ]

```
char x = 'a' + 1;
```

```
printf("%c",x); // Display Result = 'b'
```

#### Way 5 : Displays Difference between 2 ASCII in Integer[Note %d in Printf ]

```
char x = 'z' - 'a';
```

```
printf("%d",x);/*Display Result = 25 (difference between ASCII of z and a ) */
```

#### Way 6 : Displays Difference between 2 ASCII in Char [Note that %c in Printf ]

```
char x = 'z' - 'a';
```

```
printf("%c",x);/*Display Result =( difference between ASCII of z and a ) */
```





# SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

## FUNCTION DECLARATION AND DEFINITION:

- ❑ A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.
- ❑ You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.
- ❑ A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.
- ❑ The C standard library provides numerous built-in functions that your program can call. For example, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions.
- ❑ A function can also be referred as a method or a sub-routine or a procedure, etc.



# SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

## Defining a function

The general form of a function definition in C programming language is as follows –

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

A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function –

**Return Type** – A function may return a value. The **return\_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return\_type is the keyword **void**.



**Function Name** – This is the actual name of the function. The function name and the parameter list together constitute the function signature.

**Parameters** – A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

**Function Body** – The function body contains a collection of statements that define what the function does



```
main()
{
display();
}
void mumbai()
{
printf("In mumbai");
}
void pune()
{
india();
}
void display()
{
pune();
}
void india()
{
mumbai();
}
```

We have written functions in the above specified sequence , however functions are called in which order we call them.

Here functions are called this sequence -

main() □ display() □  
pune() □ india() □  
mumbai().



# Why Function is used???

## Advantages of Writing Function in C Programming

### 1. Modular and Structural Programming can be done

- ☐ We can divide c program in **smaller modules**.
- ☐ We can call module whenever require. e.g suppose we have written calculator program then we can write 4 modules (i.e add,sub,multiply,divide)
- ☐ Modular programming **makes C program more readable**.
- ☐ Modules once created , **can be re-used in other programs**.

### 2. It follows Top-Down Execution approach , So main can be kept very small.

- ☐ Every C program starts **from main function**.
- ☐ Every function is **called directly or indirectly through main**
- ☐ Example : **Top down approach**. (functions are executed from top to bottom)



### 3. Individual functions can be easily built, tested

- ☐ As we have developed C application in modules **we can test each and every module.**
- ☐ **Unit testing** is possible.
- ☐ Writing code in function will **enhance application development process.**

### 4. Program development become easy

### 5. Frequently used functions can be put together in the customized library

- ☐ We can put frequently used functions in **our custom header file.**
- ☐ After creating header file we can re use header file. We can include header file in other program.

### 6. A function can call other functions & also itself

- ☐ Function can call other function.
- ☐ Function can call itself, which is called as “recursive” function.
- ☐ Recursive functions are also useful in order to write system functions.

### 7. It is easier to understand the Program topic

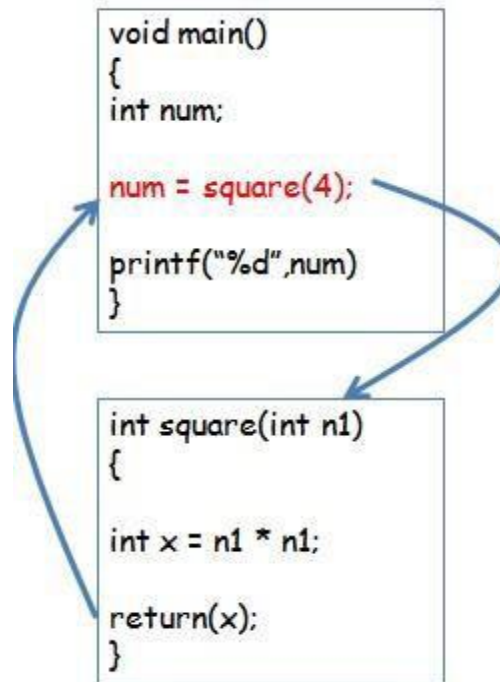
- ☐ We can get overall idea of the project just by reviewing function names.





## How Function works in C Programming?

- ❑ C programming is modular programming language.
- ❑ We must divide C program in the different modules in order to create more readable, eye catching ,effective, optimized code.
- ❑ In this article we are going to see how function is C programming works ?





## Explanation : How function works in C Programming ?

- ☐ Firstly Operating System will call our main function.
- ☐ When control comes inside main function , execution of main starts execution of C program starts)

Consider line 4 :

**num = square(4);**

- ☐ We have called a function square(4). [ See : How to call a function ? ].
- ☐ We have passed “4” as parameter to function.



*Note : Calling a function halts execution of the current function , it will execute called function*

- ☐ After execution control returned back to the calling function.
- ☐ Function will return 16 to the calling function.(i.e. main)
- ☐ Returned value will be copied into variable.
- ☐ printf will gets executed.
- ☐ main function ends.
- ☐ C program terminates.



## FUNCTION PROTOTYPE DECLARATION IN C PROGRAMMING

- ❑ Function prototype declaration is necessary in order to provide information the compiler about function, about return type, parameter list and function name etc.

### Important Points :

- ❑ Our program starts from main function. Each and every function is called directly or indirectly through main function
- ❑ Like variable we also need to declare function before using it in program.
- ❑ In C, declaration of function is called as prototype declaration
- ❑ Function declaration is also called as function prototype



## Points to remember

- ☐ Below are some of the important notable things related to prototype declaration
- ☐ It tells name of function, return type of function and argument list related information to the compiler
- ☐ Prototype declaration always ends with semicolon.
- ☐ Parameter list is optional.
- ☐ Default return type is integer.



## Header Files

Global Declaration Section  
(Write Prototype Declaration)



Write Main Function in this  
section

[www.c4learn.com](http://www.c4learn.com)

Write All Function Definitions  
in this Section



## Syntax

`return_type function_name ( type arg1, type arg2..... );`

prototype declaration comprised of three parts i.e name of the function, return type and parameter list

### Examples of prototype declaration

❑ Function with two integer arguments and integer as return type is represented

using syntax **`int sum(int,int);`**

❑ Function with integer argument and integer as return type is represented using

syntax **`int square(int);`**

❑ In the below example we have written function with no argument and no

return type **`void display(void);`**

❑ In below example we have declared function with no argument and integer

as return type **`int getValue(void);`**



## Positioning function declaration

- ☐ If function definition is written after main then and then only we write prototype declaration in global declaration section
- ☐ If function definition is written above the main function then ,no need to write prototype declaration

### Case 1 : Function definition written before main

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

```
void displayMessage()
```

```
{
```

```
printf("welcome");
```

```
}
```

```
void main()
```

```
{
```

```
displayMessage();
```

```
}
```





## Case 2 : Function definition written

**after main** #include<stdio.h>

//Prototype Declaration **void**

displayMessage();

**void** main()

{

displayMessage();

}

**void** displayMessage()

{

printf("welcome");

}

## Need of prototype declaration

- ☐ Program Execution always starts from main , but during lexical analysis (1st Phase of Compiler) token generation starts from left to right and from top to bottom.
- ☐ During code generation phase of compiler it may face issue of backward reference.



# SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

## TYPES OF CALLING

- ❑ While creating a C function, you give a definition of what the function has to do.

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

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



```
#include <stdio.h> /* function declaration */

int max(int num1, int num2);

int main ()
{
/* local variable definition */ int a = 100;

int b = 200;

int ret; /* calling a function to get max value */
    ret = max(a, b);
printf( "Max value is : %d\n", ret );
    return 0;
} /* function returning the max between two numbers */

int max(int num1, int num2)
{
/* local variable declaration */
result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
```

**output**  
**Max value is : 200**



*We have kept max() along with main() and compiled the source code. While running the final executable, it would produce the following result –*  
*Max value is : 200*

- ☐ If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.
- ☐ Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.
- ☐ While calling a function, there are two ways in which arguments can be passed to a function –
- ☐ By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

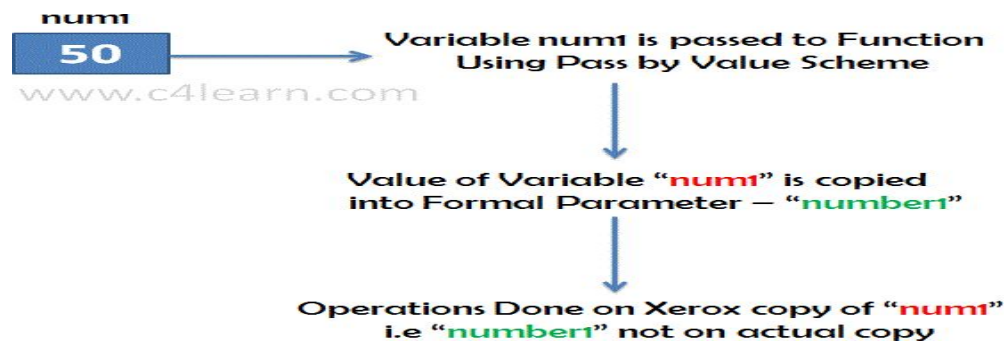


S.No.	Call Type & Description
1	<p><b><u>Call by value</u></b> This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.</p> <pre>/* function definition to swap the values */ void swap (int x, int y) {     int temp; temp = x;     /* save the value of x */     x = y; /* put y into x */     y = temp; /* put temp into y */     return; }</pre>
2	<p><b><u>Call by reference</u></b> This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.</p> <pre>/* function definition to swap the values */ void swap (int x, int y) {     int temp; temp     = *x;     *x = *y;     *y = temp;     return; }</pre>



## CALL BY VALUE

- ❑ While Passing Parameters using call by value , xerox copy of original parameter is created and passed to the called function.
- ❑ Any update made inside method will not affect the original value of variable in calling function.
- ❑ In the above example num1 and num2 are the original values and xerox copy of these values is passed to the function and these values are copied into number1, number2 variable of sum function respectively.
- ❑ As their scope is limited to only function so they cannot alter the values inside main function.





```
#include <stdio.h>
void swap(int x, int y); /* function
declaration */ int main ()
{
int a = 100; /* local variable definition */
int b = 200;
printf("Before swap, value of a : %d\n", a );
printf("Before swap, value of b : %d\n", b );/*calling a function to swap
the values */
swap(a, b);
printf("After swap, value of a : %d\n", a );
printf("After swap, value of b : %d\n", b );
return 0;
}
```

Output:

Before **swap**, value of a :100 Before swap, value of b :200

After swap, value of a :100 After swap, value of b :200

## 3.14.2 CALL BY REFERENCE

- ❑ The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.
- ❑ To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function `swap()`, which exchanges the values of the two integer their variables pointed to, by arguments





```
#include <stdio.h>
void swap(int *x, int *y); /*
function declaration */ int main ()
{
int a = 100; /* local variable definition */
int b = 200;
printf("Before swap, value of a : %d\n", a );
printf("Before swap, value of b : %d\n", b ); /*calling a function to swap
the values. * &a indicates pointer to a ie. address of variable a and * &b
indicates pointer to b ie. address of variable b.*/
swap(&a, &b);
printf("After swap, value of a : %d\n", a );
printf("After swap, value of b : %d\n", b );
return 0;
}
```

output

```
Before swap, value of a :100
Before swap, value of b :200
After swap,value of a :200
After swap,value of b :100
```



## FUNCTION WITH ARGUMENTS AND NO RETURN VALUE :

- ❑ Function accepts argument but it does not return a value back to the calling Program .
- ❑ It is Single ( One-way) Type Communication
- ❑ Generally Output is printed in the Called function

**Function declaration : void function();**

**Function call : function();**

**Function definition : void function() { statements; }**

```
#include <stdio.h>
void checkPrimeAndDisplay(int n);
int main()
{
    int n;
    printf("Enter a positive nteger: ");
    scanf("%d",&n);
    // n is passed to the function checkPrimeAndDisplay(n);
    return 0;
}
```



// void indicates that no value is returned from the function

```
void checkPrimeAndDisplay(int n)
```

```
{
```

```
int i, flag = 0;
```

```
for(i=2; i <= n/2; ++i)
```

```
{
```

```
if(n%i == 0){
```

```
flag = 1;
```

```
break; }
```

```
}
```

```
if(flag == 1)
```

```
printf("%d is not a prime
```

```
number.",n); else
```

```
printf("%d is a prime number.", n);
```

```
}
```

**OUTPUT**

Enter a positive integer : 4

4 is not a prime number



## FUNCTION WITH NO ARGUMENTS AND NO RETURN VALUE IN C

When a function has no arguments, it does not receive any data from the calling function. Similarly when it does not return a value, the calling function does not receive any data from the called function.

Syntax :

**Function declaration :** void function();

**Function call :** function();

**Function definition :** void function()  
{  
statements;  
}

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

```
void area(); // Prototype Declaration
```

```
void main()
```

```
{
```

```
area();
```

```
}
```



```
void area()  
{  
    float area_circle;  
    float rad;  
    printf("\nEnter the radius : ");  
    scanf("%f",&rad);  
    area_circle = 3.14 * rad * rad ;  
    printf("Area of Circle = %f",area_circle);  
}
```

Output :

Enter the radius : 3

Area of Circle = 28.260000



## FUNCTION WITHOUT ARGUMENTS AND RETURN VALUE

There could be occasions where we may need to design functions that may not take any arguments but returns a value to the calling function. A example for this is getchar function it has no parameters but it returns an integer an integer type data that represents a character.

**Function declaration :** int function();

**Function call :** function();

**Function definition :** int function() { statements; return x; }

```
#include<stdio.h>
int sum();
int main()
{
    int addition;
    addition = sum();
    printf("\nSum of two given values = %d", addition);
    return 0;
}
int sum()
{
    int a = 50, b = 80, sum; sum = a + b;
    return sum;
}
```

**OUTPUT**

Sum of two given values = 130



# FUNCTION WITH ARGUMENTS AND RETURN VALUE

**Syntax :**

**Function declaration :** `int function ( int );`

**Function call :** `function( x );`

**Function definition:** `int function( int x )  
{ statements; return x; }`

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

```
float calculate_area(int);
```

```
int main()
```

```
{
```

```
int radius;
```

```
float area;
```

```
printf("\nEnter the radius of the circle : ");
```

```
scanf("%d",&radius);
```

```
area = calculate_area(radius);
```

```
printf("\nArea of Circle : %f,area);
```

```
return(0);
```

```
}
```

```
float calculate_area(int radius)
```

```
{
```

```
float areaOfCircle;
```

```
areaOfCircle = 3.14 * radius * radius;
```

```
return(areaOfCircle);
```

```
}
```

**Output:**

**Enter the radius of the circle : 2**

**Area of Circle : 12.56**



# PASSING ARRAY TO FUNCTION IN C

## Array Definition :

Array is collection of elements of similar data types .

## Passing array to function :

Array can be passed to function by two ways :

- ☐ Pass Entire array
- ☐ Pass Array element by element

### 1 . Pass Entire array

- ☐ Here entire array can be passed as a argument to function .
- ☐ Function gets **complete access** to the original array .
- ☐ While passing entire array Address of first element is passed to function , any changes made inside function , directly **affects the Original value\_.**
- ☐ Function Passing method : “**Pass by Address**“





## 2 . Pass Array element by element:

- ☐ Here individual elements are passed to function as argument.
- ☐ Duplicate **carbon copy of Original variable** is passed to function .
- ☐ So any changes made inside function **does not affects the original value.**
- ☐ Function doesn't get complete access to the original array element.
- ☐ Function passing method is **“Pass by Value”**.

## 3. Passing entire array to function :

- ☐ Parameter Passing Scheme : **Pass by Reference**
- ☐ Pass **name of array** as function parameter .
- ☐ Name contains the base address i.e ( Address of 0th element )
- ☐ Array values are updated in function .
- ☐ Values are reflected inside main function also.



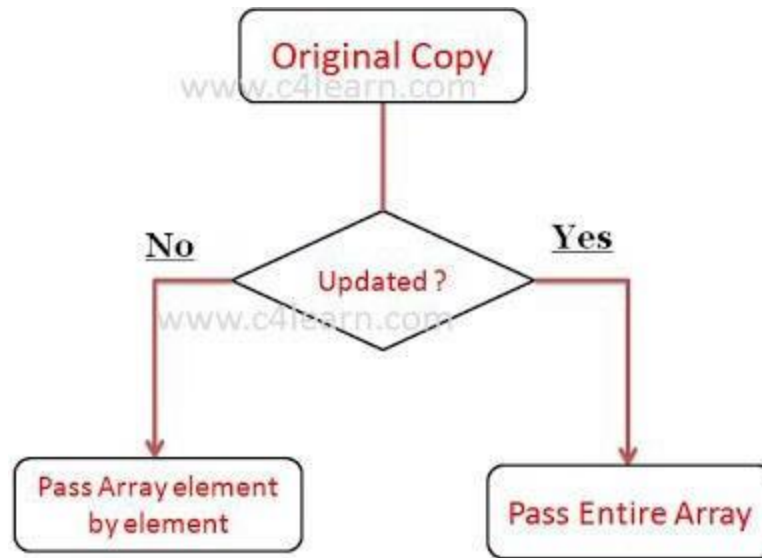
```
#include<stdio.h>
#include<conio.h>
void fun(int arr[])
{
    int i;
    for(i=0;i< 5;i++)
        arr[i] = arr[i] + 10;
}
void main()
{
    int arr[5],i;
    clrscr();
    printf("\nEnter the array elements : ");
```

```
    for(i=0;i< 5;i++)
        scanf("%d",&arr[i]);

    printf("\nPassing
    entire array .....");

    fun(arr); // Pass only name of array
    for(i=0;i< 5;i++)
        printf("\nAfter Function call a[%d]
        : %d",i,arr[i]); getch();
}
```

output  
Enter the array elements : 1 2 3 4 5  
Passing entire array .....  
After Function call a[0] : 11  
After Function call a[1] : 12  
After Function call a[2] : 13  
After Function call a[3] : 14  
After Function call a[4] : 15



## Passing Entire 1-D Array to Function in C Programming

- ☐ Array is passed to function Completely.
- ☐ Parameter Passing Method : **Pass by Reference**
- ☐ It is Also Called "**Pass by Address**"
- ☐ Original Copy is Passed to Function
- ☐ Function Body Can Modify **Original Value**.



### Example :

```
#include<stdio.h>
#include<conio.h>
void modify(int b[3]);
void main()
{
int arr[3] = {1,2,3};
modify(arr);
for(i=0;i<3;i++)
printf("%d",arr[i]);
getch();
}
void modify(int a[3])
{
int i; for(i=0;i<3;i++)
a[i] = a[i]*a[i];
}
```

### Output :

1 4 9

Here “**arr**” is same as “**a**” because Base Address of Array “**arr**” is stored in Array “**a**”

### Alternate Way of Writing Function Header :

void modify(int a[3]) **OR** void modify(int \*a)



### Passing array element by element to function :

- ❑ Individual element is passed to function using **Pass By Value** parameter passing scheme
- ❑ Original Array elements remains same as Actual Element is never Passed to Function. thus function body cannot modify **Original Value**.
- ❑ Suppose we have declared an array 'arr[5]' then its individual elements are arr[0],arr[1]...arr[4]. Thus we need 5 function calls to pass complete array to a function.

### Tabular Explanation :

Consider following array

Iteration	Element Passed to Function	Value of Element
1	arr[0]	11
2	arr[1]	22
3	arr[2]	33
4	arr[3]	44
5	arr[4]	55



## C Program to Pass Array to Function Element by Element :

```
#include<stdio.h>
#include< conio.h>
void fun(int num)
{
printf("\nElement : %d",num);
}
void main()
{
int arr[5],i;
clrscr();
printf("\nEnter the array elements : ");
for(i=0;i< 5;i++)
scanf("%d",&arr[i]);
printf("\nPassing array element by element.....");
for(i=0;i< 5;i++)
fun(arr[i]);
getch();
}
```

### Output :

```
Enter the array elements : 1 2 3 4 5 Passing array
element by element..... Element : 1
Element : 2
Element : 3
Element : 4
Element : 5
```



## DISADVANTAGE OF THIS SCHEME :

- ☐ This type of scheme in which we are calling the function again and again but with **different array element is too much time consuming**. In this scheme we need to call function by pushing the current status into the system stack.
- ☐ It is better to pass complete array to the function so that we can save some system time required for pushing and popping.



# Function Pointers

- In C, like normal data pointers (int \*, char \*, etc), we can have pointers to functions.

- Initialization

return\_type

function\_pointer(argu)=&function\_name **void**

**(\*fun\_ptr)(int) = &fun;**

- Function Definition

```
void fun(int a)
```

```
{
```

```
    printf("Value of a is %d\n", a);
```

```
}
```



```
#include<stdio.h>
void fun(int a)
{
    printf("a=%d\n",a);
}
void main()
{
    void (*fun1)(int)=&fun;

    (*fun1)(15);
}
```

**Output:**

a=15

If we remove bracket, then the expression

“void (\*fun\_ptr)(int)”

becomes

“void \*fun\_ptr(int)”

which is declaration of a function that returns void pointer.

### Interesting facts

- Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.
- Unlike normal pointers, we do not allocate de-allocate memory using function pointers.
- A function's name can also be used to get functions' address.

# Passing Function Pointer as an argument

```
#include<stdio.h>
void fun1(void(*test)())
{
int a=20;
test(a);
}
void test(int a)
{
printf("a=%d\n",a);
}
void main()
{
fun1(&test);
}
```

**Output:**

a=20

# Array of Function Pointer

```
#include<stdio.h> void
add(int a,int b)
{
    printf("add=%d\n",a+b);
}
void sub(int a,int b)
{
    printf("sub=%d\n",a-b);
}
void mul(int a,int b)
{
    printf("mul=%d\n",a*b);
}
```

```
void main()
{
    Void (*fun[])(int,int)={add,sub,mul};
    int ch;
    int a,b;
    printf("Enter a and b\n");
    scanf("%d%d",&a,&b);
    printf("Enter the operation\n");
    scanf("%d",&ch); if(ch>2)
        printf("Wrong Input\n"); else
        (*fun[ch])(a,b);
}
```

OUTPUT:

Enter a and b

4 5

Enter the operation 2

mul=20



## UNIT – III

### 1.1 Function

#### Definition

- Functions are created when the same process or an algorithm repeated several times in various places in the program.
- A function is a set of instructions that are used to perform specified tasks which repeatedly occurs in the main program.
- A function is a self-contained block or a sub-program of one or more statements that performs a special task when performed. A function is often executed (called) several times, from several different places, during a single execution of the program. After finishing a subroutine, the program will return back to the point from where the function is called.

A C program is nothing but a combination of one or more functions. Every C program starts with a user defined function main().

#### Advantage of Using Function:

1. Reduce the source code
2. Easy to maintain and modify
3. Easy to debug and to test
4. It can be called anywhere in the program.
5. Concept of modular programming
6. Better Readability
7. Information Hiding

### 1.2 Types of C Functions

*Functions are classified based on the following*

- *1.2.1 Based on who develop the function*
  - Library Functions.
  - User defined functions
- *1.2.2 Based on the number of arguments a function accepts*

#### 1.2.1.a Library functions

- **Predefined or library functions are the in-built functions** in C programming system.
- For these functions, function prototype and data definitions are written in their respective header file.
- Each header file contains one or more function declarations, data type definitions and macros.
- The library provides a basic set of mathematical, string manipulation, type conversions, file and console-based I/O functions.

**Example :**

```
printf(),scanf(),gets(),puts(),strlen(),strcpy(),pow(),exp(),sqrt();
```

**Use of library functions**

- The main advantage of the standard library is that it provides an easy working environment than other languages and consequently porting C to a new platform is relatively easy.
- C language provides built-in functions or intrinsic functions called Library Functions. The compiler itself evaluates these functions.

**List of Library functions**

S.No	Functions / Syntax	Meaning	Example
1	sqrt(x)	$\sqrt{x}$	sqrt(25)
2	log(x)	Log <sub>e</sub> x	log(7.5)
3	abs(x)	x	abs(-75)
4	fabs(x)	x	fabs(4.65)
5	exp(x)	E <sup>x</sup>	exp(5.3)
6	pow(x, y)	x <sup>y</sup>	pow(3, 2)
7	ceil(x)	Rounding x to the next integer value	ceil(5, 6)
8	fmod(x, y)	Returns the remainder of x/y	fmod(7, 3)
9	rand()	Generates a positive random number	rand()
10	srand(v)	To initialize the random number generator	srand(8)
11	sin(x)	sin value of x	sin(30)
12	cos(x)	cos value of x	cos(60)
13	tan(x)	tan value of x	tan(90)
14	toascii(x)	Returns the integer value for the particular character	toascii(a)
15	tolower(x)	To convert the character to lowercase	tolower('Z')
16	toupper(x)	To convert the character to upper case	toupper('a')

**1.2.1.b. User Defined Functions**

- The functions defined by the users according to their requirements are called **user-defined functions**. The users can modify the function according to their requirement.
- The user has full scope to implement their own ideas in the function.
- They are written by the programmer to perform a particular task that is repeatedly used in main program.
- These functions are helpful to break down a large program in to a number of smaller functions.

### Need for user defined functions

- When a complex program is written under the main() function then it leads to number of problems like,
  - The program becomes too large and complex.
  - The users can not go through at a glance.
  - The task of debugging, testing and maintenance becomes difficult.
- If the program is divided into parts, then each part may be independently coded and later combined into a single program. These sub-programs are called functions and much easier to understand, debug and test.

### Advantages of user defined functions

- The length of the source program can be reduced by dividing in into the smaller functions.
- By using functions it is very easy to locate and debug an error.
- The user-defined function can be used in many other source programs whenever necessary.
- Functions avoid coding of repeated programming of the similar instructions.
- Functions facilitate top-down programming approach.

### Difference between Library function and User defined function:

S. No	Library Function	User Defined function
1	Library functions are pre defined functions	User defined functions are the function which are created by the user as per his own requirements
2	Written in separate header file	Written in the program
3	Function name is given by developers	Function name is decided by user
4	Name of the functions cannot be changed	Name of the functions can be changed at any time
5	Example: sin(), cos(), power()	Example: fibo(), add(), cube(), fn1()

### 1.3 Aspects of C Function (or) Elements of functions:

1. Function Declaration or Function Prototype
2. Function Definition or Called Functions
3. Function call or Using the Function or Calling Function

S.No	C function aspects	syntax
1	function declaration	return_type function_name ( argument list );
2	function definition	return_type function_name ( arguments list ) { Body of function; }
3	function call	function_name ( arguments list );

#### 1.3.1. Function Declaration or Function Prototype

A Function declaration tells the compiler about a function's name, return type and parameters. The program or a function that calls a function is referred as the calling function. It should be declared in the main or source program.

Function declaration consists of four parts. They are

- Function return type
- Function name
- Parameter / argument list
- Terminating semicolon

**Syntax**

```
return_type function_name ( parameters or argument list );
```

**Where**

- ✓ **return type** of a function can be char, int, float, int\*, void.
- ✓ **function name** are identifiers.
- ✓ **Parameters** can be actual parameter and formal parameters.
- ✓ **Parameter list** is a comma separated by parameter type followed by parameter name.

**Example**

```
int mul(int x, int y);
```

int – return type

mul– function\_name

int x, int y – arguments or variables of types ‘int’

**1.3.2 Function Definition or Called Function**

**Function definition or Called Function provides the actual body of the function, that contains all the statements to be executed.**

**Syntax**

```
return_type function_name ( parameters or argument list ) → Function header
{
  // local variable declarations
  // statements to be executed
  return (return value);
}
```

} → **Function body**

*The elements of function definition are*

1. function name – user definition name
2. return type (function type) – Specifies the type of value to be returned to the calling function. Void data type specifies that no value (empty) is returned.
3. parameters or arguments – Any number of parameters or arguments with specific type.
4. local variable declarations – Specifies the variables required by the function.
5. function statements – Performs the task of the function
6. return statements – Returns the value evaluated by the function

**1.3.3 Function Call Statement or Calling Function**

**In order to use functions user need to call the function at a required place in the program. This is known as the function call.** A function can be called by using the function name followed by a list of actual parameters if any, enclosed in parentheses.



**Syntax:**

```
function_name();
function_name(parameter types);
var_name= function_name();
var_name= function_name(parameters);
```

**Working of a Function**

The working of a function is given below:

```
void abc(int,int);
void main()      // Calling function
{
    ----;
    ----;
    abc(x,y);    // Function call
                  // x, y are actual arguments
    ----
    ----
}
abc(int l, int K)    // Function Definition or called function
{
    // l, k are formal or dummy arguments
    ---
    ----
    ----
    return();      //return value
}
```

1. **Actual arguments:** The arguments of calling functions are called actual arguments.
2. **Formal/Dummy arguments:** The arguments of called functions are called formal or dummy arguments.
3. **Function Name:** A name given to the function similar to a name given to a variable.
4. **Argument/Parameter List:** The variable name enclosed within the parenthesis is the argument list.
5. **Function call:** A C compiler executes the function when a semi-colon is followed by function name.
6. **Variables:** There are two kinds of variables. They are 1. local and 2. global.
7. **Local variables:** The variables which are declared inside the function definition is called local variable.
8. **Global Variable:** Variables which are declared outside the main function is called global variable.
9. **Return value:** The result obtained by the function is sent back by the function to the function call through the return statement. It returns one value per call.

## **1.4 Types of Function or Function prototypes** **(Based on number of arguments a function accepts)**

The Function prototypes are classified into four types. They are:

1. Function with no argument and no return value.
2. Function with no argument and with return value.
3. Function with argument and no return value.
4. Function with argument and with return value.

### **1.4.1 Function with no argument and no return value:**

- Neither data is passed through the calling function nor the data is sent back from the called function.
- There is no data transfer between calling and the called function.
- The function is only executed and nothing is obtained.
- The Function acts independently. It reads data values and print result in the same block.

#### **Syntax:**

```
void Subfunction();
main()
{
....
function();
...
}
function() // → Called function
{
// Input statement;
//process;
//Output statement;
}
```

#### **Example program:**

```
#include<stdio.h>
void add();
void main()
{
clrscr();
add();
getch();
}
void add()
{
int a,b,sum;
printf("Enter a & b values:");
scanf("%d%d",&a,&b);
sum=a+b;
printf("\nThe sum=%d",sum);
}
```

#### **OUTPUT:**

```
Enter a & b values:
10
5
The sum=15
```

### **1.4.2. Function with no argument and with return value**

- In the above type of function, no arguments are passed through the main function. But the called function returns the value.
- The called function is independent. It reads values from the keyboard and returns value to the function call.
- Here both the called and calling functions partly communicate with each other.

**Syntax:**

```

returntype function();
main()
{
....
A=function();
...//Output Statement
}
returntype function() //
{// Input statement;
//process;
return(A);
}

```

**Example program:**

```

#include<stdio.h>
int add();
void main()
{
int sum;
clrscr();
sum=add();
printf("\nThe sum=%d",sum);
getch();
}

```

```

int add()
{
int a,b;
printf("Enter a & b values:");
scanf("%d%d",&a,&b);
return(a+b);
}

```

**OUTPUT:**

```

Enter a & b values:
10
5
The sum=15

```

**1.4.3. Function with argument and no return value**

- In the above type of function, arguments are passed through the calling function. The called function operates on the values. But no result is sent back.
- The functions are partly dependent on the calling function. The result obtained is utilized by the called function.

**Syntax**

```

void function( int,int);
main()
{
....//Input statement;
function(n1,n2);
...
}
function(int a,int b)
{
//process;
//output statement;
}

```

**Example program:**

```

#include<stdio.h>
void add(int,int);
void main(){
int a,b;
clrscr();
printf("Enter a & b values:");
scanf("%d%d",&a,&b);
add(a,b);
getch();
}
void add(int a,int b)
{
int sum;
sum=a+b;
printf("\nThe sum=%d",sum);
}

```

**OUTPUT:**

```

Enter a & b values:
10
5
The sum=15

```

#### 1.4.4. Function with argument and with return value

- In the above type of function, data is transferred between calling and called function.
- Both communicate with each other. The called function receives some data from the calling function and then process it and return the value to the calling function.

##### **Syntax**

```
returntype function( int,int);
    main()
    {
    ....//Input statement;
    function(n1,n2);
    //Output Statement;
    }
returntype function(int a,int b)
{
//process;
//return statement;
}
```

##### **Example Program:**

```
#include<stdio.h>
int add(int,int);
void main()
{
int a,b,sum;
clrscr();
printf("Enter a & b values:");
scanf("%d%d",&a,&b);
sum=add(a,b);
printf("\nThe sum=%d",sum);
getch();
}
void add(int a,int b)
{
return(a+b);
}
```

##### **OUTPUT:**

```
Enter a & b values:
10
5
The sum=15
```

### 1.4.5 Sample Program Using Function – Example //Area of a Circle.C

Function without arguments and without return values	Functions without arguments and with return values	Functions with arguments and without return values	Functions with arguments and with return values
<pre>#include&lt;stdio.h&gt; #include&lt;conio.h&gt; void area(); void main() { clrscr(); area(); getch(); } void area() { float r,area1; printf("Enter radius:\n "); scanf("%f",&amp;r); area1=3.14*r*r; printf("Area=%f",area1); }</pre>	<pre>#include&lt;stdio.h&gt; #include&lt;conio.h&gt; float area(); void main() { float area1; clrscr(); area1=area(); printf("Area=%f",area1); getch(); } float area() { float r; printf("Enter radius:\n "); scanf("%f",&amp;r); return(3.14*r*r); }</pre>	<pre>#include&lt;stdio.h&gt; #include&lt;conio.h&gt; void area(float); void main() { float r; clrscr(); printf("Enter radius:\n "); scanf("%f",&amp;r); area(r); getch(); } void area(float r) { float area1; area1=3.14*r*r; printf("Area=%f",area1); }</pre>	<pre>#include&lt;stdio.h&gt; #include&lt;conio.h&gt; float area(float); void main() { float r, area1; clrscr(); printf("Enter radius:\n "); scanf("%f",&amp;r); area1=area(); printf("Area=%f",area1); getch(); } float area(float r) { return(3.14*r*r); }</pre>

#### //cube.C

Function without arguments and without return values	Functions without arguments and with return values	Functions with arguments and without return values	Functions with arguments and with return values
<pre>#include&lt;stdio.h&gt; #include&lt;conio.h&gt; void cube(); void main() { clrscr(); cube(); getch(); } void area() { int n, cube1; printf("Enter number: "); scanf("%d",&amp;n); cube1=n*n*n; printf("Cube=%d",cube1); }</pre>	<pre>#include&lt;stdio.h&gt; #include&lt;conio.h&gt; int cube(); void main() { int cube1; clrscr(); cube1=cube(); printf("Cube=%d",cube1); getch(); } int cube() { int n; printf("Enter number: "); scanf("%d",&amp;n); return(n*n*n); }</pre>	<pre>#include&lt;stdio.h&gt; #include&lt;conio.h&gt; void cube(int); void main() { int n; clrscr(); printf("Enter number: "); scanf("%d",&amp;n); cube(n); getch(); } void cube(int n) { int cube1; cube1 = n*n*n; printf("Cube=%d",cube1); }</pre>	<pre>#include&lt;stdio.h&gt; #include&lt;conio.h&gt; int cube(int); void main() { int n, cube1; clrscr(); printf("Enter number: "); scanf("%d",&amp;n); cube1=cube(); printf("Cube=%d",cube1); getch(); } int cube(int n) { return(n*n*n); }</pre>

//Function with arguments and return types// Area and Circumference of the circle

```
#include<stdio.h>
#include<conio.h>
float area_cir(float);
float circum_cir(float);
void main()
{
    float rad,area,circum;
    printf("\nEnter the radius:");
    scanf("%f",&rad);
    area=area_cir(rad);
    circum=circum_cir(rad);
    printf("\nArea of the circle is  %f",area);
    printf("\nCircumference of the circle is %f ",circum);
    getch();
}
float are_cir(float r)
{
    return(22/7*r*r);
}
float circum_cir(float r)
{
    return(2*22/7*r);
}
```

**Sample output:**

```
Enter the radius:3
Area of the circle is 28.285714
Circumference of the circle is
18.857142
```

// simulate Calculator using Function

```

#include<stdio.h>
#include<conio.h>
float add(float,float);
float sub(float,float);
float mul(float,float);
float div(float,float);
void main()
{
float a,b,add1,sub1,mul1,div1;
printf("\nEnter two number:");
scanf("%f%f",&a,&b);
add1=add(a,b);
sub1=sub(a,b);
mul1=mul(a,b);
div1=div(a,b);
printf("\nAddition is %f",add1);
printf("\nSubtraction is %f",sub1);
printf("\nMultiplication is %f",mul1);
printf("\nDivision is %f",div);
getch();
}
float add(float f1,float f2)
{
return(f1+f2);
}
float sub(float f1,float f2)
{
return(f1-f2);
}
float mul(float f1,float f2)
{
return(f1*f2);
}
float div(float f1,float f2)
{
return(f1/f2);
}

```

**Output:**

Enter two number:

63

45

Addition is 108.000000

Subtraction is 18.000000

Multiplication is 2835.000000

Division is 1.400000

// calculate Factorial of a number using Function

```

#include<stdio.h>
#include<conio.h>
int fact(int);
void main()
{
int num,factorial;
clrscr();
printf("\nEnter the number:");
scanf("%d",&num);
factorial=fact(num);
printf("\nFactorial of %d is %d",num,factorial);
getch();
}
int fact(int n)
{
int i,f=1;
for(i=0;i<n;i++)
f*=i;
return(f);
}

```

**Sample output:**

Enter the number: 5  
Factorial of 5 is 120

**1.6 In-Built-Functions:****String In-Built-Functions:**

All C inbuilt functions which are declared in string.h header file

String functions	Description
<u>strcat ( )</u>	Concatenates str2 at the end of str1
<u>strncat ( )</u>	Appends a portion of string to another
<u>strcpy ( )</u>	Copies str2 into str1
<u>strncpy ( )</u>	Copies given number of characters of one string to another
<u>strlen ( )</u>	Gives the length of str1
<u>strcmp ( )</u>	Returns 0 if str1 is same as str2. Returns <0 if str1 < str2. Returns >0 if str1 > str2



<u>strncmpi ( )</u>	Same as strcmp() function. But, this function negotiates case. “A” and “a” are treated as same.
<u>strchr ( )</u>	Returns pointer to first occurrence of char in str1
<u>strrchr ( )</u>	last occurrence of given character in a string is found
<u>strstr ( )</u>	Returns pointer to first occurrence of str2 in str1
<u>strrstr ( )</u>	Returns pointer to last occurrence of str2 in str1
<u>strdup ( )</u>	Duplicates the string
<u>strlwr ( )</u>	Converts string to lowercase
<u>strupr ( )</u>	Converts string to uppercase
<u>strrev ( )</u>	Reverses the given string
<u>strset ( )</u>	Sets all character in a string to given character
<u>strnset ( )</u>	It sets the portion of characters in a string to given character
<u>strtok ( )</u>	Tokenizing given string using delimiter

### Math In-built-Functions

All C inbuilt functions which are declared in math.h header file

Function	Description
<u>floor ( )</u>	This function returns the nearest integer which is less than or equal to the argument passed to this function.
<u>round ( )</u>	This function returns the nearest integer value of the float/double/long double argument passed to this function. If decimal value is from “.1 to .5”, it returns integer value less than the argument. If decimal value is from “.6 to .9”, it returns the integer value greater than the argument.
<u>ceil ( )</u>	This function returns nearest integer value which is greater than or equal to the argument passed to this function.
<u>sin ( )</u>	This function is used to calculate sine value.
<u>cos ( )</u>	This function is used to calculate cosine.
<u>cosh ( )</u>	This function is used to calculate hyperbolic cosine.

<u>exp ( )</u>	This function is used to calculate the exponential “e” to the x <sup>th</sup> power.
<u>tan ( )</u>	This function is used to calculate tangent.
<u>tanh ( )</u>	This function is used to calculate hyperbolic tangent.
<u>sinh ( )</u>	This function is used to calculate hyperbolic sine.
<u>log ( )</u>	This function is used to calculates natural logarithm.
<u>log10 ( )</u>	This function is used to calculates base 10 logarithm.
<u>sqrt ( )</u>	This function is used to find square root of the argument passed to this function.
<u>pow ( )</u>	This is used to find the power of the given number.
<u>trunc.(.)</u>	This function truncates the decimal value from floating point value and returns integer value.

### 1.7 Recursion

- **Recursive Function:– a function that calls itself**
  - **Directly or indirectly**
- **The process of calling the function by itself again and again until some condition is satisfied is called recursive function.**
- Recursive programs must have at least one if statement to terminate recursion, otherwise infinite loop will be created.
- It doesn't contain any looping statement.
- Each recursive call is made with a new, independent set of arguments
  - Previous calls are suspended
- Allows very simple programs for very complex problems

### **The Nature of Recursion**

1. Thinking recursively

2. Every recursive solution consists of two cases:

**a. Base case :** Base case is the smallest instance of problem, which can be easily solved and there is no need to further express the problem in terms of itself, i.e. in this case no recursive call is given and the recursion terminates. Base case forms the terminating condition of the recursion. For example, calculate factorial of a number , base case is  $n==1$ .

**b. recursive case:** The problem is defined in terms of itself, while reducing the problem size. For example, when  $\text{fact}(n)$  is expressed as  $n * \text{fact}(n-1)$ , the size of the problem is reduced from  $n$  to  $n-1$ .

3. Express the solution in the form of base cases and recursive cases

For example,

$$fact(n) = \begin{cases} 1 & \text{when } n = 1 \\ n * fact(n - 1) & \text{when } n > 1 \end{cases}$$

Relation of the above form is known as recurrence relation

In general:

*if (stopping case)*

*solve it*

*else reduce the problem using recursion*

#### Recursive definition:

$$1! = 1$$

$$2! = 2 * 1 = 2 * 1!$$

$$3! = 3 * 2 * 1 = 3 * 2!$$

$$4! = 4 * 3 * 2 * 1 = 4 * 3!$$

$$5! = 5 * 4 * 3 * 2 * 1 = 5 * 4!$$

$$6! = 6 * 5 * 4 * 3 * 2 * 1 = 6 * 5!$$

$$n! = n * (n-1)!$$

#### Example

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

```
int fact(int );
```

```
void main()
```

```
{
```

```
    int n,factorial;
```

```
    clrscr();
```

```
    printf("Enter a number:");
```

```
    scanf("%d",&n);
```

```
    factorial=fact(n);
```

```
    printf("Factorial of %d is %d",n,factorial);
```

```
    getch();
```

```
}
```

```
int fact(int n)
```

```
{    if (n <= 1)
```

```
        return 1;
```

```
    else
```

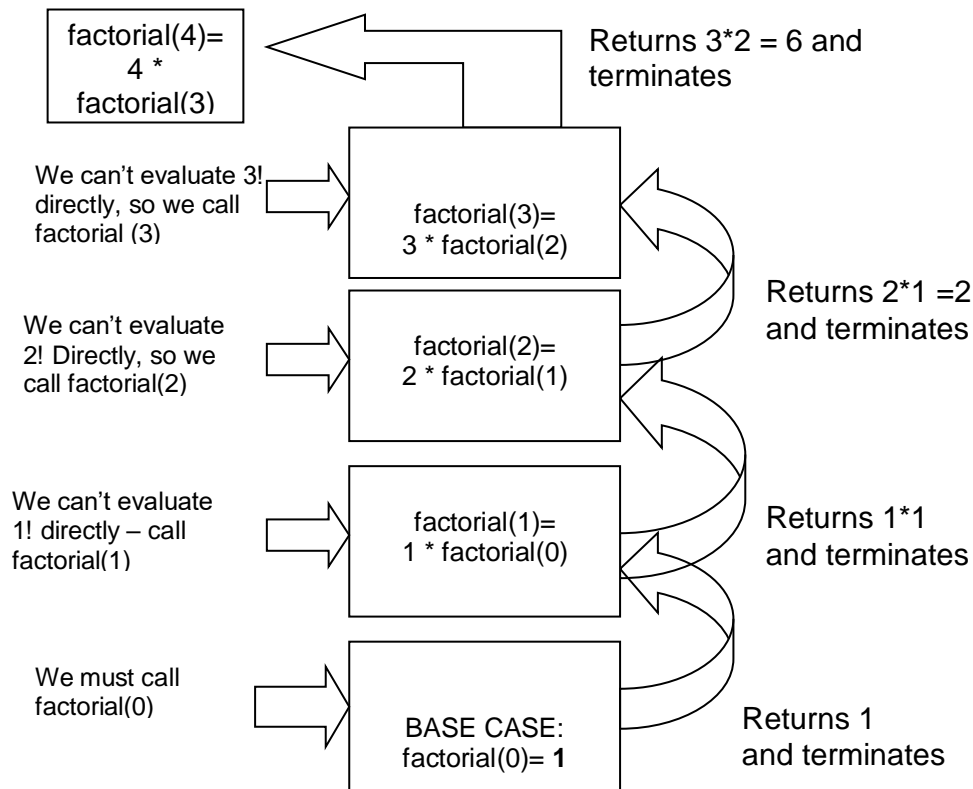
```
        return n * fact (n-1);
```

```
}
```

#### Output:

Enter a number: 5

Factorial of 5 is 120



### //C Program to find Power of a Number using Recursion

```
#include <stdio.h>
long int power (int, int);
int main()
{
    int pow, num;
    long int result;
    printf("Enter a number: ");
    scanf("%d", &num);
    printf("Enter it's power: ");
    scanf("%d", &pow);
    result = power(num, pow);
    printf("%d^%d is %ld", num, pow, result);
    return 0;
}
long int power (int num, int pow)
{
    if (pow)
    {
        return (num * power(num, pow - 1));
    }
    return 1;
}
```

**Output:**

Enter a number: 456

Enter it's power: 3

$456^3$  is 94818816

**//C Program to find Sum of Digits of a Number using Recursion**

```
#include <stdio.h>
int sum (int a);
void main()
{
    int num, result;
    printf("Enter the number: ");
    scanf("%d", &num);
    result = sum(num);
    printf("Sum of digits in %d is %d\n", num, result);
}
int sum (int num)
{
    if (num != 0)
    {
        return (num % 10 + sum (num / 10));
    }
    else
    {
        return 0;
    }
}
```

**Output:**

Enter the number: 2345

Sum of digits in 2345 is 14

**Example Program:****Computation of Sine series:**

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

void main()
{
    int i, n;
    float x, sum, t;
    clrscr();

    printf(" Enter the value for x : ");
    scanf("%f",&x);

    printf(" Enter the value for n : ");
    scanf("%d",&n);

    x=x*3.14159/180;
    t=x;
    sum=x;

    /* Loop to calculate the value of Sine */
    for(i=1;i<=n;i++)
    {
        t=(t*(-1)*x*x)/(2*i*(2*i+1));
        sum=sum+t;
    }

    printf(" The value of Sin(%f) = %.4f",x,sum);
    getch();
}
```

**Output:**

```
Enter the value for x : 45
Enter the value for n: 5
The value of Sin(0.785398)=0.7071
```

Scientific calculator using built-in functions:

```

#include<stdio.h>
#include<conio.h>
#include<math.h>

void main()
{
    int choice, i, a, b;
    float x, y, result;
    clrscr();
    do {
        printf("\nSelect your operation (0 to exit):\n");
        printf("1. Addition\n2. Subtraction\n3. Multiplication\n4. Division\n");
        printf("5. Square root\n6.  $X^Y$ \n7.  $X^2$ \n8.  $X^3$ \n");
        printf("9.  $1/X$ \n10.  $X^{(1/Y)}$ \n11.  $X^{(1/3)}$ \n");
        printf("12.  $10^X$ \n13.  $X!$ \n14.  $\log_{10}(x)$ \n15. log10(x)\n16. Modulus\n");
        printf("17. Sin(X)\n18. Cos(X)\n19. Tan(X)\n20. Cosec(X)\n");
        printf("21. Cot(X)\n22. Sec(X)\n");
        printf("Choice: ");
        scanf("%d", &choice);
        switch(choice) {
            case 1:
                printf("Enter X: ");
                scanf("%f", &x);
                printf("\nEnter Y: ");
                scanf("%f", &y);
                result = x + y;
                printf("\nResult: %f", result);
                break;
            case 2:
                printf("Enter X: ");
                scanf("%f", &x);
                printf("\nEnter Y:");
                scanf("%f", &y);
                result = x - y;
                printf("\nResult: %f", result);
                break;
            case 3:
                printf("Enter X: ");
                scanf("%f", &x);
                printf("\nEnter Y:");
                scanf("%f", &y);
                result = x * y;
                printf("\nResult: %f", result);
                break;

```

```
case 4:
printf("Enter X: ");
scanf("%f", &x);
printf("\nEnter Y:");
scanf("%f", &y);
result = x / y;
printf("\nResult: %f", result);
break;
case 5:
printf("Enter X: ");
scanf("%f", &x);
result = sqrt(x);
printf("\nResult: %f", result);
break;
case 6:
printf("Enter X: ");
scanf("%f", &x);
printf("\nEnter Y:");
scanf("%f", &y);
result = pow(x, y);
printf("\nResult: %f", result);
break;
case 7:
printf("Enter X: ");
scanf("%f", &x);
result = pow(x, 2);
printf("\nResult: %f", result);
break;
case 8:
printf("Enter X: ");
scanf("%f", &x);
result = pow(x, 3);
printf("\nResult: %f", result);
break;
case 9:
printf("Enter X: ");
scanf("%f", &x);
result = pow(x, -1);
printf("\nResult: %f", result);
break;
case 10:
printf("Enter X: ");
scanf("%f", &x);
printf("\nEnter Y: ");
scanf("%f", &y);
result = pow(x, (1/y));
```



```
printf("\nResult: %f", result);
break;
case 11:
printf("Enter X: ");
scanf("%f", &x);
y = 3;
result = pow(x, (1/y));
printf("\nResult: %f", result);
break;
case 12:
printf("Enter X:");
scanf("%f", &x);
result = pow(10, x);
printf("\nResult: %f", result);
break;
case 13:
printf("Enter X: ");
scanf("%f", &x);
result = 1;
for(i = 1; i <= x; i++) {
result = result * i;
}
printf("\nResult: %.f", result);
break;
case 14:
printf("Enter X: ");
scanf("%f", &x);
printf("\nEnter Y:");
scanf("%f", &y);
result = (x * y) / 100;
printf("\nResult: %.2f", result);
break;
case 15:
printf("Enter X:");
scanf("%f", &x);
result = log10(x);
printf("\nResult: %.2f", result);
break;
case 16:
printf("Enter a: ");
scanf("%d", &a);
printf("\nEnter b: ");
scanf("%d", &b);
result = a % b;
printf("\nResult: %d", result);
break;
```

```
case 17:
printf("Enter X: ");
scanf("%f", &x);
result = sin(x*3.1419/180);
printf("\nResult: %.2f", result);
break;
case 18:
printf("Enter X: ");
scanf("%f", &x);
result = cos(x * 3.14159 / 180);
printf("\nResult: %.2f", result);
break;
case 19:
printf("Enter X:");
scanf("%f", &x);
result = tan(x * 3.14159 / 180);
printf("\nResult: %.2f", result);
break;
case 20:
printf("Enter X: ");
scanf("%f", &x);
result = 1 / (sin(x * 3.14159 / 180));
printf("\nResult: %.2f", result);
break;
case 21:
printf("Enter X: ");
scanf("%f", &x);
result = 1 / tan(x * 3.14159 / 180);
printf("\nResult: %.2f", result);
break;
case 22:
printf("Enter X: ");
scanf("%f", &x);
result = 1 / cos(x * 3.14159 / 180);
printf("\nResult: %.2f", result);
break;
default:
printf("\nInvalid Choice!");
}} while(choice<23);
getch();
}
```

**Output:**

```

nSelect your operation (0 to exit):
1. Addition      2. Subtraction  3. Multiplication      4. Division      5. Squar
e root  6. X ^ Y      7. X ^ 2      8. X ^ 3      9. 1 / X      10. X ^
(1 / Y) 11. X ^ (1 / 3) 12. 10 ^ X      13. X!  14. %      15. log10(x)  16. Modu
lus      17. Sin(X)      18. Cos(X)      19. Tan(X)      20. Cosec(X)  21. Cot(
X)      22. Sec(X)
Choice: 1
Enter X: 3
Enter Y: 4
Result: 7.000000
Choice: 7
Enter X: 6
Result: 36.000000
Choice: 17
Enter X: 67
Result: 0.92
Choice: 19
Enter X:89
Result: 57.29
Choice: _

```

**Binary Search using recursive functions**

```

#include<stdio.h>
#include<stdlib.h>
#define size 10

int binsearch(int[], int, int, int);

int main() {
    int num, i, key, position;
    int low, high, list[size];
    clrscr();

    printf("\nEnter the total number of elements");
    scanf("%d", &num);

    printf("\nEnter the elements of list :");
    for (i = 0; i < num; i++) {
        scanf("%d", &list[i]);
    }

    low = 0;

```

```

high = num - 1;

printf("\nEnter element to be searched : ");
scanf("%d", &key);

position = binsearch(list, key, low, high);

if (position != -1) {
    printf("\nNumber present at %d", (position + 1));
} else
    printf("\n The number is not present in the list");
return (0);
}

// Binary Search function
int binsearch(int a[], int x, int low, int high) {
    int mid;

    if (low > high)
        return -1;

    mid = (low + high) / 2;

    if (x == a[mid]) {
        return (mid);
    } else if (x < a[mid]) {
        binsearch(a, x, low, mid - 1);
    } else {
        binsearch(a, x, mid + 1, high);
    }
    return 0;
}

```

### **Output:**

Enter the total number of elements 5

Enter the elements of list : 23

45

78

90

100

Enter the element to be searched: 78

Number present at 3

## 1.8 POINTERS

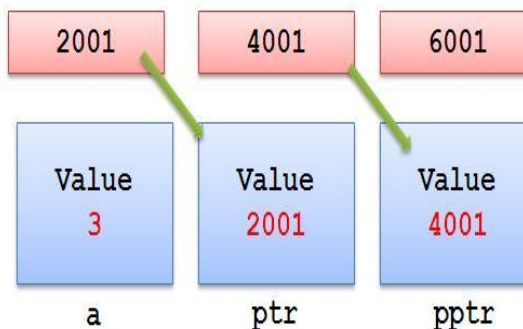
### 1.8.1. Definition:

- Pointer is a variable which stores the address of another variable
- Since Pointer is also a kind of variable , thus pointer itself will be stored at different memory location

### Types of Variables :

1. Simple Variable that stores a value such as integer,float,character
2. Complex Variable that stores address of simple variable i.e pointer variables

```
main()
{
int a = 3;
int *ptr,**pptr;
ptr = &a;
pptr = &ptr;
printf("\n The value of a = %d",a);//3
printf("\nThe address of a =%u ",ptr);//2001
printf("\nThe address of ptr=%u",pptr);//4001
}
```



### 2. Pointer Declaration:

**Syntax:**        data\_type \*var\_name;

Where, data\_type → valid type data

Var\_name → valid variable name

#### **For example;**

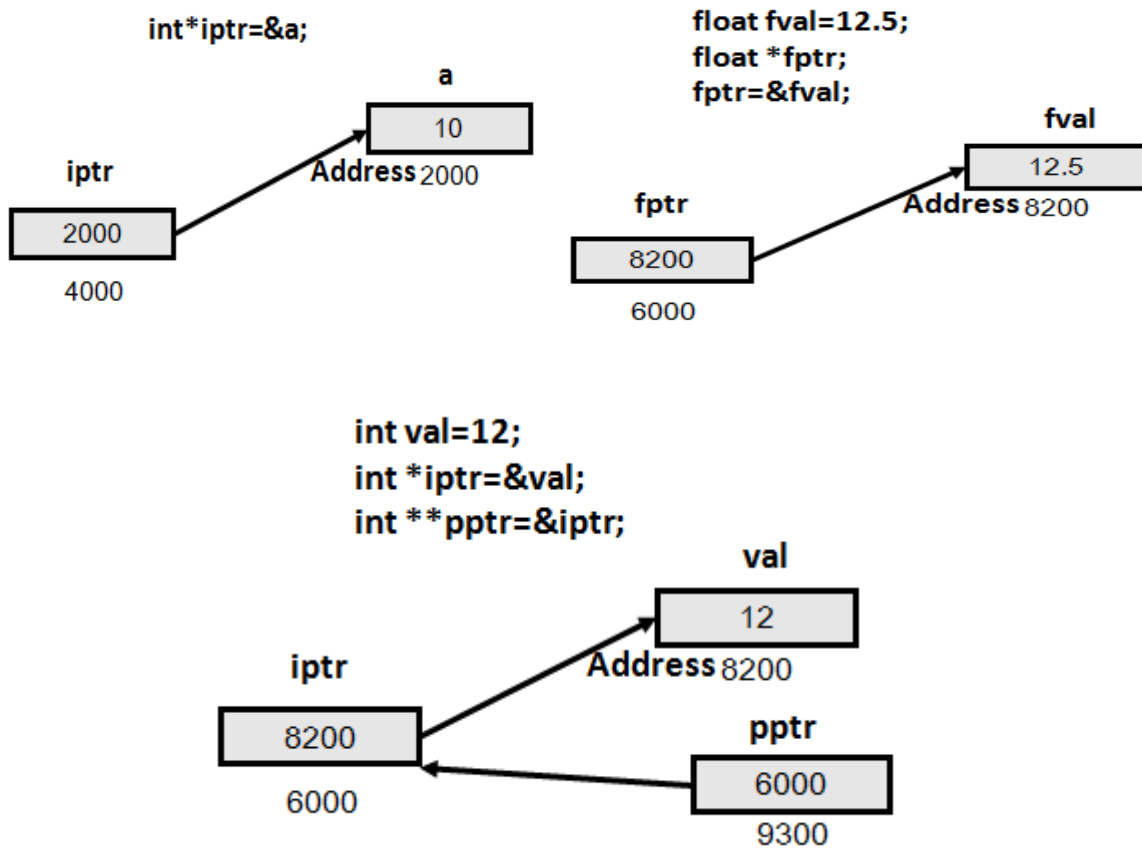
```
int *ip;      //ip is pointer to an integer
double *dp;   // dp is pointer to a double
float *fp;    //fp is pointer to a float
char *ch      //ch is pointer to a character
const int *ptric; //ptric is pointer to an integer constant or constant integer
```

### **3. How to Read:**

Pointer variable declarations are read from the right side.

i.e, int \*ip; read as ip is pointer to an integer

#### 4. A pointer variable can hold the address of a variable or a function:



5. Every pointer variable takes the same amount of memory space irrespective of whether it is a pointer to int, float, char, or any other type.

#### //Size of pointer variables

```
#include<stdio.h>
main()
{
    int *p;
    float *q;
    char *r;
    printf("Pointer to integer takes %d bytes", sizeof(p));
    printf("Pointer to float takes %d bytes ", sizeof(q));
    printf("Pointer to character takes %d bytes ", sizeof(r));
}
```

#### **Output:**

Pointer to integer takes 2 bytes  
 Pointer to float takes 2 bytes  
 Pointer to character takes 2 bytes

6. Format specifier for pointer variable is %p or %u.

## 1.8.2 .Operations on Pointers

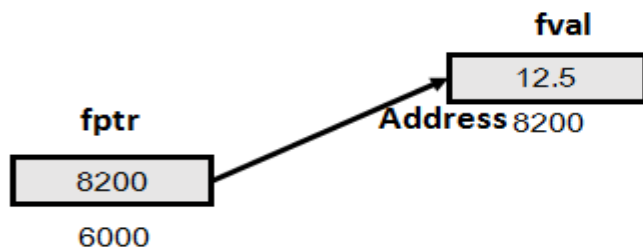
### 1.8.2.1. Referencing Operation

In referencing operation, a pointer variable is made to refer to an object. The reference to an object can be created with the help of a reference operator (i.e &)

Important points about the reference operator(&)

1. The reference operator, i.e, & is a unary operator and should appear on the left side of its operand.
2. The operand of the reference operator should be a variable of arithmetic type or pointer type.
3. The reference operator is also known as a address- of operator

```
float fval=12.5;    // fval is a floating point variable initialized with 12.5
float *fptr;        // fptr is a pointer to float type
fptr=&fval;          //The address-of fval is assigned to fptr, fval is known as
                    //referenced object and fptr is known as referencing object and
```



### 18.2.2. Dereferencing a Pointer

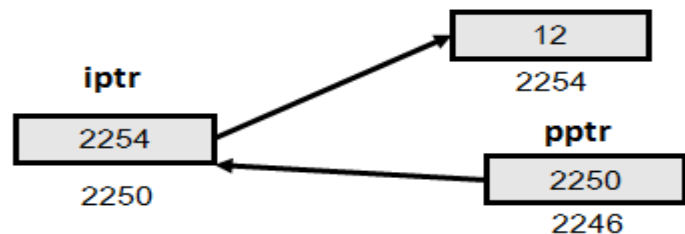
The object pointed to or referenced by a pointer can be indirectly accessed by dereferencing the pointer. A dereferencing operation allows a pointer to be followed to the data object to which it points . A pointer can be dereferenced by using a dereference operator (i.e.\*).

Important points about the dereference operator(\*)

- a. The dereference operator, i.e, \* is a unary operator and should appear on the left side of its operand.
- b. The operand of the dereference operator should be of pointer type.
- c. The dereference operator is also known as **indirection operator** or **value –at operator**.

**//Dereferncing pointers**

```
#include<stdio.h>
main()
{
int val=12;
int *iptr=&val;
int **pptr=&iptr;
printf("value is %d\n",val);
printf("value by dereferencing iptr is %d\n", *iptr);
printf("value by dereferencing pptr is %d\n", *pptr);
printf("value iptr is %p\n", iptr);
printf("value pptr is %p\n", pptr);
}
```

**Output:**

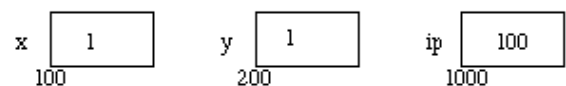
value is 12  
value by dereferencing iptr is 12  
value by dereferencing pptr is 12  
value iptr is 2254  
value pptr is 2250

```
int x = 1, y = 2;
int *ip;
```

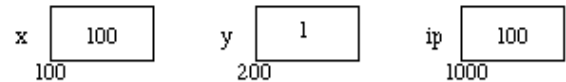
```
ip = &x;
```



```
y = *ip;
```



```
x = ip;
```

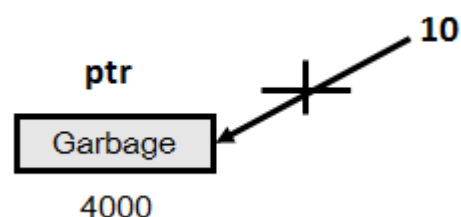


```
*ip = 3
```

**1.8.2.3. Assigning to a Pointer****Rule-1.:**

A pointer can be assigned or initialized with the address of an object. A pointer variable cannot hold a non-address value and thus can only be assigned or initialized with 1-values.

```
main()
{
int val=10;
int*ptr=val;
printf("Value of variable is %d\n", val);
printf("Pointer holds %p\n",ptr);
}
```

**Output**

Compilation error "Cannot convert int to int\*"



**Reasons:**

- Pointer variables can only hold addresses
- A pointer variable ptr cannot hold an integer value val

**What to do?**

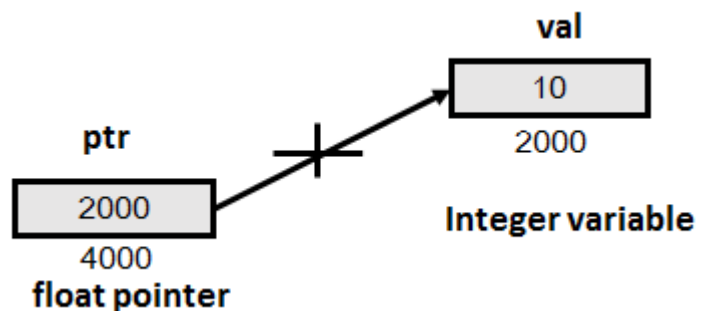
- Initialize ptr with the address of variable val by writing &val and re-execute the code

There is an expression to this rule. The constant zero can be assigned to a pointer. For example `int *ptr=0;` is valid Assignment or initialization with zero makes the pointer a special pointer known as the **NULL pointer**

**Rule 2.**

A pointer to a type cannot be initialized or assigned the address of an object of another type.

```
main()
{
int val=10;
float *ptr=&val;
printf(Value of variable is %d\n", val);
printf(Pointer holds %p\n", ptr);
}
```

**Output**

Compilation error “Cannot convert int\* to float\*”

**Reasons:**

- A pointer variable can only be assigned address of an object of the same type
- A pointer variable ptr (of type float\*) cannot hold the address of an integer variable (i.e. int\*)

**What can be done?**

- Explicitly type cast int\* to float\* by using type cast operator. Write `float* ptr=(float*)&val;` and then re-execute the code.

**Remark:**

- Explicit type casting of pointers may give unexpected results and is not recommended.

**Rule 3.**

A pointer can be assigned or initialized with another pointer of the same type. However, it is not possible to assign a pointer of one type to a pointer of another type without explicit type casting.

There is an expression to Rules 2 & 3. A Pointer to any type of object can be assigned to a pointer of type `void*` but vice-versa is not true. A **void pointer** can't be assigned to a pointer to a type without explicit type casting.

### 1.8.3. Features of Pointers:

- Pointers save the memory space.
- Execution time with pointer is faster because data is manipulated with the address.
- The memory is accessed efficiently with the pointers.
- Dynamically, memory is allocated.
- Pointers are useful for representing two-dimensional and multi-dimensional arrays.

### 1.9. Pointer Arithmetic

Arithmetic operations can be applied to pointers in a restricted form . When arithmetic operators are applied on pointers, the outcome of the operation is governed by **pointer arithmetic** .

Here are some pointers on arithmetic operations, there are

1. Addition Operation (Addition of Pointer and Integer Number) (+)
2. Subtraction Operation (a. Subtraction of Pointer and Number  
b. Differencing between two pointers)(-)
3. Increment Operation (Incrementing Pointer) (++)
4. Decrement Operation (Decrementing Pointer) (-- )
5. Relational (Comparison) Operations (<,<=,>,>=,==,!=)

#### 1.9.1. Addition Operation (Addition of Pointer and Integer Number) (+)

2. An expression of integer type can be added to an expression of pointer type. The result of such operation would have the same type as that of pointer type operand.
3. if ptr is a pointer to an object, then 'adding 1 to pointer' (i.e ptr + 1) points to the next object.
4. Similarly, ptr+ i would point to the i<sup>th</sup> object.
5. Addition of two pointers is not allowed.
6. The addition of a pointer and an integer is commutative , i.e. ptr + 1 is same as 1 + ptr.

#### How to determine ?

***Result = initial value of pointer + integer operand \* sizeof (the reference type T)***

S.No	Operator	Type of operand 1	Type of Operand 2	Resultant type
1	Addition operator (+)	Pointer to type T	int	Pointer to type T
		float*	int	float*
		int*	int	int*
		double*	int	double*
2		Pointer	Pointer	Not Allowed
		double*	double*	

S.No	Example	Initial value	Final value	Solution
1	int *ptr; ptr=ptr+3;	ptr=2000	ptr=2006	2000 + 3*(2) =2006
2	double *p2; p2=p2+7;	2000	2056	2000 + 7 * (8) =2056

**//example.c**

```
main()
{
int *ptr=(int *)1000;
ptr=ptr+3;
printf("New Value of ptr : %u",ptr);
}
//example.c
```

**Ouput:**

New Value of ptr : 1006

**Ouput:**

New Value of ptr : 2020

**//example 1 .c**

```
main()
{
float ptr=(float *)2000;
ptr=ptr+5;
printf("New Value of ptr : %u",ptr);
}
```

**Ouput:**

New Value of ptr : 2020

**1.9.2 Subtraction Operation**

1. A pointer and an integer can be subtracted.
2. Subtraction of integer and pointer is not commutative, i.e. ptr-1 is not the same as 1- ptr . The operation 1- ptr is illegal.
3. Two pointers can also be subtracted . Pointer subtraction is meaningful only if both the pointers point to the elements of the same array.

**How to determine (Subtraction of Pointer and Number)?**

*Result = initial value of pointer - integer operand \* sizeof (the reference type T)*

**How to determine (Differencing between two pointers)?**

*Result = (operand1-operand2)/ sizeof (the reference type T)*

S.No	Operator	Type of operand 1	Type of Operand 2	Resultant type
1	Subtraction operator (-)	Pointer to type T	int	Pointer to type T
		float*	int	float*
		int*	int	int*
		double*	int	double*
2		Pointer	Pointer	int
		double*	double*	int

S.No	Example	Initial value	Final value	Solution
1	int *ptr; ptr=ptr-3;	ptr=2000	ptr=1994	2000 - 3*(2) =1994
2	float*p2; p2=p2-4;	4000	3984	4000 -4*4 =3984

**//example.c**

```
main()
{
int *ptr=(int *)1000;
ptr=ptr-3;
printf("New Value of ptr : %u",ptr);
}
```

**Ouput:**

New Value of ptr : 994

**//example.c**

```
main()
{
float ptr=(float *)2000;
ptr=ptr-5;
printf("New Value of ptr : %u",ptr);
}
```

**Ouput:**

New Value of ptr : 1980

**//example.c**

```
main()
{
int num , *ptr1 ,*ptr2 ;
ptr1 = &num ;
ptr2 = ptr1 + 2 ;
printf("%d",ptr2 - ptr1);
}
```

**Ouput:**

2

**Explanation:**

- ptr1 stores the **address of Variable** num
- Value of ptr2 is incremented by **4 bytes**
- Differencing two Pointers

**1.9.3 Increment Operation (Incrementing Pointer) (++)**

- The increment operator can be applied to an operand of pointer type.
- Incrementing Pointer is generally used in array because we have contiguous memory in array and we know the contents of next memory location.
- Incrementing Pointer Variable Depends Upon data type of the Pointer variable

**How to determine ?****Post Increment:**

*Result = initial value of pointer*

**Pre Increment:**

*Result = initial value of pointer + sizeof (the reference type T)*

*In both the cases:*

*Value of pointer = value of pointer + sizeof (the reference type T)*

S.No	Operator	Type of operand	Resultant type
1	Increment operator (++)	Pointer to type T	Pointer to type T
		float*	float*
		int*	int*
		double*	double*

**//example.c**

```
main()
{
int *ptr=(int *)1000;
int *ptr1,*ptr2;
ptr1=ptr++;
ptr2=++ptr;
printf("Value of ptr1 : %u\n",ptr1);
printf("Value of ptr2: %u\n",ptr2);
printf("Value of ptr: %u\n",ptr);
}
```

**Output:**

Value of ptr1 : 1000  
Value of ptr2 : 1002  
Value of ptr : 1004

**//example.c**

```
main()
{
double *ptr=(double *)4040;
double *ptr1,*ptr2;
ptr1=++ptr;
ptr2=ptr++;
printf("Value of ptr1 : %u\n",ptr1);
printf("Value of ptr2: %u\n",ptr2);
printf("Value of ptr: %u\n",ptr);
}
```

**Output:**

Value of ptr1 : 4008

Value of ptr2 : 4008

Value of ptr : 4016

**1.9.4 Decrement Operation (Decrementing Pointer) (--)**

- The decrement operator can be applied to an operand of pointer type.
- Decrementing Pointer is generally used in array because we have contiguous memory in array and we know the contents of previous memory location.
- Decrementing Pointer Variable Depends Upon data type of the Pointer variable

**How to determine ?**

**Post Decrement:**

*Result = initial value of pointer*

**Pre Decrement:**

*Result = initial value of pointer - sizeof (the reference type T)*

*In both the cases:*

*Value of pointer = value of pointer - sizeof (the reference type T)*

S.No	Operator	Type of operand	Resultant type
1	Decrement operator (--)	Pointer to type T	Pointer to type T
		float*	float*
		int*	int*
		double*	double*

**//example.c**

```
main()
{
int *ptr=(int *)1000;
int *ptr1,*ptr2;
ptr1=--ptr;
ptr2=ptr--;
printf("Value of ptr1 : %u\n",ptr1);
printf("Value of ptr2: %u\n",ptr2);
printf("Value of ptr: %u\n",ptr);
}
```

**Output:**

Value of ptr1 : 998  
Value of ptr2 : 998  
Value of ptr : 996

**//example.c**

```
main()
{
double *ptr=(double *)4040;
double *ptr1,*ptr2;
ptr1=ptr--;
ptr2=--ptr;
printf("Value of ptr1 : %u\n",ptr1);
printf("Value of ptr2: %u\n",ptr2);
printf("Value of ptr: %u\n",ptr);
}
```

**Output:**

Value of ptr1 : 4040  
Value of ptr2 : 4024  
Value of ptr : 4024

### 1.9.5 Relational (Comparison) Operations:

A pointer can be compared with a pointer of the same type or with zero. A comparison of pointers is meaningful only when they point to the elements of the same array.

S.No	Operator	Type of operand 1	Type of Operand 2	Resultant type
1	Comparison operator (<,<=,>,>=,==,!=)	Pointer to type T	Pointer to type T	int (0→false, 1→true)
		float*	float*	int
		int*	int*	int
		double*	double*	int
2	Pointer Comparison of different data Types:	int*	float*	int
		double*	int*	int

S.No	Example	Expression	Initial value	Final value	Solution
1	float *ptr1,*ptr2; int a;	a=ptr1!=ptr2	ptr1=2000 ptr2=2008	1	a =2000!=2008 a=1
2		a=ptr1<ptr2	ptr1=2000 ptr2=2008	1	a =2000<2008 a=1
3		a=ptr1<=ptr2	ptr1=2000 ptr2=2008	1	a =2000<=2008 a=1
4		a=ptr1>ptr2	ptr1=2000 ptr2=2008	0	a =2000>2008 a=0
5		a=ptr1>=ptr2	ptr1=2000 ptr2=2008	0	a =2000>=2008 a=0
6		a=ptr1==ptr2	ptr1=2000 ptr2=2008	0	a =2000==2008 a=1

#### //example.c

```
main()
{
int *ptr1,*ptr2;
ptr1 = (int *)1000;
ptr2 = (int *)2000;
if(ptr2 > ptr1)
    printf("Ptr2 is far from ptr1");
}
```

#### Output:

Ptr2 is far from ptr1



## Pointer Comparison of Different Data Types :

- **Two Pointers of different data types can be compared .**
- In the above program we have compared two pointers of different data types.
- It is perfectly **legal in C Programming**.

```
main()
{
int *ptr1;
float *ptr2;
ptr1 = (int *)1000;
ptr2 = (float *)2000;
if(ptr2 > ptr1)
    printf("Ptr2 is far from ptr1");
}
```

### Output:

Ptr2 is far from ptr1

## 1.9.6 Illegal Pointer Operations

1. Addition of two pointers are not allowed
2. Only integers can be added to pointer. It is not valid to add a float or a double value to a pointer
3. Multiplication and division operators can't be applied on pointers.
4. Bitwise operators can't be applied on pointers.
5. A pointer variable can't be assigned a non-address value (except Zero)
6. A pointer of one type can't be assigned to a pointer of another type (except void\* ) without explicit type casting.

## 1.10. Pointers and Arrays:

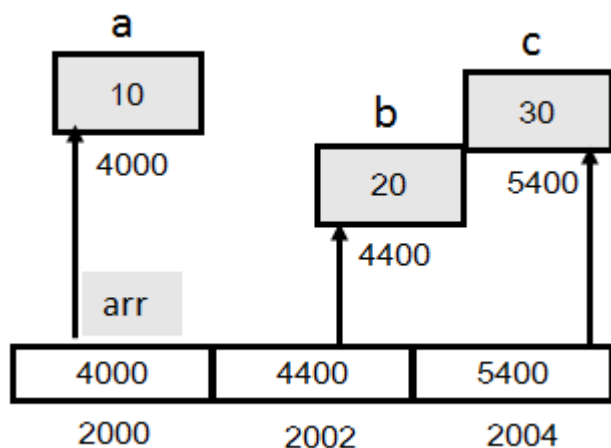
### 1.10.1. Array of Pointers:

An array of pointers is a collection of addresses. The addresses in an array of pointers could be the addresses of isolated variables or the addresses of array elements or any other addresses.

The only constraint is that all the pointers in an array must be of the same type.

#### **// Array of pointers**

```
#include<stdio.h>
void main()
{
#include<stdio.h>
void main()
{
int a=10, b=20, c=30;
int* arr[3];
clrscr();
arr[0]=&a;
```



```

arr[1]=&b;
arr[2]=&c;
printf("The values of variable are:\n");
printf("%d\t%d\t%d",a,b,c);
printf("\n the pointer variable values are %d\t%d\t%d\n",*arr[0],*arr[1],*arr[2]);
printf("\n the address are %u\t%u\t%u\n",arr[0],arr[1],arr[2]);
getch();
}

```

### Output:

```

The values of variable are:
10      20      30
the pointer variable values are 10      20      30

the address are 65524 65522 65520
C:\TURBOC3\BIN>

```

### Explanation:

- arr is an array of integer pointers and holds the addresses of variables a, b and c
- All the variables are of the same type.

### 1.10.2. Pointer to an Array

It is possible to create a pointer that points to a complete array instead of pointing to the individual elements of an array or isolated variables. Such a pointer is known as a pointer to an array.

For example,

`int (*p1)[5];` //  $\leftarrow$  p1 is a pointer to an array of 5 integers

`int (*p2)[2][2];`  $\leftarrow$  p2 is a pointer to an integer array of 2 rows and 2 columns

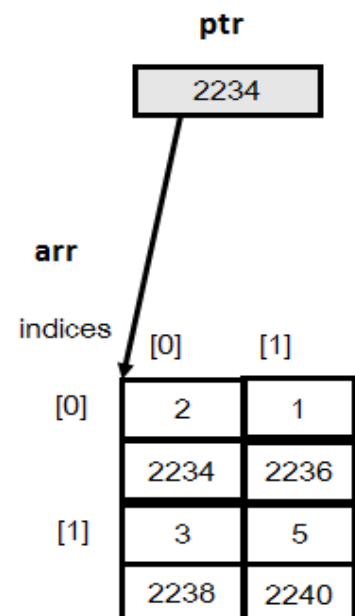
`int (*p3)[2][3][4];`  $\leftarrow$  p3 is a pointer to an integer array having 2 planes.  
Each plane has 3 rows and 4 columns

#### //pointer to an array

```

#include<stdio.h>
void main()
{
int arr[2][2]={ {2,1},{3,5}};
int (*ptr)[2]=arr;
clrscr();
printf("Address of row1 is %p\n",arr[0]);
printf("Address of row2 is %p\n",arr[1]);
printf("1st element of row1 is %d\n",arr[0][0]);
printf("1st element of row2 is %d\n",arr[1][0]);
getch();
}

```



**Output:**

Address of row1 is 234F:2234

Address of row2 is 234F:2238

1<sup>st</sup> element of row1 is 2

1<sup>st</sup> element of row2 is 3

**Explanation:**

arr refers to the address of the 1<sup>st</sup> element of the array

Elements of a 2-D array are 1-D arrays

Thus, arr refers to the address of 1<sup>st</sup> 1-D array of two integers(i.e. first row)

The type of arr is int(\*)[2]

Ptr is initialized with the starting address of row 1

Ptr+1 will point to the next row

As types of arr and ptr are same and both refer to the same address, the expression ptr[1][0] is equivalent to the expression arr[1][0]

**Pointer to a Pointer**

A pointer that holds the address of another pointer variable is known as a pointer to a pointer. Such a pointer is said to exhibit multiple levels of indirection. There can be many levels of indirection in a single declaration statement.

```
main()
{
    int i=10;
    int *p1=&i; // ← Pointer to int
    int **p2=&p1; // ← pointer to pointer to int
    int ***p3=&p2; // ← pointer to pointer to pointer to int
    int ****p4=&p3; // ← pointer to pointer to pointer to pointer to int
    int *****p5=&p4; // ← pointer to pointer to pointer to pointer to pointer to int
    printf("\n The values of variables are:\n");
    printf("%d\t",i);
    printf("%d\t",*p1);
    printf("%d\t",**p2);
    printf("%d\t",***p3);
    printf("%d\t",****p4);
    printf("%d\t",*****p5);
}
```

**OUTPUT:**

The values of variables are:

10      10      10      10      10      10

**Example Program: Sorting of names**

```
#include<stdio.h>
#include<conio.h>
void main()
{
char *t;
int i,j,k,n;
char *name[3];
clrscr();
printf("\n enter the number of elements:");
scanf("%d",&n);
printf("\n enter the names one by one:");
for(i=0;i<n;i++)
scanf("%s",name[i]);
for(i=0;i<3;i++)
{
for(j=i+1;j<3;j++)
{
k=strcmp(name[i],name[j]);
if(k>0)
{
t=name[i];
name[i]=name[j];
name[j]=t;
}
}
}
printf("ascending order is:");
for (i=0;i<3;i++)
printf("\n%s",name[i]);
getch();
}
```

**Output:**

Enter the number of elements: 4  
Enter the names one by one: Priya  
Devi  
Savitha  
Abi

Ascending order is  
Abi  
Devi  
Priya  
Savitha

**//Additional Program : To find Sum of N natural numbers Using Function**

```
#include<stdio.h>
#include<conio.h>
void sum();
void main()
{
clrscr();
sum();
getch();
}
void sum()
{
int i,sum1=0,num;
printf("Enter a number:");
scanf("%d",&num);
for(i=1;i<=num;i++)
{
sum1=sum1 + i;
}
printf("Sum of Natural Number is %d",sum1);
}
```

Enter a number:5  
Sum of Natural Number is 15

**Output:**  
Enter a number:5  
Sum of Natural Number is 15

**To find the maximum of 3 numbers using parameter passing**

```
#include<stdio.h>
#include<conio.h>
int max(int,int,int);
void main()
{
int x,y,z,big;
clrscr();
printf("Enter x,y and z values:");
scanf("%d%d%d",&x,&y,&z);
big=max(x,y,z);
printf("\nMaximum = %d",big);
getch();
}
int max(int a, int b, int c)
{
if(a>b && a>c)
return(a);
else if(b>c)
return(b);
else
return(c);}
}
```

**Output:**  
Enter x,y and z values:  
12  
34  
2  
Maximum = 34

// Write a C program using pointer to read in an array of integers and print its element in reverse order

```
include<stdio.h>
#include<conio.h>
#define MAX 30
void main()
{
    int size, i, arr[MAX];
    int *ptr;
    clrscr();
    ptr = &arr[0];
    printf("\nEnter the size of array : ");
    scanf("%d", &size);
    printf("\nEnter %d integers into array: ", size);
    for (i = 0; i < size; i++) {
        scanf("%d", ptr);
        ptr++;
    }
    ptr = &arr[size - 1];
    printf("\nElements of array in reverse order are :");
    for (i = size - 1; i >= 0; i--) {
        printf("\nElement%d is %d : ", i, *ptr);
        ptr--;
    }
    getch();
}
```

**Output:**

```
Enter the size of array : 5
Enter 5 integers into array : 11 22 33 44
55
Elements of array in reverse order are :
Element 4 is : 55
Element 4 is : 44
Element 4 is : 33
Element 4 is : 22
Element 4 is : 11
```

# ``` /*string operation without string library()*/ ```

```
#include<stdio.h>
#include<conio.h>
void slen(char *s);
void scpy(char *s,char *ss);
void scmp(char *s,char *ss);
void scat(char *s,char *ss);
void srev(char *s);
void main()
{
char s1[100],s2[100],s3[100],s4[100];
clrscr();
printf("\n Enter string1:");
gets(s1);
slen(s1);
srev(s1);
printf("\n Enter string2:");
gets(s2);
scpy(s3,s2);
printf("\n Enter the another for strcmp() and strcat() with s2:");
gets(s4);
scmp(s2,s4);
scat(s2,s4);
getch();
}
void slen(char *s) //String Length
{
int l;
for(l=0;s[l]!='\0';l++);
printf("\nLength of string (%s) is %d\n",s,l);
}
void srev(char *s) //String Reverse
{
int i,j;
char temp;
for(i=0;s[i]!='\0';i++);
i--;
for(j=0;i>j;)
{
temp=s[i];
s[i]=s[j];
s[j]=temp;
j++;
i--;
}
}
```

## Output:

Enter string1:  
Computer

Length of string ( computer) is 8

The reversed string is retupmoc

Enter string2:  
java

String copy from s2 to s3 is java

Enter the another for strcmp() and strcat() with s2 :  
program

Two strings are not equal

The concatenate(s2,s4) is javaprogram

```

printf("\nThe reversed string is %s\n",s);
}
void scpy(char *s, char *ss) //String Copy
{
int i;
for(i=0;ss[i]!='\0';i++)
s[i]=ss[i];
s[i]='\0';
printf("\nString copy from s2 to s3 is %s\n",s);
}
void scmp(char *s, char *ss) //String Compare
{
int flag,i;
for(i=0;s[i]!='\0'||ss[i]!='\0';i++)
{
if(s[i]==ss[i])
flag=1;
else
{
flag=0;
break;
}
}
if(flag==1)
printf("\n Two strings are equal\n");
else
printf("\n Two strings are not equal");
}
void scat(char *s, char *ss) //String Concatenation
{
int i,j;
for(i=0;s[i]!='\0';i++);
for(j=0;ss[j]!='\0';j++)
{
s[i]=ss[j];
i++;
}
s[i]='\0';
printf("\n The concatenate(s2,s4) is %s\n",s);
}

```



## 1.11 PARAMETER PASSING METHODS:

There are two ways by which arguments are passed in the function. They are:

1. Call by value/ Pass by value
2. Call by reference / Address or Pass by reference / Address.

### 1.11.1 Call by value / Pass by value/Passing Arguments by value:

In this method, values of actual arguments are copied to the formal parameters of the function.

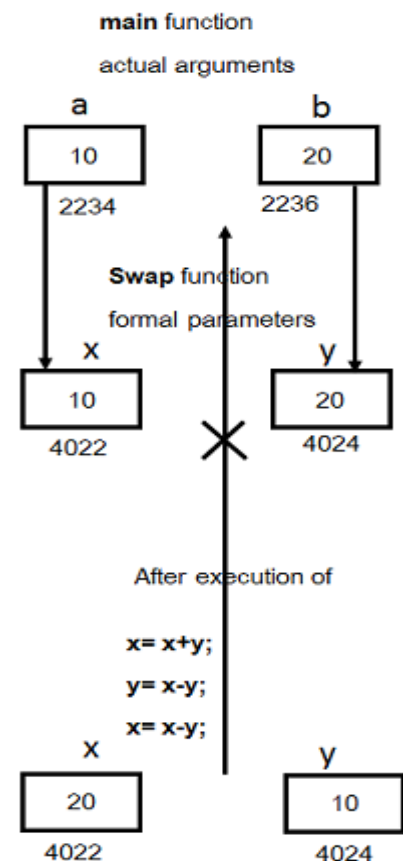
If the arguments are passed by value, the changes made in the values of formal parameters inside the called function are not reflected back to the calling function.

#### Example Program:

```
#include<stdio.h>
void swap(int,int); //Function declaraction;
void main()
{
    int a,b;
    clrscr();
    printf("Enter a and b values:");
    scanf("%d%d",&a,&b);
    printf("Before swap(), a=%d\tb=%d\n",a,b);
    swap(a,b);
    printf("After swap(), a=%d\tb=%d\n",a,b);
    getch();
}
void swap(int x,int y)
{
    x= x +y;
    y=x-y;
    x=x-y;
    printf("\nIn swap(), x=%\ty=%d",x,y);
}
```

#### OUTPUT:

```
Enter a and b values: 10    20
Before swap(), a=10    b=20
In swap(), x=20    y=10
After swap(), a=10    b=20
```



**Explanation:**

- On the execution of the function call, i.e, swap(a,b);, the values of actual arguments a and b are copied in to the formal parameters x and y
- Formal parameters are allocated at separate memory locations.
- A change made in the formal parameters is independent of the actual arguments .
- On returning from the called function, the formal parameters are destroyed and the access to the actual arguments gives values that are unchanged.

**1.11.2 Call by Reference:**

In this method, the addresses of the actual arguments are passed to the formal parameters of the function.

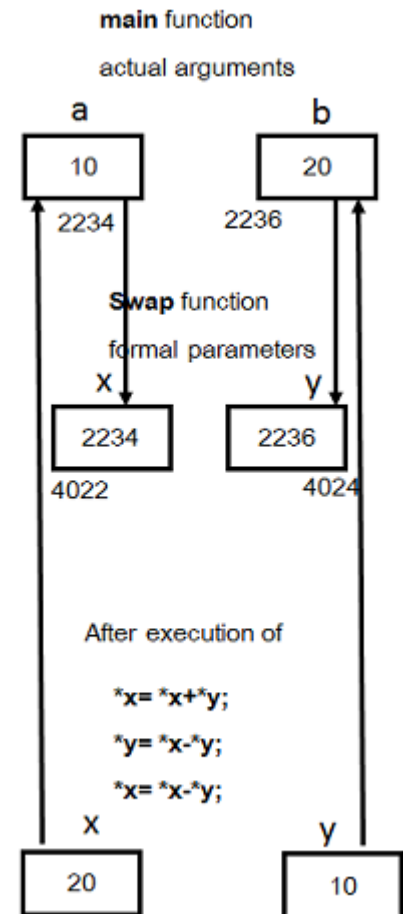
If the arguments are passed by reference , the changes made in the values pointed to by the formal parameters in the called function are reflected back to the calling function

**Example Program:**

```
#include<stdio.h>
void swap(int*,int*);
void main()
{
    int a,b;
    clrscr();
    printf("Enter a and b values:");
    scanf("%d%d",&a,&b);
    printf("Before swap(), a=%d\tb=%d\n",a,b);
    swap(&a,&b);
    printf("After swap(), a=%d\tb=%d\n",a,b);
    getch();
}
void swap(int *x,int *y)
{
    *x= *x + *y;
    *y = *x - *y;
    *x = *x - *y;
    printf("\nIn swap(), x=%d\t y=%d",*x,*y);
}
```

**OUTPUT:**

```
Enter a and b values: 10    20
Before swap(), a=10    b=20
In swap(), x=20    y=10
After swap(), a=20    b=10
```



**Explanation:**

- Addresses of the actual arguments are passed instead of their values
- Changes made in the called function are actually done in the memory locations of the actual arguments
- On returning from the called function, the formal parameters are destroyed but since the changes were made at the memory locations of the actual arguments, they can still be found there.

S.No	Call by Value	Call by Reference
1	Value of variable is passed.	Value of variable is passed.
2	Any changes made to the formal variable will not affect the actual parameter as different memories are allocated for both of them.	Whatever changes we make to the formal parameter will affect the actual parameter as a same memory is allocated for both of them.
3	A temp variable is created in the function stack which does not affect the original variable.	We generally use pointers for this.
4	New location is created , this method is slow	Existing memory location is used through its address, this method is fast