C STANDARD LIBRARY

collection of built-in functions

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

C is a general-purpose, procedural, imperative computer programming language developed in 1972 by Dennis M. Ritchie at the Bell Telephone Laboratories to develop the Unix operating system.

C is the most widely used computer language that keeps fluctuating at number one scale of popularity along with Java programming language which is also equally popular and most widely used among modern software programmers.

The C Standard Library is a set of C built-in functions, constants and header files like <assert.h>, <ctype.h>, etc. This library will work as a reference manual for C programmers.

Audience

The C Standard Library is a reference for C programmers to help them in their projects related to system programming. All the C functions have been explained in a user-friendly way and they can be copied and pasted in your C projects.

Prerequisites

A basic understanding of the C Programming language will help you in understanding the C built-in functions covered in this library.

Copyright & Disclaimer

Table of Contents

1. C Library – <assert.h>

Introduction

The **assert.h** header file of the C Standard Library provides a macro called **assert** which can be used to verify assumptions made by the program and print a diagnostic message if this assumption is false.

The defined macro **assert** refers to another macro **NDEBUG** which is not a part of <assert.h>. If NDEBUG is defined as a macro name in the source file, at the point where <assert.h> is included, the **assert** macro is defined as follows:

```
#define assert(ignore) ((void)0)
```

Library Macros

Following is the only function defined in the header assert.h:

S.N.	Function & Description
1	void assert(int expression) This is actually a macro and not a function, which can be used to add diagnostics in your C program.

void assert(int expression)

Description

The C library macro **void assert(int expression)** allows diagnostic information to be written to the standard error file. In other words, it can be used to add diagnostics in your C program.

Declaration

Following is the declaration for assert() Macro.

```
void assert(int expression);
```

Parameters

- **expression** -- This can be a variable or any C expression. If **expression** evaluates to TRUE, assert() does nothing. If **expression** evaluates to FALSE, assert() displays an error message on **stderr** (standard error stream to display error messages and diagnostics) and aborts program execution.

Return Value

This macro does not return any value.

Example

The following example shows the usage of assert() macro:

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

int main()
{
   int a;
   char str[50];

   printf("Enter an integer value: ");
   scanf("%d\n", &a);
   assert(a >= 10);
   printf("Integer entered is %d\n", a);

   printf("Enter string: ");
   scanf("%s\n", &str);
   assert(str != NULL);
   printf("String entered is: %s\n", str);

   return(0);
}
```

Let us compile and run the above program in the interactive mode as shown below:

```
Enter an integer value: 11
Integer entered is 11
Enter string: tutorialspoint
String entered is: tutorialspoint
```

2. C Library – <ctype.h>

Introduction

The **ctype.h** header file of the C Standard Library declares several functions that are useful for testing and mapping characters.

All the functions accepts **int** as a parameter, whose value must be EOF or representable as an unsigned char.

All the functions return non-zero (true) if the argument c satisfies the condition described, and zero (false) if not.

Library Functions

Following are the functions defined in the header ctype.h:

S.N.	Function & Description
1	int isalnum(int c) This function checks whether the passed character is alphanumeric.
2	int isalpha(int c) This function checks whether the passed character is alphabetic.
3	int iscntrl(int c) This function checks whether the passed character is control character.
4	int isdigit(int c) This function checks whether the passed character is decimal digit.
5	int isgraph(int c) This function checks whether the passed character has graphical representation using locale.
6	int islower(int c) This function checks whether the passed character is lowercase letter.
7	int isprint(int c) This function checks whether the passed character is printable.

8	int ispunct(int c) This function checks whether the passed character is a punctuation character.
9	int isspace(int c) This function checks whether the passed character is white-space.
10	int isupper(int c) This function checks whether the passed character is an uppercase letter.
11	int isxdigit(int c) This function checks whether the passed character is a hexadecimal digit.

int isalnum(int c)

Description

The C library function **void isalnum(int c)** checks if the passed character is alphanumeric.

Declaration

Following is the declaration for isalnum() function.

```
int isalnum(int c);
```

Parameters

- **c** -- This is the character to be checked.

Return Value

This function returns non-zero value if c is a digit or a letter, else it returns 0.

Example

The following example shows the usage of isalnum() function.

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

int main()
{
   int var1 = 'd';
   int var2 = '2';
   int var3 = '\t';
   int var4 = ' ';
```

```
   if( isalnum(var1) )
   {
      printf("var1 = |%c| is alphanumeric\n", var1 );
   }
   else
   {
      printf("var1 = |%c| is not alphanumeric\n", var1 );
   }
   if( isalnum(var2) )
   {
      printf("var2 = |%c| is alphanumeric\n", var2 );
   }
   else
   {
      printf("var2 = |%c| is not alphanumeric\n", var2 );
   }
   if( isalnum(var3) )
   {
      printf("var3 = |%c| is alphanumeric\n", var3 );
   }
   else
   {
      printf("var3 = |%c| is not alphanumeric\n", var3 );
   }
   if( isalnum(var4) )
   {
      printf("var4 = |%c| is alphanumeric\n", var4 );
   }
   else
   {
      printf("var4 = |%c| is not alphanumeric\n", var4 );
   }

   return(0);
}
```

Let us compile and run the above program to produce the following result:

```
var1 = |d| is alphanumeric
var2 = |2| is alphanumeric
var3 = | | is not alphanumeric
var4 = | | is not alphanumeric
```

int isalpha(int c)

Description

The C library function **void isalpha(int c)** checks if the passed character is alphabetic.

Declaration

Following is the declaration for isalpha() function.

```
int isalpha(int c);
```

Parameters

- **c** -- This is the character to be checked.

Return Value

This function returns non-zero value if c is an alphabet, else it returns 0.

Example

The following example shows the usage of isalpha() function.

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

int main()
{
   int var1 = 'd';
   int var2 = '2';
   int var3 = '\t';
   int var4 = ' ';

   if( isalpha(var1) )
   {
      printf("var1 = |%c| is an alphabet\n", var1 );
   }
   else
   {
      printf("var1 = |%c| is not an alphabet\n", var1 );
```

```
   }
   if( isalpha(var2) )
   {
      printf("var2 = |%c| is an alphabet\n", var2 );
   }
   else
   {
      printf("var2 = |%c| is not an alphabet\n", var2 );
   }
   if( isalpha(var3) )
   {
      printf("var3 = |%c| is an alphabet\n", var3 );
   }
   else
   {
      printf("var3 = |%c| is not an alphabet\n", var3 );
   }
   if( isalpha(var4) )
   {
      printf("var4 = |%c| is an alphabet\n", var4 );
   }
   else
   {
      printf("var4 = |%c| is not an alphabet\n", var4 );
   }

   return(0);
}
```

Let us compile and run the above program to produce the following result:

```
var1 = |d| is an alphabet
var2 = |2| is not an alphabet
var3 = | | is not an alphabet
var4 = | | is not an alphabet
```

int iscntrl(int c)

Description

The C library function **void iscntrl(int c)** checks if the passed character is a control character.

According to standard ASCII character set, control characters are between ASCII codes 0x00 (NUL), 0x1f (US), and 0x7f (DEL). Specific compiler implementations for certain platforms may define additional control characters in the extended character set (above 0x7f).

Declaration

Following is the declaration for iscntrl() function.

```
int iscntrl(int c);
```

Parameters

- **c** -- This is the character to be checked.

Return Value

This function returns non-zero value if c is a control character, else it returns 0.

Example

The following example shows the usage of iscntrl() function.

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

int main ()
{
   int i = 0, j = 0;
   char str1[] = "all \a about \t programming";
   char str2[] = "tutorials \n point";

   /* Prints string till control character \a */
   while( !iscntrl(str1[i]) )
   {
      putchar(str1[i]);
      i++;
   }

   /* Prints string till control character \n */
   while( !iscntrl(str2[j]) )
   {
      putchar(str2[j]);
      j++;
   }
```

```
   return(0);
}
```

Let us compile and run the above program to produce the following result:

```
all tutorials
```

int isdigit(int c)

Description

The C library function **void isdigit(int c)** checks if the passed character is a decimal digit character.

Decimal digits are (numbers): 0 1 2 3 4 5 6 7 8 9.

Declaration

Following is the declaration for isdigit() function.

```
int isdigit(int c);
```

Parameters

- **c** -- This is the character to be checked.

Return Value

This function returns non-zero value if c is a digit, else it returns 0.

Example

The following example shows the usage of isdigit() function.

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

int main()
{
   int var1 = 'h';
   int var2 = '2';

   if( isdigit(var1) )
   {
      printf("var1 = |%c| is a digit\n", var1 );
   }
   else
   {
      printf("var1 = |%c| is not a digit\n", var1 );
```

```
   }
   if( isdigit(var2) )
   {
      printf("var2 = |%c| is a digit\n", var2 );
   }
   else
   {
      printf("var2 = |%c| is not a digit\n", var2 );
   }

   return(0);
}
```

Let us compile and run the above program to produce the following result:

```
var1 = |h| is not a digit
var2 = |2| is a digit
```

int isgraph(int c)

Description

The C library function **void isgraph(int c)** checks if the character has graphical representation.

The characters with graphical representations are all those characters that can be printed except for whitespace characters (like ' '), which is not considered as **isgraph** characters.

Declaration

Following is the declaration for isgraph() function.

```
int isgraph(int c);
```

Parameters

- **c** -- This is the character to be checked.

Return Value

This function returns non-zero value if c has a graphical representation as character, else it returns 0.

Example

The following example shows the usage of isgraph() function.

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

```
#include <ctype.h>

int main()
{
   int var1 = '3';
   int var2 = 'm';
   int var3 = ' ';

   if( isgraph(var1) )
   {
       printf("var1 = |%c| can be printed\n", var1 );
   }
   else
   {
      printf("var1 = |%c| can't be printed\n", var1 );
   }
   if( isgraph(var2) )
   {
       printf("var2 = |%c| can be printed\n", var2 );
   }
   else
   {
      printf("var2 = |%c| can't be printed\n", var2 );
   }
   if( isgraph(var3) )
   {
       printf("var3 = |%c| can be printed\n", var3 );
   }
   else
   {
      printf("var3 = |%c| can't be printed\n", var3 );
   }
   return(0);
}
```

Let us compile and run the above program to produce the following result:

```
var1 = |3| can be printed
var2 = |m| can be printed
```

```
var3 = | | can't be printed
```

int islower(int c)

Description

The C library function **int islower(int c)** checks whether the passed character is a lowercase letter.

Declaration

Following is the declaration for islower() function.

```
int islower(int c);
```

Parameters

- **c** -- This is the character to be checked.

Return Value

This function returns a non-zero value(true) if c is a lowercase alphabetic letter else, zero (false).

Example

The following example shows the usage of islower() function.

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

int main()
{
   int var1 = 'Q';
   int var2 = 'q';
   int var3 = '3';

   if( islower(var1) )
   {
       printf("var1 = |%c| is lowercase character\n", var1 );
   }
   else
   {
      printf("var1 = |%c| is not lowercase character\n", var1 );
   }
   if( islower(var2) )
   {
```

```
        printf("var2 = |%c| is lowercase character\n", var2 );
   }
   else
   {
      printf("var2 = |%c| is not lowercase character\n", var2 );
   }
   if( islower(var3) )
   {
        printf("var3 = |%c| is lowercase character\n", var3 );
   }
   else
   {
      printf("var3 = |%c| is not lowercase character\n", var3 );
   }

   return(0);
}
```

Let us compile and run the above program to produce the following result:

```
var1 = |Q| is not lowercase character
var2 = |q| is lowercase character
var3 = |3| is not lowercase character
```

int isprint(int c)

Description

The C library function **int isprint(int c)** checks whether the passed character is printable. A printable character is a character that is not a control character.

Declaration

Following is the declaration for isprint() function.

```
int isprint(int c);
```

Parameters

- **c** -- This is the character to be checked.

Return Value

This function returns a non-zero value(true) if c is a printable character else, zero (false).

Example

The following example shows the usage of isprint() function.

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

int main()
{
   int var1 = 'k';
   int var2 = '8';
   int var3 = '\t';
   int var4 = ' ';

   if( isprint(var1) )
   {
      printf("var1 = |%c| can be printed\n", var1 );
   }
   else
   {
      printf("var1 = |%c| can't be printed\n", var1 );
   }
   if( isprint(var2) )
   {
      printf("var2 = |%c| can be printed\n", var2 );
   }
   else
   {
      printf("var2 = |%c| can't be printed\n", var2 );
   }
   if( isprint(var3) )
   {
      printf("var3 = |%c| can be printed\n", var3 );
   }
   else
   {
      printf("var3 = |%c| can't be printed\n", var3 );
   }
   if( isprint(var4) )
```

```
   {
      printf("var4 = |%c| can be printed\n", var4 );
   }
   else
   {
      printf("var4 = |%c| can't be printed\n", var4 );
   }

   return(0);
}
```

Let us compile and run the above program to produce the following result:

```
var1 = |k| can be printed
var2 = |8| can be printed
var3 = |    | can't be printed
var4 = | | can be printed
```

int ispunct(int c)

Description

The C library function **int ispunct(int c)** checks whether the passed character is a punctuation character. A punctuation character is any graphic character (as in isgraph) that is not alphanumeric (as in isalnum).

Declaration

Following is the declaration for ispunct() function.

```
int ispunct(int c);
```

Parameters

- **c** -- This is the character to be checked.

Return Value

This function returns a non-zero value(true) if c is a punctuation character else, zero (false).

Example

The following example shows the usage of ispunct() function.

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

int main()
{
   int var1 = 't';
   int var2 = '1';
   int var3 = '/';
   int var4 = ' ';

   if( ispunct(var1) )
   {
       printf("var1 = |%c| is a punctuation character\n", var1 );
   }
   else
   {
       printf("var1 = |%c| is not a punctuation character\n", var1 );
   }
   if( ispunct(var2) )
   {
       printf("var2 = |%c| is a punctuation character\n", var2 );
   }
   else
   {
       printf("var2 = |%c| is not a punctuation character\n", var2 );
   }
   if( ispunct(var3) )
   {
       printf("var3 = |%c| is a punctuation character\n", var3 );
   }
   else
   {
       printf("var3 = |%c| is not a punctuation character\n", var3 );
   }
   if( ispunct(var4) )
   {
       printf("var4 = |%c| is a punctuation character\n", var4 );
   }
```

```
   else
   {
       printf("var4 = |%c| is not a punctuation character\n", var4 );
   }

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
var1 = |t| is not a punctuation character
var2 = |1| is not a punctuation character
var3 = |/| is a punctuation character
var4 = | | is not a punctuation character
```

int isspace(int c)

Description

The C library function **int isspace(int c)** checks whether the passed character is white-space.

Standard white-space characters are:

```
' '     (0x20)    space (SPC)
'\t' (0x09) horizontal tab (TAB)
'\n' (0x0a) newline (LF)
'\v' (0x0b) vertical tab (VT)
'\f' (0x0c) feed (FF)
'\r' (0x0d) carriage return (CR)
```

Declaration

Following is the declaration for isspace() function.

```
int isspace(int c);
```

Parameters

- **c** -- This is the character to be checked.

Return Value

This function returns a non-zero value(true) if c is a white-space character else, zero (false).

Example

The following example shows the usage of isspace() function.

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

int main()
{
   int var1 = 't';
   int var2 = '1';
   int var3 = ' ';

   if( isspace(var1) )
   {
       printf("var1 = |%c| is a white-space character\n", var1 );
   }
   else
   {
       printf("var1 = |%c| is not a white-space character\n", var1 );
   }
   if( isspace(var2) )
   {
       printf("var2 = |%c| is a white-space character\n", var2 );
   }
   else
   {
       printf("var2 = |%c| is not a white-space character\n", var2 );
   }
   if( isspace(var3) )
   {
       printf("var3 = |%c| is a white-space character\n", var3 );
   }
   else
   {
       printf("var3 = |%c| is not a white-space character\n", var3 );
   }

   return(0);
```

```
}
```

Let us compile and run the above program that will produce the following result:

```
var1 = |t| is not a white-space character
var2 = |1| is not a white-space character
var3 = | | is a white-space character
```

int isupper(int c)

Description

The C library function **int isupper(int c)** checks whether the passed character is uppercase letter.

Declaration

Following is the declaration for isupper() function.

```
int isupper(int c);
```

Parameters

- **c** -- This is the character to be checked.

Return Value

This function returns a non-zero value(true) if c is an uppercase alphabetic letter else, zero (false).

Example

The following example shows the usage of isupper() function.

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

int main()
{
   int var1 = 'M';
   int var2 = 'm';
   int var3 = '3';

   if( isupper(var1) )
   {
      printf("var1 = |%c| is uppercase character\n", var1 );
   }
   else
```

```
   {
      printf("var1 = |%c| is not uppercase character\n", var1 );
   }
   if( isupper(var2) )
   {
      printf("var2 = |%c| is uppercase character\n", var2 );
   }
   else
   {
      printf("var2 = |%c| is not uppercase character\n", var2 );
   }
   if( isupper(var3) )
   {
      printf("var3 = |%c| is uppercase character\n", var3 );
   }
   else
   {
      printf("var3 = |%c| is not uppercase character\n", var3 );
   }

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
var1 = |M| is uppercase character
var2 = |m| is not uppercase character
var3 = |3| is not uppercase character
```

int isxdigit(int c)

Description

The C library function **int isxdigit(int c)** checks whether the passed character is a hexadecimal digit.

Declaration

Following is the declaration for isxdigit() function.

```
int isxdigit(int c);
```

Parameters

- **c** -- This is the character to be checked.

Return Value

This function returns a non-zero value(true) if c is a hexadecimal digit else, zero (false).

Example

The following example shows the usage of isxdigit() function.

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

int main()
{
   char var1[] = "tuts";
   char var2[] = "0xE";

   if( isxdigit(var1[0]) )
   {
     printf("var1 = |%s| is hexadecimal character\n", var1 );
   }
   else
   {
     printf("var1 = |%s| is not hexadecimal character\n", var1 );
   }

   if( isxdigit(var2[0] ))
   {
     printf("var2 = |%s| is hexadecimal character\n", var2 );
   }
   else
   {
     printf("var2 = |%s| is not hexadecimal character\n", var2 );
   }

   return(0);
}
```

Let us compile and run the above program to produce the following result:

```
var1 = |tuts| is not hexadecimal character
var2 = |0xE| is hexadecimal character
```

The library also contains two conversion functions that accepts and returns an "int".

S.N.	Function & Description
1	int tolower(int c) This function converts uppercase letters to lowercase.
2	int toupper(int c) This function converts lowercase letters to uppercase.

int tolower(int c)

Description

The C library function **int tolower(int c)** converts a given letter to lowercase.

Declaration

Following is the declaration for tolower() function.

```
int tolower(int c);
```

Parameters

- **c** -- This is the letter to be converted to lowercase.

Return Value

This function returns lowercase equivalent to c, if such value exists, else c remains unchanged. The value is returned as an **int** value that can be implicitly casted to **char**.

Example

The following example shows the usage of tolower() function.

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

int main()
{
   int i = 0;
   char c;
   char str[] = "TUTORIALS POINT";

   while( str[i] )
   {
      putchar(tolower(str[i]));
```

```
      i++;
   }

   return(0);
}
```

Let us compile and run the above program to produce the following result:

```
tutorials point
```

int toupper(int c)

Description

The C library function **int toupper(int c)** converts lowercase letter to uppercase.

Declaration

Following is the declaration for toupper() function.

```
int toupper(int c);
```

Parameters

- **c** -- This is the letter to be converted to uppercase.

Return Value

This function returns uppercase equivalent to c, if such value exists, else c remains unchanged. The value is returned as an **int** value that can be implicitly casted to **char**.

Example

The following example shows the usage of toupper() function.

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

int main()
{
   int i = 0;
   char c;
   char str[] = "Tutorials Point";

   while(str[i])
   {
      putchar (toupper(str[i]));
      i++;
```

```
   }

  return(0);
}
```

Let us compile and run the above program to produce the following result:

```
TUTORIALS POINT
```

Character Classes

S.N.	Character Class & Description
1	**Digits** This is a set of whole numbers { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }.
2	**Hexadecimal digits** This is the set of - { 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f }.
3	**Lowercase letters** This is a set of lowercase letters { a b c d e f g h i j k l m n o p q r s t u v w x y z }.
4	**Uppercase letters** This is a set of uppercase letters {A B C D E F G H I J K L M N O P Q R S T U V W X Y Z }.
5	**Letters** This is a set of lowercase and uppercase letters.
6	**Alphanumeric characters** This is a set of Digits, Lowercase letters and Uppercase letters.
7	**Punctuation characters** This is a set of ! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~
8	**Graphical characters** This is a set of Alphanumeric characters and Punctuation characters.

9	**Space characters** This is a set of tab, newline, vertical tab, form feed, carriage return, and space.
10	**Printable characters** This is a set of Alphanumeric characters, Punctuation characters and Space characters.
11	**Control characters** In ASCII, these characters have octal codes 000 through 037, and 177 (DEL).
12	**Blank characters** These are spaces and tabs.
13	**Alphabetic characters** This is a set of Lowercase letters and Uppercase letters.

3. C Library – <errno.h>

Introduction

The **errno.h** header file of the C Standard Library defines the integer variable **errno**, which is set by system calls and some library functions in the event of an error to indicate what went wrong. This macro expands to a modifiable lvalue of type int, therefore it can be both read and modified by a program.

The **errno** is set to zero at program startup. Certain functions of the standard C library modify its value to other than zero to signal some types of error. You can also modify its value or reset to zero at your convenience.

The **errno.h** header file also defines a list of macros indicating different error codes, which will expand to integer constant expressions with type **int**.

Library Macros

Following are the macros defined in the header errno.h:

S.N.	Macro & Description
1	extern int errno This is the macro set by system calls and some library functions in the event of an error to indicate what went wrong.
2	EDOM Domain Error This macro represents a domain error, which occurs if an input argument is outside the domain, over which the mathematical function is defined and errno is set to EDOM.
3	ERANGE Range Error This macro represents a range error, which occurs if an input argument is outside the range, over which the mathematical function is defined and errno is set to ERANGE.

extern int errno

Description

The C library macro **extern int errno** is set by system calls and some library functions in the event of an error to indicate if anything went wrong.

Declaration

Following is the declaration for errno macro.

```
extern int errno
```

Parameters

- NA

Return Value

- NA

Example

The following example shows the usage of errno Macro.

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

extern int errno ;

int main ()
{
   FILE *fp;

   fp = fopen("file.txt", "r");
   if( fp == NULL )
   {
      fprintf(stderr, "Value of errno: %d\n", errno);
      fprintf(stderr, "Error opening file: %s\n", strerror(errno));
   }
   else
   {
      fclose(fp);
   }

   return(0);
}
```

Let us compile and run the above program that will produce the following result in case file **file.txt** does not exist:

```
Value of errno: 2
Error opening file: No such file or directory
```

EDOM Domain Error

Description

As mentioned above, the C library macro **EDOM** represents a domain error, which occurs if an input argument is outside the domain, over which the mathematical function is defined and errno is set to EDOM.

Declaration

Following is the declaration for EDOM Macro.

```
#define EDOM some_value
```

Parameters

- NA

Return Value

- NA

Example

The following example shows the usage of EDOM Macro.

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

int main()
{
   double val;

   errno = 0;
   val = sqrt(-10);
   if(errno == EDOM)
   {
      printf("Invalid value \n");
   }
   else
   {
```

```
      printf("Valid value\n");
   }

   errno = 0;
   val = sqrt(10);
   if(errno == EDOM)
   {
      printf("Invalid value\n");
   }
   else
   {
      printf("Valid value\n");
   }

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Invalid value
Valid value
```

ERANGE Range Error

Description

As mentioned above, the C library macro **ERANGE** represents a range error, which occurs if an input argument is outside the range, over which the mathematical function is defined and errno is set to ERANGE.

Declaration

Following is the declaration for ERANGE Macro.

```
#define ERANGE some_value
```

Parameters

- NA

Return Value

- NA

Example

The following example shows the usage of ERANGE Macro.

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

int main()
{
   double x;
   double value;

   x = 1.000000;
   value = log(x);
   if( errno == ERANGE )
   {
      printf("Log(%f) is out of range\n", x);
   }
   else
   {
      printf("Log(%f) = %f\n", x, value);
   }
   x = 0.000000;
   value = log(x);
   if( errno == ERANGE )
   {
      printf("Log(%f) is out of range\n" x);
   }
   else
   {
      printf("Log(%f) = %f\n", x, value);
   }
   return 0;
}
```

Let us compile and run the above program that will produce the following result:

```
Log(1.000000) = 1.609438
Log(0.000000) is out of range
```

4. C Library – <float.h>

Introduction

The **float.h** header file of the C Standard Library contains a set of various platform-dependent constants related to floating point values. These constants are proposed by ANSI C. They allow making more portable programs. Before checking all the constants, it is good to understand that floating-point number is composed of following four elements:

Component	Component Description
S	sign (+/-)
b	base or radix of the exponent representation, 2 for binary, 10 for decimal, 16 for hexadecimal, and so on...
e	exponent, an integer between a minimum e_{min} and a maximum e_{max}.
p	precision, the number of base-b digits in the significand

Based on the above 4 components, a floating point will have its value as follows:

```
floating-point = ( S ) p x b^e

or

floating-point = (+/-) precision x base^exponent
```

Library Macros

The following values are implementation-specific and defined with the #define directive, but these values may not be any lower than what is given here. Note that in all instances FLT refers to type **float**, DBL refers to **double**, and LDBL refers to **long double**.

Macro	Description
FLT_ROUNDS	Defines the rounding mode for floating point addition and it can have any of the following values: -1 - indeterminable 0 - towards zero

	1 - to nearest 2 - towards positive infinity 3 - towards negative infinity
FLT_RADIX 2	This defines the base radix representation of the exponent. A base-2 is binary, base-10 is the normal decimal representation, base-16 is Hex.
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	These macros define the number of digits in the number (in the FLT_RADIX base).
FLT_DIG 6 DBL_DIG 10 LDBL_DIG 10	These macros define the maximum number decimal digits (base-10) that can be represented without change after rounding.
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	These macros define the minimum negative integer value for an exponent in base FLT_RADIX.
FLT_MIN_10_EXP -37 DBL_MIN_10_EXP -37 LDBL_MIN_10_EXP -37	These macros define the minimum negative integer value for an exponent in base 10.
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	These macros define the maximum integer value for an exponent in base FLT_RADIX.
FLT_MAX_10_EXP +37 DBL_MAX_10_EXP +37 LDBL_MAX_10_EXP +37	These macros define the maximum integer value for an exponent in base 10.
FLT_MAX 1E+37 DBL_MAX 1E+37 LDBL_MAX 1E+37	These macros define the maximum finite floating-point value.
FLT_EPSILON 1E-5	These macros define the least significant digit

DBL_EPSILON 1E-9 LDBL_EPSILON 1E-9	representable.
FLT_MIN 1E-37 DBL_MIN 1E-37 LDBL_MIN 1E-37	These macros define the minimum floating-point values.

Example

The following example shows the usage of few of the constants defined in float.h file.

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

int main()
{
   printf("The maximum value of float = %.10e\n", FLT_MAX);
   printf("The minimum value of float = %.10e\n", FLT_MIN);

   printf("The number of digits in the number = %.10e\n", FLT_MANT_DIG);
}
```

Let us compile and run the above program that will produce the following result:

```
The maximum value of float = 3.4028234664e+38
The minimum value of float = 1.1754943508e-38
The number of digits in the number = 7.2996655210e-312
```

5. C Library – <limits.h>

Introduction

The **limits.h** header determines various properties of the various variable types. The macros defined in this header, limits the values of various variable types like char, int and long.

These limits specify that a variable cannot store any value beyond these limits, for example an unsigned character can store up to a maximum value of 255.

Library Macros

The following values are implementation-specific and defined with the #define directive, but these values may not be any lower than what is given here.

Macro	Value	Description
CHAR_BIT	8	Defines the number of bits in a byte.
SCHAR_MIN	-128	Defines the minimum value for a signed char.
SCHAR_MAX	127	Defines the maximum value for a signed char.
UCHAR_MAX	255	Defines the maximum value for an unsigned char.
CHAR_MIN	0	Defines the minimum value for type char and its value will be equal to SCHAR_MIN if char represents negative values, otherwise zero.
CHAR_MAX	127	Defines the value for type char and its value will be equal to SCHAR_MAX if char represents negative values, otherwise UCHAR_MAX.
MB_LEN_MAX	1	Defines the maximum number of bytes in a multi-byte character.
SHRT_MIN	-32768	Defines the minimum value for a short int.

SHRT_MAX	+32767	Defines the maximum value for a short int.
USHRT_MAX	65535	Defines the maximum value for an unsigned short int.
INT_MIN	-32768	Defines the minimum value for an int.
INT_MAX	+32767	Defines the maximum value for an int.
UINT_MAX	65535	Defines the maximum value for an unsigned int.
LONG_MIN	-2147483648	Defines the minimum value for a long int.
LONG_MAX	+2147483647	Defines the maximum value for a long int.
ULONG_MAX	4294967295	Defines the maximum value for an unsigned long int.

Example

The following example shows the usage of few of the constants defined in limit.h file.

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

int main()
{

   printf("The number of bits in a byte %d\n", CHAR_BIT);

   printf("The minimum value of SIGNED CHAR = %d\n", SCHAR_MIN);
   printf("The maximum value of SIGNED CHAR = %d\n", SCHAR_MAX);
   printf("The maximum value of UNSIGNED CHAR = %d\n", UCHAR_MAX);

   printf("The minimum value of SHORT INT = %d\n", SHRT_MIN);
   printf("The maximum value of SHORT INT = %d\n", SHRT_MAX);

   printf("The minimum value of INT = %d\n", INT_MIN);
   printf("The maximum value of INT = %d\n", INT_MAX);
```

```
   printf("The minimum value of CHAR = %d\n", CHAR_MIN);
   printf("The maximum value of CHAR = %d\n", CHAR_MAX);

   printf("The minimum value of LONG = %ld\n", LONG_MIN);
   printf("The maximum value of LONG = %ld\n", LONG_MAX);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
The number of bits in a byte 8
The minimum value of SIGNED CHAR = -128
The maximum value of SIGNED CHAR = 127
The maximum value of UNSIGNED CHAR = 255
The minimum value of SHORT INT = -32768
The maximum value of SHORT INT = 32767
The minimum value of INT = -32768
The maximum value of INT = 32767
The minimum value of CHAR = -128
The maximum value of CHAR = 127
The minimum value of LONG = -2147483648
The maximum value of LONG = 2147483647
```

6. C Library – <locale.h>

Introduction

The **locale.h** header defines the location specific settings, such as date formats and currency symbols. You will find several macros defined along with an important structure **struct lconv** and two important functions listed below.

Library Macros

Following are the macros defined in the header and these macros will be used in two functions listed below:

S.N.	Macro & Description
1	**LC_ALL** Sets everything.
2	**LC_COLLATE** Affects strcoll and strxfrm functions.
3	**LC_CTYPE** Affects all character functions.
4	**LC_MONETARY** Affects the monetary information provided by localeconv function.
5	**LC_NUMERIC** Affects decimal-point formatting and the information provided by localeconv function.
6	**LC_TIME** Affects the strftime function.

Library Functions

Following are the functions defined in the header locale.h:

S.N.	Function & Description
1	char *setlocale(int category, const char *locale) Sets or reads location dependent information.
2	struct lconv *localeconv(void) Sets or reads location dependent information.

char *setlocale(int category, const char *locale)

Description

The C library function **char *setlocale(int category, const char *locale)** sets or reads location dependent information.

Declaration

Following is the declaration for setlocale() function.

```
char *setlocale(int category, const char *locale)
```

Parameters

- **category** -- This is a named constant specifying the category of the functions affected by the locale setting.
  - **LC_ALL** for all of the below.
  - **LC_COLLATE** for string comparison. See strcoll().
  - **LC_CTYPE** for character classification and conversion. For example: strtoupper().
  - **LC_MONETARY** for monetary formatting for localeconv().
  - **LC_NUMERIC** for decimal separator for localeconv().
  - **LC_TIME** for date and time formatting with strftime().
  - **LC_MESSAGES** for system responses.
- **locale** -- If locale is NULL or the empty string "", the locale names will be set from the values of environment variables with the same names as the above categories.

Return Value

A successful call to setlocale() returns an opaque string that corresponds to the locale set. The return value is NULL if the request cannot be honored.

Example

The following example shows the usage of setlocale() function.

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

int main ()
{
   time_t currtime;
   struct tm *timer;
   char buffer[80];

   time( &currtime );
   timer = localtime( &currtime );

   printf("Locale is: %s\n", setlocale(LC_ALL, "en_GB"));
   strftime(buffer,80,"%c", timer );
   printf("Date is: %s\n", buffer);


   printf("Locale is: %s\n", setlocale(LC_ALL, "de_DE"));
   strftime(buffer,80,"%c", timer );
   printf("Date is: %s\n", buffer);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Locale is: en_GB
Date is: Thu 23 Aug 2012 06:39:32 MST
Locale is: de_DE
Date is: Do 23 Aug 2012 06:39:32 MST
```

struct lconv *localeconv(void)

Description

The C library function **struct lconv *localeconv(void)** sets or reads location dependent information. These are returned in an object of the **lconv** structure type.

Declaration

Following is the declaration for localeconv() function.

```
struct lconv *localeconv(void)
```

Parameters

- NA

Return Value

This function returns a pointer to a **struct lconv** for the current locale, which has the following structure:

```
typedef struct {
   char *decimal_point;
   char *thousands_sep;
   char *grouping;
   char *int_curr_symbol;
   char *currency_symbol;
   char *mon_decimal_point;
   char *mon_thousands_sep;
   char *mon_grouping;
   char *positive_sign;
   char *negative_sign;
   char int_frac_digits;
   char frac_digits;
   char p_cs_precedes;
   char p_sep_by_space;
   char n_cs_precedes;
   char n_sep_by_space;
   char p_sign_posn;
   char n_sign_posn;
} lconv
```

Example

The following example shows the usage of localeconv() function.

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

int main ()
{
   struct lconv * lc;

   setlocale(LC_MONETARY, "it_IT");
   lc = localeconv();
   printf("Local Currency Symbol: %s\n",lc->currency_symbol);
   printf("International Currency Symbol: %s\n",lc->int_curr_symbol);

   setlocale(LC_MONETARY, "en_US");
   lc = localeconv();
   printf("Local Currency Symbol: %s\n",lc->currency_symbol);
   printf("International Currency Symbol: %s\n",lc->int_curr_symbol);

   setlocale(LC_MONETARY, "en_GB");
   lc = localeconv();
   printf ("Local Currency Symbol: %s\n",lc->currency_symbol);
   printf ("International Currency Symbol: %s\n",lc->int_curr_symbol);
   printf("Decimal Point = %s\n", lc->decimal_point);

   return 0;
}
```

Let us compile and run the above program that will produce the following result:

```
Local Currency Symbol: EUR
International Currency Symbol: EUR
Local Currency Symbol: $
International Currency Symbol: USD
Local Currency Symbol: £
International Currency Symbol: GBP
Decimal Point = .
```

Library Structure

```
typedef struct {
   char *decimal_point;
   char *thousands_sep;
   char *grouping;
   char *int_curr_symbol;
   char *currency_symbol;
   char *mon_decimal_point;
   char *mon_thousands_sep;
   char *mon_grouping;
   char *positive_sign;
   char *negative_sign;
   char int_frac_digits;
   char frac_digits;
   char p_cs_precedes;
   char p_sep_by_space;
   char n_cs_precedes;
   char n_sep_by_space;
   char p_sign_posn;
   char n_sign_posn;
} lconv
```

Following is the description of each of the fields:

S.N.	Field & Description
1	**decimal_point** Decimal point character used for non-monetary values.
2	**thousands_sep** Thousands place separator character used for non-monetary values.
3	**grouping** A string that indicates the size of each group of digits in non-monetary quantities. Each character represents an integer value, which designates the number of digits in the current group. A value of 0 means that the previous value is to be used for the rest of the groups.
4	**int_curr_symbol**

	It is a string of the international currency symbols used. The first three characters are those specified by ISO 4217:1987 and the fourth is the character, which separates the currency symbol from the monetary quantity.
5	**currency_symbol** The local symbol used for currency.
6	**mon_decimal_point** The decimal point character used for monetary values.
7	**mon_thousands_sep** The thousands place grouping character used for monetary values.
8	**mon_grouping** A string whose elements defines the size of the grouping of digits in monetary values. Each character represents an integer value which designates the number of digits in the current group. A value of 0 means that the previous value is to be used for the rest of the groups.
9	**positive_sign** The character used for positive monetary values.
10	**negative_sign** The character used for negative monetary values.
11	**int_frac_digits** Number of digits to show after the decimal point in international monetary values.
12	**frac_digits** Number of digits to show after the decimal point in monetary values.
13	**p_cs_precedes** If equals to 1, then the currency_symbol appears before a positive monetary value. If equals to 0, then the currency_symbol appears after a positive monetary value.
14	**p_sep_by_space** If equals to 1, then the currency_symbol is separated by a space from a positive monetary value. If equals to 0, then there is no space between the

	currency_symbol and a positive monetary value.
15	**n_cs_precedes** If equals to 1, then the currency_symbol precedes a negative monetary value. If equals to 0, then the currency_symbol succeeds a negative monetary value.
16	**n_sep_by_space** If equals to 1, then the currency_symbol is separated by a space from a negative monetary value. If equals to 0, then there is no space between the currency_symbol and a negative monetary value.
17	**p_sign_posn** Represents the position of the positive_sign in a positive monetary value.
18	**n_sign_posn** Represents the position of the negative_sign in a negative monetary value.

The following values are used for **p_sign_posn** and **n_sign_posn**:

Value	Description
0	Parentheses encapsulates the value and the currency_symbol.
1	The sign precedes the value and currency_symbol.
2	The sign succeeds the value and currency_symbol.
3	The sign immediately precedes the value and currency_symbol.
4	The sign immediately succeeds the value and currency_symbol.

7. C Library – <math.h>

Introduction

The **math.h** header defines various mathematical functions and one macro. All the functions available in this library take **double** as an argument and return **double** as the result.

Library Macros

There is only one macro defined in this library:

S.N.	Macro & Description
1	**HUGE_VAL** This macro is used when the result of a function may not be representable as a floating point number. If magnitude of the correct result is too large to be represented, the function sets errno to ERANGE to indicate a range error, and returns a particular, very large value named by the macro HUGE_VAL or its negation (- HUGE_VAL). If the magnitude of the result is too small, a value of zero is returned instead. In this case, errno might or might not be set to ERANGE.

Library Functions

Following are the functions defined in the header math.h:

S.N.	Function & Description
1	double acos(double x) Returns the arc cosine of x in radians.
2	double asin(double x) Returns the arc sine of x in radians.
3	double atan(double x) Returns the arc tangent of x in radians.
4	double atan2(doubly y, double x) Returns the arc tangent in radians of y/x based on the signs of both values to

	determine the correct quadrant.
5	double cos(double x) Returns the cosine of a radian angle x.
6	double cosh(double x) Returns the hyperbolic cosine of x.
7	double sin(double x) Returns the sine of a radian angle x.
8	double sinh(double x) Returns the hyperbolic sine of x.
9	double tanh(double x) Returns the hyperbolic tangent of x.
10	double exp(double x) Returns the value of **e** raised to the xth power.
11	double frexp(double x, int *exponent) The returned value is the mantissa and the integer pointed to by exponent is the exponent. The resultant value is x = mantissa * 2 ^ exponent.
12	double ldexp(double x, int exponent) Returns **x** multiplied by 2 raised to the power of exponent.
13	double log(double x) Returns the natural logarithm (base-e logarithm) of **x**.
14	double log10(double x) Returns the common logarithm (base-10 logarithm) of **x**.
15	double modf(double x, double *integer) The returned value is the fraction component (part after the decimal), and sets integer to the integer component.
16	double pow(double x, double y)

	Returns **x** raised to the power of **y**.
17	double sqrt(double x) Returns the square root of **x**.
18	double ceil(double x) Returns the smallest integer value greater than or equal to **x**.
19	double fabs(double x) Returns the absolute value of **x**.
20	double floor(double x) Returns the largest integer value less than or equal to **x**.
21	double fmod(double x, double y) Returns the remainder of **x** divided by **y**.

double acos(double x)

Description

The C library function **double acos(double x)** returns the arc cosine of **x** in radians.

Declaration

Following is the declaration for acos() function.

```
double acos(double x)
```

Parameters

- **x** -- This is the floating point value in the interval [-1, +1].

Return Value

This function returns principal arc cosine of x, in the interval [0, pi] radians.

Example

The following example shows the usage of acos() function.

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

#define PI 3.14159265
```

```
int main ()
{
   double x, ret, val;

   x = 0.9;
   val = 180.0 / PI;

   ret = acos(x) * val;
   printf("The arc cosine of %lf is %lf degrees", x, ret);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
The arc cosine of 0.900000 is 25.855040 degrees
```

double asin(double x)

Description

The C library function **double asin(double x)** returns the arc sine of **x** in radians.

Declaration

Following is the declaration for asin() function.

```
double asin(double x)
```

Parameters

- **x** -- This is the floating point value in the interval [-1,+1].

Return Value

This function returns the arc sine of x, in the interval [-pi/2,+pi/2] radians.

Example

The following example shows the usage of asin() function.

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

#define PI 3.14159265

int main ()
{
```

```
   double x, ret, val;
   x = 0.9;
   val = 180.0 / PI;

   ret = asin(x) * val;
   printf("The arc sine of %lf is %lf degrees", x, ret);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
The arc sine of 0.900000 is 64.190609 degrees
```

double atan(double x)

Description

The C library function **double atan(double x)** returns the arc tangent of **x** in radians.

Declaration

Following is the declaration for atan() function.

```
double atan(double x)
```

Parameters

- **x** -- This is the floating point value.

Return Value

This function returns the principal arc tangent of x, in the interval [-pi/2,+pi/2] radians.

Example

The following example shows the usage of atan() function.

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

#define PI 3.14159265

int main ()
{
   double x, ret, val;
   x = 1.0;
   val = 180.0 / PI;
```

```
   ret = atan (x) * val;
   printf("The arc tangent of %lf is %lf degrees", x, ret);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
The arc tangent of 1.000000 is 45.000000 degrees
```

double atan2(doubly y, double x)

Description

The C library function **double atan2(doubly y, double x)** returns the arc tangent in radians of **y/x** based on the signs of both values to determine the correct quadrant.

Declaration

Following is the declaration for atan2() function.

```
double atan2(doubly y, double x)
```

Parameters

- **x** -- This is the floating point value representing an x-coordinate.
- **y** -- This is the floating point value representing a y-coordinate.

Return Value

This function returns the principal arc tangent of y/x, in the interval [-pi,+pi] radians.

Example

The following example shows the usage of atan2() function.

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

#define PI 3.14159265

int main ()
{
   double x, y, ret, val;

   x = -7.0;
   y = 7.0;
```

```
   val = 180.0 / PI;

   ret = atan2 (y,x) * val;
   printf("The arc tangent of x = %lf, y = %lf ", x, y);
   printf("is %lf degrees\n", ret);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
The arc tangent of x = -7.000000, y = 7.000000 is 135.000000 degrees
```

double cos(double x)

Description

The C library function **double cos(double x)** returns the cosine of a radian angle **x**.

Declaration

Following is the declaration for cos() function.

```
double cos(double x)
```

Parameters

- **x** -- This is the floating point value representing an angle expressed in radians.

Return Value

This function returns the cosine of x.

Example

The following example shows the usage of cos() function.

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

#define PI 3.14159265

int main ()
{
   double x, ret, val;

   x = 60.0;
   val = PI / 180.0;
```

```
   ret = cos( x*val );
   printf("The cosine of %lf is %lf degrees\n", x, ret);

   x = 90.0;
   val = PI / 180.0;
   ret = cos( x*val );
   printf("The cosine of %lf is %lf degrees\n", x, ret);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
The cosine of 60.000000 is 0.500000 degrees
The cosine of 90.000000 is 0.000000 degrees
```

double cosh(double x)

Description

The C library function **double cosh(double x)** returns the hyperbolic cosine of **x**.

Declaration

Following is the declaration for cosh() function.

```
double cosh(double x)
```

Parameters

- **x** -- This is the floating point value.

Return Value

This function returns hyperbolic cosine of x.

Example

The following example shows the usage of cosh() function.

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

int main ()
{
   double x;

   x = 0.5;
```

```
   printf("The hyperbolic cosine of %lf is %lf\n", x, cosh(x));

   x = 1.0;
   printf("The hyperbolic cosine of %lf is %lf\n", x, cosh(x));

   x = 1.5;
   printf("The hyperbolic cosine of %lf is %lf\n", x, cosh(x));

   return(0);
}
```

Let us compile and run the above program to produce the following result:

```
The hyperbolic cosine of 0.500000 is 1.127626
The hyperbolic cosine of 1.000000 is 1.543081
The hyperbolic cosine of 1.500000 is 2.352410
```

double sin(double x)

Description

The C library function **double sin(double x)** returns the sine of a radian angle **x**.

Declaration

Following is the declaration for sin() function.

```
double sin(double x)
```

Parameters

- **x** -- This is the floating point value representing an angle expressed in radians.

Return Value

This function returns sine of x.

Example

The following example shows the usage of sin() function.

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

```
#define PI 3.14159265

int main ()
{
   double x, ret, val;

   x = 45.0;
   val = PI / 180;
   ret = sin(x*val);
   printf("The sine of %lf is %lf degrees", x, ret);

   return(0);
}
```

Let us compile and run the above program to produce the following result:

```
The sine of 45.000000 is 0.707107 degrees
```

double sinh(double x)

Description

The C library function **double sinh(double x)** returns the hyperbolic sine of **x**.

Declaration

Following is the declaration for sinh() function.

```
double sinh(double x)
```

Parameters

- **x** -- This is the floating point value.

Return Value

This function returns hyperbolic sine of x.

Example

The following example shows the usage of sinh() function.

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

int main ()
{
```

```
   double x, ret;
   x = 0.5;

   ret = sinh(x);
   printf("The hyperbolic sine of %lf is %lf degrees", x, ret);

   return(0);
}
```

Let us compile and run the above program, this will produce the following result:

```
The hyperbolic sine of 0.500000 is 0.521095 degrees
```

double tanh(double x)

Description

The C library function **double tanh(double x)** returns the hyperbolic tangent of **x**.

Declaration

Following is the declaration for tanh() function.

```
double tanh(double x)
```

Parameters

- **x** -- This is the floating point value.

Return Value

This function returns hyperbolic tangent of x.

Example

The following example shows the usage of tanh() function.

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

int main ()
{
   double x, ret;
   x = 0.5;

   ret = tanh(x);
   printf("The hyperbolic tangent of %lf is %lf degrees", x, ret);
```

```
   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
The hyperbolic tangent of 0.500000 is 0.462117 degrees
```

double exp(double x)

Description

The C library function **double exp(double x)** returns the value of **e** raised to the **xth** power.

Declaration

Following is the declaration for exp() function.

```
double exp(double x)
```

Parameters

- **x** -- This is the floating point value.

Return Value

This function returns the exponential value of x.

Example

The following example shows the usage of exp() function.

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

int main ()
{
   double x = 0;

   printf("The exponential value of %lf is %lf\n", x, exp(x));
   printf("The exponential value of %lf is %lf\n", x+1, exp(x+1));
   printf("The exponential value of %lf is %lf\n", x+2, exp(x+2));

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
The exponential value of 0.000000 is 1.000000
```

```
The exponential value of 1.000000 is 2.718282
The exponential value of 2.000000 is 7.389056
```

double frexp(double x, int *exponent)

Description

The C library function **double frexp(double x, int *exponent)** return value is the mantissa, and the integer pointed to by **exponent** is the exponent. The resultant value is **x = mantissa * 2 ^ exponent**.

Declaration

Following is the declaration for frexp() function.

```
double frexp(double x, int *exponent)
```

Parameters

- **x** -- This is the floating point value to be computed.
- **exponent** -- This is the pointer to an **int** object where the value of the exponent is to be stored.

Return Value

This function returns the normalized fraction. If the argument x is not zero, the normalized fraction is **x** times a power of two, and its absolute value is always in the range 1/2 (inclusive) to 1 (exclusive). If **x** is zero, then the normalized fraction is zero and zero is stored in exp.

Example

The following example shows the usage of frexp() function.

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

int main ()
{
   double x = 1024, fraction;
   int e;

   fraction = frexp(x, &e);
   printf("x = %.2lf = %.2lf * 2^%d\n", x, fraction, e);

   return(0);
}
```

Let us compile and run the above program to produce the following result:

```
x = 1024.00 = 0.50 * 2^11
```

double ldexp(double x, int exponent)

Description

The C library function **double ldexp(double x, int exponent)** returns **x** multiplied by 2 raised to the power of **exponent**.

Declaration

Following is the declaration for ldexp() function.

```
double ldexp(double x, int exponent)
```

Parameters

- **x** -- This is the floating point value representing the significand.
- **exponent** -- This is the value of the exponent.

Return Value

This function returns x * 2 ^{exp}

Example

The following example shows the usage of ldexp() function.

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

int main ()
{
   double x, ret;
   int n;

   x = 0.65;
   n = 3;
   ret = ldexp(x ,n);
   printf("%f * 2^%d = %f\n", x, n, ret);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
0.650000 * 2^3 = 5.200000
```

double log(double x)

Description

The C library function **double log(double x)** returns the natural logarithm (base-e logarithm) of **x**.

Declaration

Following is the declaration for log() function.

```
double log(double x)
```

Parameters

- **x** -- This is the floating point value.

Return Value

This function returns natural logarithm of x.

Example

The following example shows the usage of log() function.

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

int main ()
{
   double x, ret;
   x = 2.7;

   /* finding log(2.7) */
   ret = log(x);
   printf("log(%lf) = %lf", x, ret);
   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
log(2.700000) = 0.993252
```

double log10(double x)

Description

The C library function **double log10(double x)** returns the common logarithm (base-10 logarithm) of **x**.

Declaration

Following is the declaration for log10() function.

```
double log10(double x)
```

Parameters

- **x** -- This is the floating point value.

Return Value

This function returns the common logarithm of x, for values of x greater than zero.

Example

The following example shows the usage of log10() function.

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

int main ()
{
   double x, ret;
   x = 10000;

   /* finding value of log1010000 */
   ret = log10(x);
   printf("log10(%lf) = %lf\n", x, ret);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
log10(10000.000000) = 4.000000
```

double modf(double x, double *integer)

Description

The C library function **double modf(double x, double *integer)** returns the fraction component (part after the decimal), and sets integer to the integer component.

Declaration

Following is the declaration for modf() function.

```
double modf(double x, double *integer)
```

Parameters

- **x** -- This is the floating point value.
- **integer** -- This is the pointer to an object where the integral part is to be stored.

Return Value
This function returns the fractional part of x, with the same sign.

Example
The following example shows the usage of modf() function.

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

int main ()
{
   double x, fractpart, intpart;

   x = 8.123456;
   fractpart = modf(x, &intpart);

   printf("Integral part = %lf\n", intpart);
   printf("Fraction Part = %lf \n", fractpart);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Integral part = 8.000000
Fraction Part = 0.123456
```

double pow(double x, double y)

Description
The C library function **double pow(double x, double y)** returns **x** raised to the power of **y** i.e. x^y.

Declaration
Following is the declaration for pow() function.

```
double pow(double x, double y)
```

Parameters
- **x** -- This is the floating point base value.

- **y** -- This is the floating point power value.

Return Value
This function returns the result of raising **x** to the power **y**.

Example
The following example shows the usage of pow() function.

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

int main ()
{
   printf("Value 8.0 ^ 3 = %lf\n", pow(8.0, 3));

   printf("Value 3.05 ^ 1.98 = %lf", pow(3.05, 1.98));

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Value 8.0 ^ 3 = 512.000000
Value 3.05 ^ 1.98 = 9.097324
```

double sqrt(double x)

Description
The C library function **double sqrt(double x)** returns the square root of **x**.

Declaration
Following is the declaration for sqrt() function.

```
double sqrt(double x)
```

Parameters
- **x** -- This is the floating point value.

Return Value
This function returns the square root of x.

Example
The following example shows the usage of sqrt() function.

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

int main ()
{

   printf("Square root of %lf is %lf\n", 4.0, sqrt(4.0) );
   printf("Square root of %lf is %lf\n", 5.0, sqrt(5.0) );

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Square root of 4.000000 is 2.000000
Square root of 5.000000 is 2.236068
```

double ceil(double x)

Description

The C library function **double ceil(double x)** returns the smallest integer value greater than or equal to **x**.

Declaration

Following is the declaration for ceil() function.

```
double ceil(double x)
```

Parameters

- **x** -- This is the floating point value.

Return Value

This function returns the smallest integral value not less than **x**.

Example

The following example shows the usage of ceil() function.

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

int main ()
{
   float val1, val2, val3, val4;
```

```
   val1 = 1.6;
   val2 = 1.2;
   val3 = 2.8;
   val4 = 2.3;

   printf ("value1 = %.1lf\n", ceil(val1));
   printf ("value2 = %.1lf\n", ceil(val2));
   printf ("value3 = %.1lf\n", ceil(val3));
   printf ("value4 = %.1lf\n", ceil(val4));

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
value1 = 2.0
value2 = 2.0
value3 = 3.0
value4 = 3.0
```

double fabs(double x)

Description

The C library function **double fabs(double x)** returns the absolute value of **x**.

Declaration

Following is the declaration for fabs() function.

```
double fabs(double x)
```

Parameters

- **x** -- This is the floating point value.

Return Value

This function returns the absolute value of x.

Example

The following example shows the usage of fabs() function.

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

```
int main ()
{
   int a, b;
   a = 1234;
   b = -344;

   printf("The absolute value of %d is %lf\n", a, fabs(a));
   printf("The absolute value of %d is %lf\n", b, fabs(b));

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
The absolute value of 1234 is 1234.000000
The absolute value of -344 is 344.000000
```

double floor(double x)

Description

The C library function **double floor(double x)** returns the largest integer value less than or equal to **x**.

Declaration

Following is the declaration for floor() function.

```
double floor(double x)
```

Parameters

- **x** -- This is the floating point value.

Return Value

This function returns the largest integral value not greater than **x**.

Example

The following example shows the usage of floor() function.

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

int main ()
{
```

```
   float val1, val2, val3, val4;

   val1 = 1.6;
   val2 = 1.2;
   val3 = 2.8;
   val4 = 2.3;

   printf("Value1 = %.1lf\n", floor(val1));
   printf("Value2 = %.1lf\n", floor(val2));
   printf("Value3 = %.1lf\n", floor(val3));
   printf("Value4 = %.1lf\n", floor(val4));

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Value1 = 1.0
Value2 = 1.0
Value3 = 2.0
Value4 = 2.0
```

double fmod(double x, double y)

Description

The C library function **double fmod(double x, double y)** returns the remainder of **x** divided by **y**.

Declaration

Following is the declaration for fmod() function.

```
double fmod(double x, double y)
```

Parameters

- **x** -- This is the floating point value with the division numerator i.e. x.
- **y** -- This is the floating point value with the division denominator i.e. y.

Return Value

This function returns the remainder of dividing x/y.

Example

The following example shows the usage of fmod() function.

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

int main ()
{
   float a, b;
   int c;
   a = 9.2;
   b = 3.7;
   c = 2;
   printf("Remainder of %f / %d is %lf\n", a, c, fmod(a,c));
   printf("Remainder of %f / %f is %lf\n", a, b, fmod(a,b));

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Remainder of 9.200000 / 2 is 1.200000
Remainder of 9.200000 / 3.700000 is 1.800000
```

8. C Library – <setjmp.h>

Introduction

The **setjmp.h** header defines the macro **setjmp()**, one function **longjmp()**, and one variable type **jmp_buf**, for bypassing the normal function call and return discipline.

Library Variables

Following is the variable type defined in the header setjmp.h:

S.N.	Variable & Description
1	**jmp_buf** This is an array type used for holding information for macro **setjmp()** and function **longjmp()**.

Library Macros

There is only one macro defined in this library:

S.N.	Macro & Description
1	**int setjmp(jmp_buf environment)** This macro saves the current *environment* into the variable **environment** for later use by the function **longjmp()**. If this macro returns directly from the macro invocation, it returns zero but if it returns from a **longjmp()** function call, then a non-zero value is returned.

int setjmp(jmp_buf environment)

Description

The C library macro **int setjmp(jmp_buf environment)**, saves the current **environment** into the variable environment for later use by the function **longjmp()**. If this macro returns directly from the macro invocation, it returns zero but if it returns from a **longjmp()** function call, then it returns the value passed to longjmp as a second argument.

Declaration

Following is the declaration for setjmp() macro.

```
int setjmp(jmp_buf environment)
```

Parameters

- **environment** -- This is the object of type jmp_buf where the environment information is stored.

Return Value

This macro may return more than once. First time, on its direct invocation, it always returns zero. When longjmp is called with the information set to the environment, the macro returns again; now it returns the value passed to longjmp as second argument.

Example

The following example shows the usage of setjmp() macro.

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

int main()
{
   int val;
   jmp_buf env_buffer;

   /* save calling environment for longjmp */
   val = setjmp( env_buffer );
   if( val != 0 )
   {
      printf("Returned from a longjmp() with value = %s\n", val);
      exit(0);
   }
   printf("Jump function call\n");
   jmpfunction( env_buffer );

   return(0);
}
void jmpfunction(jmp_buf env_buf)
{
   longjmp(env_buf, "tutorialspoint.com");
}
```

Let us compile and run the above program, this will produce the following result:

```
Jump function call
Returned from a longjmp() with value = tutorialspoint.com
```

Library Functions

Following is the only one function defined in the header setjmp.h:

S.N.	Function & Description
1	**void longjmp(jmp_buf environment, int value)** This function restores the environment saved by the most recent call to **setjmp()** macro in the same invocation of the program with the corresponding **jmp_buf** argument.

void longjmp(jmp_buf environment, int value)

Description

The C library function **void longjmp(jmp_buf environment, int value)** restores the environment saved by the most recent call to **setjmp()** macro in the same invocation of the program with the corresponding **jmp_buf** argument.

Declaration

Following is the declaration for longjmp() function.

```
void longjmp(jmp_buf environment, int value)
```

Parameters

- **environment** -- This is the object of type **jmp_buf** containing information to restore the environment at the setjmp's calling point.
- **value** -- This is the value to which the **setjmp** expression evaluates.

Return Value

This function does not return any value.

Example

The following example shows the usage of longjmp() function.

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

int main()
{
   int val;
   jmp_buf env_buffer;

   /* save calling environment for longjmp */
```

```
   val = setjmp( env_buffer );
   if( val != 0 )
   {
      printf("Returned from a longjmp() with value = %s\n", val);
      exit(0);
   }
   printf("Jump function call\n");
   jmpfunction( env_buffer );

   return(0);
}

void jmpfunction(jmp_buf env_buf)
{
   longjmp(env_buf, "tutorialspoint.com");
}
```

Let us compile and run the above program that will produce the following result:

```
Jump function call
Returned from a longjmp() with value = tutorialspoint.com
```

9. C Library – <signal.h>

Introduction

The **signal.h** header defines a variable type **sig_atomic_t**, two function calls, and several macros to handle different signals reported during a program's execution.

Library Variables

Following is the variable type defined in the header signal.h:

S.N.	Variable & Description
1	**sig_atomic_t** This is of **int** type and is used as a variable in a signal handler. This is an integral type of an object that can be accessed as an atomic entity, even in the presence of asynchronous signals.

Library Macros

Following are the macros defined in the header signal.h and these macros will be used in two functions listed below. The **SIG_** macros are used with the signal function to define signal functions.

S.N.	Macro & Description
1	**SIG_DFL** Default signal handler.
2	**SIG_ERR** Represents a signal error.
3	**SIG_IGN** Signal ignore.

The **SIG** macros are used to represent a signal number in the following conditions:

S.N.	Macro & Description
1	**SIGABRT** Abnormal program termination.
2	**SIGFPE** Floating-point error like division by zero.
3	**SIGILL** Illegal operation.
4	**SIGINT** Interrupt signal such as ctrl-C.
5	**SIGSEGV** Invalid access to storage like segment violation.
6	**SIGTERM** Termination request.

Library Functions

Following are the functions defined in the header signal.h:

S.N.	Function & Description
1	void (*signal(int sig, void (*func)(int)))(int) This function sets a function to handle signal i.e. a signal handler.
2	int raise(int sig) This function causes signal **sig** to be generated. The sig argument is compatible with the SIG macros.

void (*signal(int sig, void (*func)(int)))(int)

Description

The C library function **void (*signal(int sig, void (*func)(int)))(int)** sets a function to handle signal i.e. a signal handler with signal number **sig**.

Declaration

Following is the declaration for signal() function.

```
void (*signal(int sig, void (*func)(int)))(int)
```

Parameters

- **sig** -- This is the signal number to which a handling function is set. The following are few important standard signal numbers:

macro	signal
SIGABRT	(Signal Abort) Abnormal termination, such as is initiated by the function.
SIGFPE	(Signal Floating-Point Exception) Erroneous arithmetic operation, such as zero divide or an operation resulting in overflow (not necessarily with a floating-point operation).
SIGILL	(Signal Illegal Instruction) Invalid function image, such as an illegal instruction. This is generally due to a corruption in the code or to an attempt to execute data.
SIGINT	(Signal Interrupt) Interactive attention signal. Generally generated by the application user.
SIGSEGV	(Signal Segmentation Violation) Invalid access to storage: When a program tries to read or write outside the memory it is allocated for it.
SIGTERM	(Signal Terminate) Termination request sent to program.

- **func** -- This is a pointer to a function. This can be a function defined by the programmer or one of the following predefined functions:

SIG_DFL	Default handling: The signal is handled by the default action for that particular signal.
SIG_IGN	Ignore Signal: The signal is ignored.

Return Value

This function returns the previous value of the signal handler, or SIG_ERR on error.

Example

The following example shows the usage of signal() function.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

void sighandler(int);

int main()
{
   signal(SIGINT, sighandler);

   while(1)
   {
      printf("Going to sleep for a second...\n");
      sleep(1);
   }

   return(0);
}

void sighandler(int signum)
{
   printf("Caught signal %d, coming out...\n", signum);
   exit(1);
}
```

Let us compile and run the above program that will produce the following result and program will go in infinite loop. To come out of the program we used CTRL + C keys.

```
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
Caught signal 2, coming out...
```

int raise(int sig)

Description

The C library function **int raise(int sig)** causes signal **sig** to be generated. The **sig** argument is compatible with the SIG macros.

Declaration

Following is the declaration for signal() function.

```
int raise(int sig)
```

Parameters

- **sig** -- This is the signal number to send. Following are few important standard signal constants:

macro	signal
SIGABRT	(Signal Abort) Abnormal termination, such as is initiated by the abort function.
SIGFPE	(Signal Floating-Point Exception) Erroneous arithmetic operation, such as zero divide or an operation resulting in overflow (not necessarily with a floating-point operation).
SIGILL	(Signal Illegal Instruction) Invalid function image, such as an illegal instruction. This is generally due to a corruption in the code or to an attempt to execute data.
SIGINT	(Signal Interrupt) Interactive attention signal. Generally generated by the application user.
SIGSEGV	(Signal Segmentation Violation) Invalid access to storage: When a program tries to read or write outside the memory it is allocated for it.
SIGTERM	(Signal Terminate) Termination request sent to program.

Return Value

This function returns zero if successful, and non-zero otherwise.

Example

The following example shows the usage of signal() function.

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

void signal_catchfunc(int);
```

```
int main()
{
    int ret;

   ret = signal(SIGINT, signal_catchfunc);

   if( ret == SIG_ERR)
   {
      printf("Error: unable to set signal handler.\n");
      exit(0);
   }
   printf("Going to raise a signal\n");
   ret = raise(SIGINT);
   if( ret !=0 )
   {
      printf("Error: unable to raise SIGINT signal.\n");
      exit(0);
   }

   printf("Exiting...\n");
   return(0);
}

void signal_catchfunc(int signal)
{
   printf("!! signal caught !!\n");
}
```

Let us compile and run the above program to produce the following result:

```
Going to raise a signal
!! signal caught !!
Exiting...
```

10. C Library – <stdarg.h>

Introduction

The **stdarg.h** header defines a variable type **va_list** and three macros which can be used to get the arguments in a function when the number of arguments are not known i.e. variable number of arguments.

A function of variable arguments is defined with the ellipsis (,...) at the end of the parameter list.

Library Variables

Following is the variable type defined in the header stdarg.h:

S.N.	Variable & Description
1	**va_list** This is a type suitable for holding information needed by the three macros **va_start(), va_arg()** and **va_end()**.

Library Macros

Following are the macros defined in the header stdarg.h:

S.N.	Macro & Description
1	void va_start(va_list ap, last_arg) This macro initializes **ap** variable to be used with the **va_arg** and **va_end** macros. The **last_arg** is the last known fixed argument being passed to the function i.e. the argument before the ellipsis.
2	type va_arg(va_list ap, type) This macro retrieves the next argument in the parameter list of the function with type **type**.
3	void va_end(va_list ap) This macro allows a function with variable arguments which used the **va_start** macro to return. If **va_end** is not called before returning from the function, the result is undefined.

void va_start(va_list ap, last_arg)

Description

The C library macro **void va_start(va_list ap, last_arg)** initializes **ap** variable to be used with the **va_arg** and **va_end** macros. The **last_arg** is the last known fixed argument being passed to the function i.e. the argument before the ellipsis.

This macro must be called before using **va_arg** and **va_end**.

Declaration

Following is the declaration for va_start() macro.

```
void va_start(va_list ap, last_arg);
```

Parameters

- **ap** -- This is the object of **va_list** and it will hold the information needed to retrieve the additional arguments with **va_arg**.
- **last_arg** -- This is the last known fixed argument being passed to the function.

Return Value

NA

Example

The following example shows the usage of va_start() macro.

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

int sum(int, ...);

int main(void)
{
   printf("Sum of 10, 20 and 30 = %d\n",  sum(3, 10, 20, 30) );
   printf("Sum of 4, 20, 25 and 30 = %d\n",  sum(4, 4, 20, 25, 30) );

   return 0;
}

int sum(int num_args, ...)
{
   int val = 0;
   va_list ap;
   int i;
```

```
   va_start(ap, num_args);
   for(i = 0; i < num_args; i++)
   {
      val += va_arg(ap, int);
   }
   va_end(ap);

   return val;
}
```

Let us compile and run the above program to produce the following result:

```
Sum of 10, 20 and 30 = 60
Sum of 4, 20, 25 and 30 = 79
```

type va_arg(va_list ap, type)

Description

The C library macro **type va_arg(va_list ap, type)** retrieves the next argument in the parameter list of the function with **type**. This does not determine whether the retrieved argument is the last argument passed to the function.

Declaration

Following is the declaration for va_arg() macro.

```
type va_arg(va_list ap, type)
```

Parameters

- **ap** -- This is the object of type va_list with information about the additional arguments and their retrieval state. This object should be initialized by an initial call to va_start before the first call to va_arg.
- **type** -- This is a type name. This type name is used as the type of the expression, this macro expands to.

Return Value

This macro returns the next additional argument as an expression of type **type**.

Example

The following example shows the usage of va_arg() macro.

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

int sum(int, ...);
```

```
int main()
{
   printf("Sum of 15 and 56 = %d\n",  sum(2, 15, 56) );
   return 0;
}

int sum(int num_args, ...)
{
   int val = 0;
   va_list ap;
   int i;

   va_start(ap, num_args);
   for(i = 0; i < num_args; i++)
   {
      val += va_arg(ap, int);
   }
   va_end(ap);

   return val;
}
```

Let us compile and run the above program to produce the following result:

```
Sum of 15 and 56 = 71
```

void va_end(va_list ap)

Description

The C library macro **void va_end(va_list ap)** allows a function with variable arguments which used the **va_start** macro to return. If **va_end** is not called before returning from the function, the result is undefined.

Declaration

Following is the declaration for va_end() macro.

```
void va_end(va_list ap)
```

Parameters

- **ap** -- This is the va_list object previously initialized by va_start in the same function.

Return Value

This macro does not return any value.

Example

The following example shows the usage of va_end() macro.

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

int mul(int, ...);

int main()
{
   printf("15 * 12 = %d\n",  mul(2, 15, 12) );

   return 0;
}

int mul(int num_args, ...)
{
   int val = 1;
   va_list ap;
   int i;

   va_start(ap, num_args);
   for(i = 0; i < num_args; i++)
   {
      val *= va_arg(ap, int);
   }
   va_end(ap);

   return val;
}
```

Let us compile and run the above program to produce the following result:

```
15 * 12 =  180
```

11. C Library – <stddef.h>

Introduction

The **stddef.h** header defines various variable types and macros. Many of these definitions also appear in other headers.

Library Variables

Following are the variable types defined in the header stddef.h:

S.N.	Variable & Description
1	**ptrdiff_t** This is the signed integral type and is the result of subtracting two pointers.
2	**size_t** This is the unsigned integral type and is the result of the **sizeof** keyword.
3	**wchar_t** This is an integral type of the size of a wide character constant.

Library Macros

Following are the macros defined in the header stddef.h:

S.N.	Macro & Description
1	**NULL** This macro is the value of a null pointer constant.
2	**offsetof(type, member-designator)** This results in a constant integer of type size_t which is the offset in bytes of a structure member from the beginning of the structure. The member is given by *member-designator*, and the name of the structure is given in *type*.

NULL

Description

The C library Macro **NULL** is the value of a null pointer constant. It may be defined as **((void*)0), 0** or **0L** depending on the compiler vendor.

Declaration

Following may be the declaration for NULL Macro depending on the compiler.

```
#define NULL ((char *)0)

or

#define NULL 0L

or

#define NULL 0
```

Parameters

- NA

Return Value

- NA

Example

The following example shows the usage of NULL Macro.

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

int main ()
{
   FILE *fp;

   fp = fopen("file.txt", "r");
   if( fp != NULL )
   {
      printf("Opend file file.txt successfully\n");
      fclose(fp);
   }
```

```
   fp = fopen("nofile.txt", "r");
   if( fp == NULL )
   {
      printf("Could not open file nofile.txt\n");
   }

   return(0);
}
```

Assuming we have an existing file **file.txt** but **nofile.txt** does not exist. Let us compile and run the above program that will produce the following result:

```
Opend file file.txt successfully
Could not open file nofile.txt
```

offsetof(type, member-designator)

Description

The C library macro **offsetof(type, member-designator)** results in a constant integer of type **size_t** which is the offset in bytes of a structure member from the beginning of the structure. The member is given by member-designator, and the name of the structure is given in type.

Declaration

Following is the declaration for offsetof() Macro.

```
offsetof(type, member-designator)
```

Parameters

- **type** -- This is the class type in which member-designator is a valid member designator.
- **member-designator** -- This is the member designator of class type.

Return Value

This macro returns the value of type **size_t** which is the offset value of member in type.

Example

The following example shows the usage of offsetof() Macro.

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

struct address {
   char name[50];
```

```
   char street[50];
   int phone;
};

int main()
{
   printf("name offset = %d byte in address structure.\n",
   offsetof(struct address, name));

   printf("street offset = %d byte in address structure.\n",
   offsetof(struct address, street));

   printf("phone offset = %d byte in address structure.\n",
   offsetof(struct address, phone));

   return(0);
}
```

Let us compile and run the above program, this will produce the following result:

```
name offset = 0 byte in address structure.
street offset = 50 byte in address structure.
phone offset = 100 byte in address structure.
```

12. C Library – <stdio.h>

Introduction

The **stdio.h** header defines three variable types, several macros, and various functions for performing input and output.

Library Variables

Following are the variable types defined in the header stdio.h:

S.N.	Variable & Description
1	**size_t** This is the unsigned integral type and is the result of the **sizeof** keyword.
2	**FILE** This is an object type suitable for storing information for a file stream.
3	**fpos_t** This is an object type suitable for storing any position in a file.

Library Macros

Following are the macros defined in the header stdio.h:

S.N.	Macro & Description
1	**NULL** This macro is the value of a null pointer constant.
2	**_IOFBF, _IOLBF** and **_IONBF** These are the macros which expand to integral constant expressions with distinct values and suitable for the use as third argument to the **setvbuf** function.
3	**BUFSIZ** This macro is an integer, which represents the size of the buffer used by the **setbuf** function.

4	**EOFM** This macro is a negative integer, which indicates that the end-of-file has been reached.
5	**FOPEN_MAX** This macro is an integer, which represents the maximum number of files that the system can guarantee to be opened simultaneously.
6	**FILENAME_MAX** This macro is an integer, which represents the longest length of a char array suitable for holding the longest possible filename. If the implementation imposes no limit, then this value should be the recommended maximum value.
7	**L_tmpnam** This macro is an integer, which represents the longest length of a char array suitable for holding the longest possible temporary filename created by the **tmpnam** function.
8	**SEEK_CUR, SEEK_END,** and **SEEK_SET** These macros are used in the **fseek** function to locate different positions in a file.
9	**TMP_MAX** This macro is the maximum number of unique filenames that the function **tmpnam** can generate.
10	**stderr, stdin,** and **stdout** These macros are pointers to FILE types which correspond to the standard error, standard input, and standard output streams.

Library Functions

Following are the functions defined in the header stdio.h:

Follow the same sequence of functions for better understanding and to make use of **Try it** (online compiler) option, because file created in the first function will be used in subsequent functions.

S.N.	Function & Description
1	int fclose(FILE *stream)

	Closes the stream. All buffers are flushed.
2	void clearerr(FILE *stream) Clears the end-of-file and error indicators for the given stream.
3	int feof(FILE *stream) Tests the end-of-file indicator for the given stream.
4	int ferror(FILE *stream) Tests the error indicator for the given stream.
5	int fflush(FILE *stream) Flushes the output buffer of a stream.
6	int fgetpos(FILE *stream, fpos_t *pos) Gets the current file position of the stream and writes it to pos.
7	FILE *fopen(const char *filename, const char *mode) Opens the filename pointed to by filename using the given mode.
8	size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream) Reads data from the given stream into the array pointed to by ptr.
9	FILE *freopen(const char *filename, const char *mode, FILE *stream) Associates a new filename with the given open stream and same time closing the old file in stream.
10	int fseek(FILE *stream, long int offset, int whence) Sets the file position of the stream to the given offset. The argument *offset* signifies the number of bytes to seek from the given *whence* position.
11	int fsetpos(FILE *stream, const fpos_t *pos) Sets the file position of the given stream to the given position. The argument *pos* is a position given by the function fgetpos.
12	long int ftell(FILE *stream) Returns the current file position of the given stream.

13	size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream) Writes data from the array pointed to by ptr to the given stream.
14	int remove(const char *filename) Deletes the given filename so that it is no longer accessible.
15	int rename(const char *old_filename, const char *new_filename) Causes the filename referred to, by old_filename to be changed to new_filename.
16	void rewind(FILE *stream) Sets the file position to the beginning of the file of the given stream.
17	void setbuf(FILE *stream, char *buffer) Defines how a stream should be buffered.
18	int setvbuf(FILE *stream, char *buffer, int mode, size_t size) Another function to define how a stream should be buffered.
19	FILE *tmpfile(void) Creates a temporary file in binary update mode (wb+).
20	char *tmpnam(char *str) Generates and returns a valid temporary filename which does not exist.
21	int fprintf(FILE *stream, const char *format, ...) Sends formatted output to a stream.
22	int printf(const char *format, ...) Sends formatted output to stdout.
23	int sprintf(char *str, const char *format, ...) Sends formatted output to a string.
24	int vfprintf(FILE *stream, const char *format, va_list arg) Sends formatted output to a stream using an argument list.

25	int vprintf(const char *format, va_list arg) Sends formatted output to stdout using an argument list.
26	int vsprintf(char *str, const char *format, va_list arg) Sends formatted output to a string using an argument list.
27	int fscanf(FILE *stream, const char *format, ...) Reads formatted input from a stream.
28	int scanf(const char *format, ...) Reads formatted input from stdin.
29	int sscanf(const char *str, const char *format, ...) Reads formatted input from a string.
30	int fgetc(FILE *stream) Gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream.
31	char *fgets(char *str, int n, FILE *stream) Reads a line from the specified stream and stores it into the string pointed to by str. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.
32	int fputc(int char, FILE *stream) Writes a character (an unsigned char) specified by the argument char to the specified stream and advances the position indicator for the stream.
33	int fputs(const char *str, FILE *stream) Writes a string to the specified stream up to but not including the null character.
34	int getc(FILE *stream) Gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream.
35	int getchar(void) Gets a character (an unsigned char) from stdin.

36	char *gets(char *str) Reads a line from stdin and stores it into the string pointed to, by str. It stops when either the newline character is read or when the end-of-file is reached, whichever comes first.
37	int putc(int char, FILE *stream) Writes a character (an unsigned char) specified by the argument char to the specified stream and advances the position indicator for the stream.
38	int putchar(int char) Writes a character (an unsigned char) specified by the argument char to stdout.
39	int puts(const char *str) Writes a string to stdout up to but not including the null character. A newline character is appended to the output.
40	int ungetc(int char, FILE *stream) Pushes the character char (an unsigned char) onto the specified stream so that the next character is read.
41	void perror(const char *str) Prints a descriptive error message to stderr. First the string str is printed followed by a colon and then a space.

int fclose(FILE *stream)

Description

The C library function **int fclose(FILE *stream)** closes the stream. All buffers are flushed.

Declaration

Following is the declaration for fclose() function.

```
int fclose(FILE *stream)
```

Parameters

- **stream** -- This is the pointer to a FILE object that specifies the stream to be closed.

Return Value

This method returns zero if the stream is successfully closed. On failure, EOF is returned.

Example

The following example shows the usage of fclose() function.

```
#include <stdio.h>

int main()
{
   FILE *fp;

   fp = fopen("file.txt", "w");

   fprintf(fp, "%s", "This is tutorialspoint.com");
   fclose(fp);

   return(0);
}
```

Let us compile and run the above program that will create a file **file.txt**, and then it will write following text line and finally it will close the file using **fclose()** function.

```
This is tutorialspoint.com
```

void clearerr(FILE *stream)

Description

The C library function **void clearerr(FILE *stream)** clears the end-of-file and error indicators for the given stream.

Declaration

Following is the declaration for clearerr() function.

```
void clearerr(FILE *stream)
```

Parameters

- **stream** -- This is the pointer to a FILE object that identifies the stream.

Return Value

This should not fail and do not set the external variable errno but in case it detects that its argument is not a valid stream, it must return -1 and set errno to EBADF.

Example

The following example shows the usage of clearerr() function.

```
#include <stdio.h>
int main()
{
   FILE *fp;
   char c;

   fp = fopen("file.txt", "w");

   c = fgetc(fp);
   if( ferror(fp) )
   {
      printf("Error in reading from file : file.txt\n");
   }
   clearerr(fp);
   if( ferror(fp) )
   {
      printf("Error in reading from file : file.txt\n");
   }
   fclose(fp);

   return(0);
}
```

Assuming we have a text file **file.txt**, which is an empty file, let us compile and run the above program, this will produce the following result because we try to read a file which we opened in write only mode.

```
Error reading from file "file.txt"
```

int feof(FILE *stream)

Description

The C library function **int feof(FILE *stream)** tests the end-of-file indicator for the given stream.

Declaration

Following is the declaration for feof() function.

```
int feof(FILE *stream)
```

Parameters

- **stream** -- This is the pointer to a FILE object that identifies the stream.

Return Value

This function returns a non-zero value when End-of-File indicator associated with the stream is set, else zero is returned.

Example

The following example shows the usage of feof() function.

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   if(fp == NULL)
   {
      perror("Error in opening file");
      return(-1);
   }
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

Assuming we have a text file **file.txt**, which has the following content. This file will be used as an input for our example program:

```
This is tutorialspoint.com
```

Let us compile and run the above program, this will produce the following result:

```
This is tutorialspoint.com
```

int ferror(FILE *stream)

Description

The C library function **int ferror(FILE *stream)** tests the error indicator for the given stream.

Declaration

Following is the declaration for ferror() function.

```
int ferror(FILE *stream)
```

Parameters

- **stream** -- This is the pointer to a FILE object that identifies the stream.

Return Value

If the error indicator associated with the stream was set, the function returns a non-zero value else, it returns a zero value.

Example

The following example shows the usage of ferror() function.

```
#include <stdio.h>

int main()
{
   FILE *fp;
   char c;

   fp = fopen("file.txt", "w");

   c = fgetc(fp);
   if( ferror(fp) )
   {
      printf("Error in reading from file : file.txt\n");
   }
   clearerr(fp);
   if( ferror(fp) )
   {
      printf("Error in reading from file : file.txt\n");
```

```
   }
   fclose(fp);

   return(0);
}
```

Assuming we have a text file **file.txt**, which is an empty file. Let us compile and run the above program that will produce the following result because we try to read a file which we opened in **write only** mode.

```
Error reading from file "file.txt"
```

int fflush(FILE *stream)

Description

The C library function **int fflush(FILE *stream)** flushes the output buffer of a stream.

Declaration

Following is the declaration for fflush() function.

```
int fflush(FILE *stream)
```

Parameters

- **stream** -- This is the pointer to a FILE object that specifies a buffered stream.

Return Value

This function returns a zero value on success. If an error occurs, EOF is returned and the error indicator is set (i.e. feof).

Example

The following example shows the usage of fflush() function.

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

int main()
{

   char buff[1024];

   memset( buff, '\0', sizeof( buff ));

   fprintf(stdout, "Going to set full buffering on\n");
   setvbuf(stdout, buff, _IOFBF, 1024);
```

```
   fprintf(stdout, "This is tutorialspoint.com\n");
   fprintf(stdout, "This output will go into buff\n");
   fflush( stdout );

   fprintf(stdout, "and this will appear when programm\n");
   fprintf(stdout, "will come after sleeping 5 seconds\n");

   sleep(5);

   return(0);
}
```

Let us compile and run the above program that will produce the following result. Here program keeps buffering the output into **buff** until it faces first call to **fflush()**, after which it again starts buffering the output and finally sleeps for 5 seconds. It sends remaining output to the STDOUT before program comes out.

```
Going to set full buffering on
This is tutorialspoint.com
This output will go into buff
and this will appear when programm
will come after sleeping 5 seconds
```

int fgetpos(FILE *stream, fpos_t *pos)

Description

The C library function **int fgetpos(FILE *stream, fpos_t *pos)** gets the current file position of the **stream** and writes it to **pos**.

Declaration

Following is the declaration for fgetpos() function.

```
int fgetpos(FILE *stream, fpos_t *pos)
```

Parameters

- **stream** -- This is the pointer to a FILE object that identifies the stream.
- **pos** -- This is the pointer to a fpos_t object.

Return Value

This function returns zero on success, else non-zero value in case of an error.

Example

The following example shows the usage of fgetpos() function.

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   fpos_t position;

   fp = fopen("file.txt","w+");
   fgetpos(fp, &position);
   fputs("Hello, World!", fp);

   fsetpos(fp, &position);
   fputs("This is going to override previous content", fp);
   fclose(fp);

   return(0);
}
```

Let us compile and run the above program to create a file **file.txt** which will have the following content. First of all we get the initial position of the file using **fgetpos()** function and then we write *Hello, World!* in the file, but later we have used **fsetpos()** function to reset the write pointer at the beginning of the file and then over-write the file with the following content:

```
This is going to override previous content
```

Now let us see the content of the above file using the following program:

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;
   int n = 0;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
```

```
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }

   fclose(fp);

   return(0);
}
```

FILE *fopen(const char *filename, const char *mode)

Description

The C library function **FILE *fopen(const char *filename, const char *mode)** opens the **filename** pointed to, by filename using the given **mode**.

Declaration

Following is the declaration for fopen() function.

```
FILE *fopen(const char *filename, const char *mode)
```

Parameters

- **filename** -- This is the C string containing the name of the file to be opened.
- **mode** -- This is the C string containing a file access mode. It includes:

mode	Description
"r"	Opens a file for reading. The file must exist.
"w"	Creates an empty file for writing. If a file with the same name already exists, its content is erased and the file is considered as a new empty file.
"a"	Appends to a file. Writing operations, append data at the end of the file. The file is created if it does not exist.
"r+"	Opens a file to update both reading and writing. The file must exist.
"w+"	Creates an empty file for both reading and writing.

"a+"	Opens a file for reading and appending.

Return Value

This function returns a FILE pointer. Otherwise, NULL is returned and the global variable errno is set to indicate the error.

Example

The following example shows the usage of fopen() function.

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

int main()
{
   FILE * fp;

   fp = fopen ("file.txt", "w+");
   fprintf(fp, "%s %s %s %d", "We", "are", "in", 2012);

   fclose(fp);

   return(0);
}
```

Let us compile and run the above program that will create a file **file.txt** with the following content:

```
We are in 2012
```

Now let us see the content of the above file using the following program:

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
```

```
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)

Description

The C library function **size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)** reads data from the given **stream** into the array pointed to, by **ptr**.

Declaration

Following is the declaration for fread() function.

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
```

Parameters

- **ptr** -- This is the pointer to a block of memory with a minimum size of *size*nmemb* bytes.
- **size** -- This is the size in bytes of each element to be read.
- **nmemb** -- This is the number of elements, each one with a size of **size** bytes.
- **stream** -- This is the pointer to a FILE object that specifies an input stream.

Return Value

The total number of elements successfully read are returned as a size_t object, which is an integral data type. If this number differs from the nmemb parameter, then either an error had occurred or the End Of File was reached.

Example

The following example shows the usage of fread() function.

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

int main()
{
   FILE *fp;
   char c[] = "this is tutorialspoint";
```

```
   char buffer[20];

   /* Open file for both reading and writing */
   fp = fopen("file.txt", "w+");

   /* Write data to the file */
   fwrite(c, strlen(c) + 1, 1, fp);

   /* Seek to the beginning of the file */
   fseek(fp, SEEK_SET, 0);

   /* Read and display data */
   fread(buffer, strlen(c)+1, 1, fp);
   printf("%s\n", buffer);
   fclose(fp);

   return(0);
}
```

Let us compile and run the above program that will create a file **file.txt** and write a content *this is tutorialspoint*. After that, we use **fseek()** function to reset writing pointer to the beginning of the file and prepare the file content which is as follows:

```
this is tutorialspoint
```

FILE *freopen(const char *filename, const char *mode, FILE *stream)

Description

The C library function **FILE *freopen(const char *filename, const char *mode, FILE *stream)** associates a new **filename** with the given open stream and at the same time closes the old file in the stream.

Declaration

Following is the declaration for freopen() function.

```
FILE *freopen(const char *filename, const char *mode, FILE *stream)
```

Parameters

- **filename** -- This is the C string containing the name of the file to be opened.
- **mode** -- This is the C string containing a file access mode. It includes:

mode	Description
"r"	Opens a file for reading. The file must exist.
"w"	Creates an empty file for writing. If a file with the same name already exists then its content is erased and the file is considered as a new empty file.
"a"	Appends to a file. Writing operations appends data at the end of the file. The file is created if it does not exist.
"r+"	Opens a file to update both reading and writing. The file must exist.
"w+"	Creates an empty file for both reading and writing.
"a+"	Opens a file for reading and appending.

- **stream** -- This is the pointer to a FILE object that identifies the stream to be re-opened.

Return Value

If the file was re-opened successfully, the function returns a pointer to an object identifying the stream or else, null pointer is returned.

Example

The following example shows the usage of freopen() function.

```
#include <stdio.h>

int main ()
{
   FILE *fp;

   printf("This text is redirected to stdout\n");

   fp = freopen("file.txt", "w+", stdout);

   printf("This text is redirected to file.txt\n");

   fclose(fp);

   return(0);
```

```
}
```

Let us compile and run the above program that will send the following line at STDOUT because initially we did not open stdout:

```
This text is redirected to stdout
```

After a call to **freopen()**, it associates STDOUT to file **file.txt**, so whatever we write at STDOUT, goes inside **file.txt**. So, the file **file.txt** will have the following content.

```
This text is redirected to file.txt
```

Now let's see the content of the above file using the following program:

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

int fseek(FILE *stream, long int offset, int whence)

Description

The C library function **int fseek(FILE *stream, long int offset, int whence)** sets the file position of the **stream** to the given **offset**.

Declaration

Following is the declaration for fseek() function.

```
int fseek(FILE *stream, long int offset, int whence)
```

Parameters

- **stream** -- This is the pointer to a FILE object that identifies the stream.
- **offset** -- This is the number of bytes to offset from whence.
- **whence** -- This is the position from where offset is added. It is specified by one of the following constants:

Constant	Description
SEEK_SET	Beginning of file
SEEK_CUR	Current position of the file pointer
SEEK_END	End of file

Return ValueThis function returns zero if successful, or else it returns a non-zero value.

Example

The following example shows the usage of fseek() function.

```
#include <stdio.h>

int main ()
{
   FILE *fp;

   fp = fopen("file.txt","w+");
   fputs("This is tutorialspoint.com", fp);

   fseek( fp, 7, SEEK_SET );
   fputs(" C Programming Langauge", fp);
   fclose(fp);

   return(0);
}
```

Let us compile and run the above program that will create a file **file.txt** with the following content. Initially program creates the file and writes *This is tutorialspoint.com*, but later we had reset the write pointer at 7th position from the beginning and used puts() statement which over-write the file with the following content:

```
This is C Programming Langauge
```

Now let's see the content of the above file using the following program:

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

int fsetpos(FILE *stream, const fpos_t *pos)

Description

The C library function **int fsetpos(FILE *stream, const fpos_t *pos)** sets the file position of the given **stream** to the given position. The argument **pos** is a position given by the function fgetpos.

Declaration

Following is the declaration for fsetpos() function.

```
int fsetpos(FILE *stream, const fpos_t *pos)
```

Parameters

- **stream** -- This is the pointer to a FILE object that identifies the stream.

- **pos** -- This is the pointer to a fpos_t object containing a position previously obtained with fgetpos.

Return Value

This function returns zero value if successful, or else it returns a non-zero value and sets the global variable **errno** to a positive value, which can be interpreted with perror.

Example

The following example shows the usage of fsetpos() function.

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   fpos_t position;

   fp = fopen("file.txt","w+");
   fgetpos(fp, &position);
   fputs("Hello, World!", fp);

   fsetpos(fp, &position);
   fputs("This is going to override previous content", fp);
   fclose(fp);

   return(0);
}
```

Let us compile and run the above program to create a file **file.txt** which will have the following content. First of all we get the initial position of the file using **fgetpos()** function, and then we write *Hello, World!* in the file but later we used **fsetpos()** function to reset the write pointer at the beginning of the file and then over-write the file with the following content:

```
This is going to override previous content
```

Now let's see the content of the above file using the following program:

```
#include <stdio.h>

int main ()
{
   FILE *fp;
```

```
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

long int ftell(FILE *stream)

Description

The C library function **long int ftell(FILE *stream)** returns the current file position of the given stream.

Declaration

Following is the declaration for ftell() function.

```
long int ftell(FILE *stream)
```

Parameters

- **stream** -- This is the pointer to a FILE object that identifies the stream.

Return Value

This function returns the current value of the position indicator. If an error occurs, -1L is returned, and the global variable errno is set to a positive value.

Example

The following example shows the usage of ftell() function.

```
#include <stdio.h>

int main ()
{
```

```
   FILE *fp;
   int len;

   fp = fopen("file.txt", "r");
   if( fp == NULL )
   {
      perror ("Error opening file");
      return(-1);
   }
   fseek(fp, 0, SEEK_END);

   len = ftell(fp);
   fclose(fp);

   printf("Total size of file.txt = %d bytes\n", len);

   return(0);
}
```

Let us assume we have a text file **file.txt**, which has the following content:

```
This is tutorialspoint.com
```

Now let us compile and run the above program that will produce the following result if file has above mentioned content otherwise it will give different result based on the file content:

```
Total size of file.txt = 27 bytes
```

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)

Description

The C library function **size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)** writes data from the array pointed to, by **ptr** to the given **stream**.

Declaration

Following is the declaration for fwrite() function.

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
```

Parameters

- **ptr** -- This is the pointer to the array of elements to be written.

- **size** -- This is the size in bytes of each element to be written.
- **nmemb** -- This is the number of elements, each one with a size of **size** bytes.
- **stream** -- This is the pointer to a FILE object that specifies an output stream.

Return Value

This function returns the total number of elements successfully returned as a size_t object, which is an integral data type. If this number differs from the nmemb parameter, it will show an error.

Example

The following example shows the usage of fwrite() function.

```
#include<stdio.h>

int main ()
{
   FILE *fp;
   char str[] = "This is tutorialspoint.com";

   fp = fopen( "file.txt" , "w" );
   fwrite(str , 1 , sizeof(str) , fp );

   fclose(fp);

   return(0);
}
```

Let us compile and run the above program that will create a file **file.txt** which will have following content:

```
This is tutorialspoint.com
```

Now let's see the content of the above file using the following program:

```
#include <stdio.h>

int main ()
{
```

```
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

int remove(const char *filename)

Description

The C library function **int remove(const char *filename)** deletes the given **filename** so that it is no longer accessible.

Declaration

Following is the declaration for remove() function.

```
int remove(const char *filename)
```

Parameters

- **filename** -- This is the C string containing the name of the file to be deleted.

Return Value

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

Example

The following example shows the usage of remove() function.

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

```
int main ()
{
   int ret;
   FILE *fp;
   char filename[] = "file.txt";

   fp = fopen(filename, "w");

   fprintf(fp, "%s", "This is tutorialspoint.com");
   fclose(fp);

   ret = remove(filename);

   if(ret == 0)
   {
      printf("File deleted successfully");
   }
   else
   {
      printf("Error: unable to delete the file");
   }

   return(0);
}
```

Let us assume we have a text file **file.txt** having some content. So we are going to delete this file, using the above program. Let us compile and run the above program to produce the following message and the file will be deleted permanently.

```
File deleted successfully
```

int rename(const char *old_filename, const char *new_filename)

Description

The C library function int rename(const char *old_filename, const char *new_filename) causes the filename referred to by old_filename to be changed to new_filename.

Declaration

Following is the declaration for rename() function.

```
int rename(const char *old_filename, const char *new_filename)
```

Parameters

- **old_filename** -- This is the C string containing the name of the file to be renamed and/or moved.
- **new_filename** -- This is the C string containing the new name for the file.

Return Value

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

Example

The following example shows the usage of rename() function.

```
#include <stdio.h>

int main ()
{
   int ret;
   char oldname[] = "file.txt";
   char newname[] = "newfile.txt";

   ret = rename(oldname, newname);

   if(ret == 0)
   {
      printf("File renamed successfully");
   }
   else
   {
      printf("Error: unable to rename the file");
   }
   return(0);
}
```

Let us assume we have a text file **file.txt**, having some content. So, we are going to rename this file, using the above program. Let us compile and run the above program to produce the following message and the file will be renamed to **newfile.txt** file.

```
File renamed successfully
```

void rewind(FILE *stream)

Description

The C library function **void rewind(FILE *stream)** sets the file position to the beginning of the file of the given **stream**.

Declaration

Following is the declaration for rewind() function.

```
void rewind(FILE *stream)
```

Parameters

- **stream** -- This is the pointer to a FILE object that identifies the stream.

Return Value

This function does not return any value.

Example

The following example shows the usage of rewind() function.

```
#include <stdio.h>

int main()
{
   char str[] = "This is tutorialspoint.com";
   FILE *fp;
   int ch;

   /* First let's write some content in the file */
   fp = fopen( "file.txt" , "w" );
   fwrite(str , 1 , sizeof(str) , fp );
   fclose(fp);

   fp = fopen( "file.txt" , "r" );
   while(1)
   {
      ch = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", ch);
```

```
   }
   rewind(fp);
   printf("\n");
   while(1)
   {
      ch = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", ch);

   }
   fclose(fp);
   return(0);
}
```

Let us assume we have a text file **file.txt** that have the following content:

```
This is tutorialspoint.com
```

Now let us compile and run the above program to produce the following result:

```
This is tutorialspoint.com
This is tutorialspoint.com
```

void setbuf(FILE *stream, char *buffer)

Description

The C library function **void setbuf(FILE *stream, char *buffer)** defines how a stream should be buffered. This function should be called once the file associated with the stream has already been opened, but before any input or output operation has taken place.

Declaration

Following is the declaration for setbuf() function.

```
void setbuf(FILE *stream, char *buffer)
```

Parameters

- **stream** -- This is the pointer to a FILE object that identifies an open stream.
- **buffer** -- This is the user allocated buffer. This should have a length of at least BUFSIZ bytes, which is a macro constant to be used as the length of this array.

Return Value

This function does not return any value.

Example

The following example shows the usage of setbuf() function.

```
#include <stdio.h>

int main()
{
   char buf[BUFSIZ];

   setbuf(stdout, buf);
   puts("This is tutorialspoint");

   fflush(stdout);
   return(0);
}
```

Let us compile and run the above program to produce the following result. Here program sends output to the STDOUT just before it comes out, otherwise it keeps buffering the output. You can also use fflush() function to flush the output.

```
This is tutorialspoint
```

int setvbuf(FILE *stream, char *buffer, int mode, size_t size)

Description

The C library function **int setvbuf(FILE *stream, char *buffer, int mode, size_t size)**defines how a stream should be buffered.

Declaration

Following is the declaration for setvbuf() function.

```
int setvbuf(FILE *stream, char *buffer, int mode, size_t size)
```

Parameters

- **stream** -- This is the pointer to a FILE object that identifies an open stream.
- **buffer** -- This is the user allocated buffer. If set to NULL, the function automatically allocates a buffer of the specified size.
- **mode** -- This specifies a mode for file buffering:

mode	Description

_IOFBF	**Full buffering**: On output, data is written once the buffer is full. On Input the buffer is filled when an input operation is requested and the buffer is empty.
_IOLBF	**Line buffering**: On output, data is written when a newline character is inserted into the stream or when the buffer is full, whatsoever happens first. On Input, the buffer is filled till the next newline character when an input operation is requested and buffer is empty.
_IONBF	**No buffering**: No buffer is used. Each I/O operation is written as soon as possible. The buffer and size parameters are ignored.

- **size** -- This is the buffer size in bytes

Return Value

This function returns zero on success else, non-zero value is returned.

Example

The following example shows the usage of setvbuf() function.

```
#include <stdio.h>

int main()
{

   char buff[1024];

   memset( buff, '\0', sizeof( buff ));

   fprintf(stdout, "Going to set full buffering on\n");
   setvbuf(stdout, buff, _IOFBF, 1024);

   fprintf(stdout, "This is tutorialspoint.com\n");
   fprintf(stdout, "This output will go into buff\n");
   fflush( stdout );

   fprintf(stdout, "and this will appear when programm\n");
   fprintf(stdout, "will come after sleeping 5 seconds\n");

   sleep(5);
```

```
   return(0);
}
```

Let us compile and run the above program to produce the following result. Here program keeps buffering the output into **buff** until it faces first call to fflush(), after which it again starts buffering the output and finally sleeps for 5 seconds. It sends remaining output to the STDOUT before the program comes out.

```
Going to set full buffering on
This is tutorialspoint.com
This output will go into buff
and this will appear when programm
will come after sleeping 5 seconds
```

FILE *tmpfile(void)

Description

The C library function **FILE *tmpfile(void)** creates a temporary file in binary update mode (wb+). The temporary file created is automatically deleted when the stream is closed (fclose) or when the program terminates.

Declaration

Following is the declaration for tmpfile() function.

```
FILE *tmpfile(void)
```

Parameters

- NA

Return Value

If successful, the function returns a stream pointer to the temporary file created. If the file cannot be created, then NULL is returned.

Example

The following example shows the usage of tmpfile() function.

```
#include <stdio.h>

int main ()
{
   FILE *fp;

   fp = tmpfile();
   printf("Temporary file created\n");
```

```
   /* you can use tmp file here */

   fclose(fp);

   return(0);
}
```

Let us compile and run the above program to create a temporary file in /tmp folder but once your program is out, it will be deleted automatically and the program will produce the following result:

```
Temporary file created
```

char *tmpnam(char *str)

Description

The C library function **char *tmpnam(char *str)** generates and returns a valid temporary filename which does not exist. If **str** is null then it simply returns the tmp file name.

Declaration

Following is the declaration for tmpnam() function.

```
char *tmpnam(char *str)
```

Parameters

- **str** -- This is the pointer to an array of chars where the proposed temp name will be stored as a C string.

Return Value

- Return value is a pointer to the C string containing the proposed name for a temporary file. If str was a null pointer, this points to an internal buffer that will be overwritten the next time this function is called.
- If str was not a null pointer, str is returned. If the function fails to create a suitable filename, it returns a null pointer.

Example

The following example shows the usage of tmpnam() function.

```
#include <stdio.h>

int main()
{
```

```
   char buffer[L_tmpnam];
   char *ptr;



   tmpnam(buffer);
   printf("Temporary name 1: %s\n", buffer);

   ptr = tmpnam(NULL);
   printf("Temporary name 2: %s\n", ptr);

   return(0);
}
```

Let us compile and run the above program to produce the following result:

```
Temporary name 1: /tmp/filebaalTb
Temporary name 2: /tmp/filedCIbb0
```

int fprintf(FILE *stream, const char *format, ...)

Description

The C library function **int fprintf(FILE *stream, const char *format, ...)** sends formatted output to a stream.

Declaration

Following is the declaration for fprintf() function.

```
int fprintf(FILE *stream, const char *format, ...)
```

Parameters

- **stream** -- This is the pointer to a FILE object that identifies the stream.
- **format** -- This is the C string that contains the text to be written to the stream. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and formatted as requested. Format tags prototype is **%[flags][width][.precision][length]specifier**, which is explained below:

specifier	Output
c	Character
d or i	Signed decimal integer

e	Scientific notation (mantissa/exponent) using e character
E	Scientific notation (mantissa/exponent) using E character
f	Decimal floating point
g	Uses the shorter of %e or %f
G	Uses the shorter of %E or %f
o	Signed octal
s	String of characters
u	Unsigned decimal integer
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer (capital letters)
p	Pointer address
n	Nothing printed
%	Character

flags	**Description**
-	Left-justifies within the given field width; Right justification is the default (see width sub-specifier).
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a -ve sign.
(space)	If no sign is written, a blank space is inserted before the value.
#	Used with o, x or X specifiers. The value is preceded with 0, 0x or 0X respectively for values different than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow then no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros

	are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).
width	**Description**
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.
.precision	**Description**
.number	For integer specifiers (d, i, o, u, x, X): precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For e, E and f specifiers: this is the number of digits to be printed after the decimal point. For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. For c type: it has no effect. When no precision is specified, the default is 1. If the period is specified without an explicit value for precision, 0 is assumed.
.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.
length	**Description**
h	The argument is interpreted as a short int or unsigned short int (only applies to integer specifiers: i, d, o, u, x and X).
l	The argument is interpreted as a long int or unsigned long int for integer specifiers (i, d, o, u, x and X), and as a wide character or wide character string for specifiers c and s.
L	The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g and G).

- **additional arguments** -- Depending on the format string, the function may expect a sequence of additional arguments, each containing one value to be inserted instead of each %-tag specified in the format parameter, if any. There should be the same number of these arguments as the number of %-tags that expect a value.

Return Value

If successful, the total number of characters written is returned otherwise, a negative number is returned.

Example

The following example shows the usage of fprintf() function.

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

int main()
{
   FILE * fp;

   fp = fopen ("file.txt", "w+");
   fprintf(fp, "%s %s %s %d", "We", "are", "in", 2012);

   fclose(fp);

   return(0);
}
```

Let us compile and run the above program that will create a file **file.txt** with the following content:

```
We are in 2012
```

Now let's see the content of the above file using the following program:

```
#include <stdio.h>
int main ()
{
   FILE *fp;
```

```
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

int printf(const char *format, ...)

Description

The C library function **int printf(const char *format, ...)** sends formatted output to stdout.

Declaration

Following is the declaration for printf() function.

```
int printf(const char *format, ...)
```

Parameters

- **format** -- This is the string that contains the text to be written to stdout. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and formatted as requested. Format tags prototype is **%[flags][width][.precision][length]specifier**, which is explained below:

specifier	Output
c	Character
d or i	Signed decimal integer

e	Scientific notation (mantissa/exponent) using e character
E	Scientific notation (mantissa/exponent) using E character
f	Decimal floating point
g	Uses the shorter of %e or %f.
G	Uses the shorter of %E or %f
o	Signed octal
s	String of characters
u	Unsigned decimal integer
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer (capital letters)
p	Pointer address
n	Nothing printed
%	Character

flags	Description
-	Left-justify within the given field width; Right justification is the default (see width sub-specifier).
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a -ve sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceded with 0, 0x or 0X respectively for values different than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G

	the result is the same as with e or E but trailing zeros are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).

width	**Description**
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	**Description**
.number	For integer specifiers (d, i, o, u, x, X): precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For e, E and f specifiers: this is the number of digits to be printed after the decimal point. For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. For c type: it has no effect. When no precision is specified, the default is 1. If the period is specified without an explicit value for precision, 0 is assumed.
.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

length	**Description**
h	The argument is interpreted as a short int or unsigned short int (only applies to integer specifiers: i, d, o, u, x and X).
l	The argument is interpreted as a long int or unsigned long int for integer specifiers (i, d, o, u, x and X), and as a wide character or wide character string for specifiers c and s.
L	The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g and G).

- **additional arguments** -- Depending on the format string, the function may expect a sequence of additional arguments, each containing one value to be inserted instead of each %-tag specified in the format parameter (if any). There should be the same number of these arguments as the number of %-tags that expect a value.

Return Value

If successful, the total number of characters written is returned. On failure, a negative number is returned.

Example

The following example shows the usage of printf() function.

```
#include <stdio.h>

int main ()
{
   int ch;

   for( ch = 75 ; ch <= 100; ch++ ) {
      printf("ASCII value = %d, Character = %c\n", ch , ch );
   }

   return(0);
}
```

Let us compile and run the above program to produce the following result:

```
ASCII value = 75, Character = K
ASCII value = 76, Character = L
ASCII value = 77, Character = M
ASCII value = 78, Character = N
ASCII value = 79, Character = O
ASCII value = 80, Character = P
ASCII value = 81, Character = Q
ASCII value = 82, Character = R
ASCII value = 83, Character = S
ASCII value = 84, Character = T
ASCII value = 85, Character = U
ASCII value = 86, Character = V
ASCII value = 87, Character = W
ASCII value = 88, Character = X
ASCII value = 89, Character = Y
```

```
ASCII value = 90, Character = Z
ASCII value = 91, Character = [
ASCII value = 92, Character = \
ASCII value = 93, Character = ]
ASCII value = 94, Character = ^
ASCII value = 95, Character = _
ASCII value = 96, Character = `
ASCII value = 97, Character = a
ASCII value = 98, Character = b
ASCII value = 99, Character = c
ASCII value = 100, Character = d
```

int sprintf(char *str, const char *format, ...)

Description

The C library function **int sprintf(char *str, const char *format, ...)** sends formatted output to a string pointed to, by **str**.

Declaration

Following is the declaration for sprintf() function.

```
int sprintf(char *str, const char *format, ...)
```

Parameters

- **str** -- This is the pointer to an array of char elements where the resulting C string is stored.
- **format** -- This is the String that contains the text to be written to buffer. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and formatted as requested. Format tags prototype: **%[flags][width][.precision][length]specifier**, as explained below:

specifier	Output
c	Character
d or i	Signed decimal integer
e	Scientific notation (mantissa/exponent) using e character
E	Scientific notation (mantissa/exponent) using E character

f	Decimal floating point
g	Uses the shorter of %e or %f.
G	Uses the shorter of %E or %f
o	Signed octal
s	String of characters
u	Unsigned decimal integer
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer (capital letters)
p	Pointer address
n	Nothing printed
%	Character

flags	Description
-	Left-justify within the given field width; Right justification is the default (see width sub-specifier).
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a -ve sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceded with 0, 0x or 0X respectively for values different than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is

	specified (see width sub-specifier).
width	**Description**
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.
.precision	**Description**
.number	For integer specifiers (d, i, o, u, x, X): precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For e, E and f specifiers: this is the number of digits to be printed after the decimal point. For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. For c type: it has no effect. When no precision is specified, the default is 1. If the period is specified without an explicit value for precision, 0 is assumed.
.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.
length	**Description**
h	The argument is interpreted as a short int or unsigned short int (only applies to integer specifiers: i, d, o, u, x and X).
l	The argument is interpreted as a long int or unsigned long int for integer specifiers (i, d, o, u, x and X), and as a wide character or wide character string for specifiers c and s.
L	The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g and G).

- **additional arguments** -- Depending on the format string, the function may expect a sequence of additional arguments, each containing one value to be inserted instead of each %-tag specified in the format parameter (if any). There

should be the same number of these arguments as the number of %-tags that expect a value.

Return Value

If successful, the total number of characters written is returned excluding the null-character appended at the end of the string, otherwise a negative number is returned in case of failure.

Example

The following example shows the usage of sprintf() function.

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

int main()
{
   char str[80];

   sprintf(str, "Value of Pi = %f", M_PI);
   puts(str);

   return(0);
}
```

Let us compile and run the above program, this will produce the following result:

```
Value of Pi = 3.141593
```

int vfprintf(FILE *stream, const char *format, va_list arg)

Description

The C library function **int vfprintf(FILE *stream, const char *format, va_list arg)** sends formatted output to a stream using an argument list passed to it.

Declaration

Following is the declaration for vfprintf() function.

```
int vfprintf(FILE *stream, const char *format, va_list arg)
```

Parameters

- **stream** -- This is the pointer to a FILE object that identifies the stream.
- **format** -- This is the C string that contains the text to be written to the stream. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and formatted as requested. Format tags prototype: **%[flags][width][.precision][length]specifier**, as explained below:

specifier	Output
c	Character
d or i	Signed decimal integer
e	Scientific notation (mantissa/exponent) using e character
E	Scientific notation (mantissa/exponent) using E character
f	Decimal floating point
g	Uses the shorter of %e or %f
G	Uses the shorter of %E or %f
o	Signed octal
s	String of characters
u	Unsigned decimal integer
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer (capital letters)
p	Pointer address
n	Nothing printed
%	Character

flags	Description
-	Left-justify within the given field width; Right justification is the default (see width sub-specifier).
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a -ve sign.

(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceded with 0, 0x or 0X respectively for values different than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).
width	**Description**
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.
.precision	**Description**
.number	For integer specifiers (d, i, o, u, x, X): precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For e, E and f specifiers: this is the number of digits to be printed after the decimal point. For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. For c type: it has no effect. When no precision is specified, the default is 1. If the period is specified without an explicit value for precision, 0 is assumed.
.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.
length	**Description**
h	The argument is interpreted as a short int or unsigned short int (only applies to integer specifiers: i, d, o, u, x and X).

l	The argument is interpreted as a long int or unsigned long int for integer specifiers (i, d, o, u, x and X), and as a wide character or wide character string for specifiers c and s.
L	The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g and G).

- **arg** -- An object representing the variable arguments list. This should be initialized by the va_start macro defined in <stdarg>.

Return Value

If successful, the total number of characters written is returned otherwise, a negative number is returned.

Example

The following example shows the usage of vfprintf() function.

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

void WriteFrmtd(FILE *stream, char *format, ...)
{
   va_list args;

   va_start(args, format);
   vfprintf(stream, format, args);
   va_end(args);
}

int main ()
{
   FILE *fp;

   fp = fopen("file.txt","w");

   WriteFrmtd(fp, "This is just one argument %d \n", 10);

   fclose(fp);

   return(0);
```

```
}
```

Let us compile and run the above program that will open a file **file.txt** for writing in the current directory and will write the following content:

```
This is just one argument 10
```

Now let's see the content of the above file using the following program:

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

int vprintf(const char *format, va_list arg)

Description

The C library function **int vprintf(const char *format, va_list arg)** sends formatted output to stdout using an argument list passed to it.

Declaration

Following is the declaration for vprintf() function.

```
int vprintf(const char *format, va_list arg)
```

Parameters

- **format** -- This is the String that contains the text to be written to buffer. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and formatted as requested. Format tags prototype would be: **%[flags][width][.precision][length]specifier**, as explained below:

specifier	Output
c	Character
d or i	Signed decimal integer
e	Scientific notation (mantissa/exponent) using e character
E	Scientific notation (mantissa/exponent) using E character
f	Decimal floating point
g	Uses the shorter of %e or %f
G	Uses the shorter of %E or %f
o	Signed octal
s	String of characters
u	Unsigned decimal integer
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer (capital letters)
p	Pointer address
n	Nothing printed
%	Character
flags	**Description**
-	Left-justify within the given field width; Right justification is the default (see width sub-specifier).

+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a -ve sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceded with 0, 0x or 0X respectively for values different than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).
width	**Description**
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.
.precision	**Description**
.number	For integer specifiers (d, i, o, u, x, X): precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For e, E and f specifiers: this is the number of digits to be printed after the decimal point. For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. For c type: it has no effect. When no precision is specified, the default is 1. If the period is specified without an explicit value for precision, 0 is assumed.
.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

length	Description
h	The argument is interpreted as a short int or unsigned short int (only applies to integer specifiers: i, d, o, u, x and X).
l	The argument is interpreted as a long int or unsigned long int for integer specifiers (i, d, o, u, x and X), and as a wide character or wide character string for specifiers c and s.
L	The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g and G).

- **arg** -- An object representing the variable arguments list. This should be initialized by the va_start macro defined in <stdarg>.

Return Value

If successful, the total number of characters written is returned otherwise a negative number is returned.

Example

The following example shows the usage of vprintf() function.

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

void WriteFrmtd(char *format, ...)
{
   va_list args;

   va_start(args, format);
   vprintf(format, args);
   va_end(args);
}

int main ()
{
   WriteFrmtd("%d variable argument\n", 1);
   WriteFrmtd("%d variable %s\n", 2, "arguments");

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
1 variable argument
2 variable arguments
```

int vsprintf(char *str, const char *format, va_list arg)

Description

The C library function **int vsprintf(char *str, const char *format, va_list arg)** sends formatted output to a string using an argument list passed to it.

Declaration

Following is the declaration for vsprintf() function.

```
int vsprintf(char *str, const char *format, va_list arg)
```

Parameters

- **str** -- This is the array of char elements where the resulting string is to be stored.
- **format** -- This is the C string that contains the text to be written to the str. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and are formatted as requested. Format tags prototype: **%[flags][width][.precision][length]specifier**, as explained below:

specifier	Output
c	Character
d or i	Signed decimal integer
e	Scientific notation (mantissa/exponent) using e character
E	Scientific notation (mantissa/exponent) using E character
f	Decimal floating point
g	Uses the shorter of %e or %f
G	Uses the shorter of %E or %f
o	Signed octal
s	String of characters

u	Unsigned decimal integer
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer (capital letters)
p	Pointer address
n	Nothing printed
%	Character

flags	**Description**
-	Left-justify within the given field width; Right justification is the default (see width sub-specifier).
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a -ve sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceded with 0, 0x or 0X respectively for values different than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).

width	**Description**
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	Description
.number	For integer specifiers (d, i, o, u, x, X): precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For e, E and f specifiers: this is the number of digits to be printed after the decimal point. For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. For c type: it has no effect. When no precision is specified, the default is 1. If the period is specified without an explicit value for precision, 0 is assumed.
.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

length	Description
h	The argument is interpreted as a short int or unsigned short int (only applies to integer specifiers: i, d, o, u, x and X).
l	The argument is interpreted as a long int or unsigned long int for integer specifiers (i, d, o, u, x and X), and as a wide character or wide character string for specifiers c and s.
L	The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g and G).

- **arg** -- An object representing the variable arguments list. This should be initialized by the va_start macro defined in <stdarg>.

Return Value

If successful, the total number of characters written is returned, otherwise a negative number is returned.

Example

The following example shows the usage of vsprintf() function.

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

```
#include <stdarg.h>

char buffer[80];
int vspfunc(char *format, ...)
{
   va_list aptr;
   int ret;

   va_start(aptr, format);
   ret = vsprintf(buffer, format, aptr);
   va_end(aptr);

   return(ret);
}

int main()
{
   int i = 5;
   float f = 27.0;
   char str[50] = "tutoriaspoint.com";

   vspfunc("%d %f %s", i, f, str);
   printf("%s\n", buffer);

   return(0);
}
```

Let us compile and run the above program, this will produce the following result:

```
5 27.000000 tutoriaspoint.com
```

int fscanf(FILE *stream, const char *format, ...)

Description

The C library function **int fscanf(FILE *stream, const char *format, ...)** reads formatted input from a stream.

Declaration

Following is the declaration for fscanf() function.

```
int fscanf(FILE *stream, const char *format, ...)
```

Parameters

- **stream** -- This is the pointer to a FILE object that identifies the stream.
- **format** -- This is the C string that contains one or more of the following items: *Whitespace character, Non-whitespace character* and *Format specifiers*. A format specifier will be as **[=%[*][width][modifiers]type=]**, which is explained below:

argument	Description
*	This is an optional starting asterisk indicates that the data is to be read from the stream but ignored, i.e. it is not stored in the corresponding argument.
width	This specifies the maximum number of characters to be read in the current reading operation.
modifiers	Specifies a size different from int (in the case of d, i and n), unsigned int (in the case of o, u and x) or float (in the case of e, f and g) for the data pointed by the corresponding additional argument: h : short int (for d, i and n), or unsigned short int (for o, u and x) l : long int (for d, i and n), or unsigned long int (for o, u and x), or double (for e, f and g) L : long double (for e, f and g)
type	A character specifying the type of data to be read and how it is expected to be read. See next table.

fscanf type specifiers:

type	Qualifying Input	Type of argument
c	Single character: Reads the next character. If a width different from 1 is specified, the function reads width characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.	char *
d	Decimal integer: Number optionally preceded with a + or - sign	int *
e, E, f, g, G	Floating point: Decimal number containing a decimal point, optionally preceded by a + or - sign and optionally followed by the e or E character and a decimal number. Two examples of valid entries are -732.103 and 7.12e4	float *

o	Octal Integer:	int *
s	String of characters. This will read subsequent characters until a whitespace is found (whitespace characters are considered to be blank, newline and tab).	char *
u	Unsigned decimal integer.	unsigned int *
x, X	Hexadecimal Integer	int *

- **additional arguments** -- Depending on the format string, the function may expect a sequence of additional arguments, each containing one value to be inserted instead of each %-tag specified in the format parameter(if any). There should be the same number of these arguments as the number of %-tags that expect a value.

Return Value

This function returns the number of input items successfully matched and assigned, which can be fewer than provided for, or even zero in the event of an early matching failure.

Example

The following example shows the usage of fscanf() function.

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

int main()
{
   char str1[10], str2[10], str3[10];
   int year;
   FILE * fp;

   fp = fopen ("file.txt", "w+");
   fputs("We are in 2012", fp);

   rewind(fp);
   fscanf(fp, "%s %s %s %d", str1, str2, str3, &year);

   printf("Read String1 |%s|\n", str1 );
   printf("Read String2 |%s|\n", str2 );
```

```
   printf("Read String3 |%s|\n", str3 );
   printf("Read Integer |%d|\n", year );

   fclose(fp);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Read String1 |We|
Read String2 |are|
Read String3 |in|
Read Integer |2012|
```

int scanf(const char *format, ...)

Description

The C library function **int scanf(const char *format, ...)** reads formatted input from stdin.

Declaration

Following is the declaration for scanf() function.

```
int scanf(const char *format, ...)
```

Parameters

- **format** -- This is the C string that contains one or more of the following items:

  Whitespace character, Non-whitespace character and Format specifiers. A format specifier will be like **[=%[*][width][modifiers]type=]** as explained below:

argument	Description
*	This is an optional starting asterisk indicates that the data is to be read from the stream but ignored, i.e. it is not stored in the corresponding argument.
width	This specifies the maximum number of characters to be read in the current reading operation.
modifiers	Specifies a size different from int (in the case of d, i and n), unsigned int (in the case of o, u and x) or float (in the case of e, f and g) for the data pointed by the corresponding additional argument: h : short int (for d, i and n), or unsigned short int (for o, u and x) l : long int (for d, i and n),

	or unsigned long int (for o, u and x), or double (for e, f and g) L : long double (for e, f and g)
type	A character specifying the type of data to be read and how it is expected to be read. See next table.

fscanf type specifiers:

type	Qualifying Input	Type of argument
c	Single character: Reads the next character. If a width different from 1 is specified, the function reads width characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.	char *
d	Decimal integer: Number optionally preceded with a + or - sign	int *
e, E, f, g, G	Floating point: Decimal number containing a decimal point, optionally preceded by a + or - sign and optionally followed by the e or E character and a decimal number. Two examples of valid entries are -732.103 and 7.12e4	float *
o	Octal Integer:	int *
s	String of characters. This will read subsequent characters until a whitespace is found (whitespace characters are considered to be blank, newline and tab).	char *
u	Unsigned decimal integer.	unsigned int *
x, X	Hexadecimal Integer	int *

- **additional arguments** -- Depending on the format string, the function may expect a sequence of additional arguments, each containing one value to be inserted instead of each %-tag specified in the format parameter, if any. There should be the same number of these arguments as the number of %-tags that expect a value.

Return Value

If successful, the total number of characters written is returned, otherwise a negative number is returned.

Example

The following example shows the usage of scanf() function.

```
#include <stdio.h>

int main()
{
   char str1[20], str2[30];

   printf("Enter name: ");
   scanf("%s", &str1);

   printf("Enter your website name: ");
   scanf("%s", &str2);

   printf("Entered Name: %s\n", str1);
   printf("Entered Website:%s", str2);

   return(0);
}
```

Let us compile and run the above program that will produce the following result in interactive mode:

```
Enter name: admin
Enter your website name: www.tutorialspoint.com

Entered Name: admin
Entered Website: www.tutorialspoint.com
```

int sscanf(const char *str, const char *format, ...)

Description

The C library function **int sscanf(const char *str, const char *format, ...)** reads formatted input from a string.

Declaration

Following is the declaration for sscanf() function.

```
int sscanf(const char *str, const char *format, ...)
```

Parameters

- **str** -- This is the C string that the function processes as its source to retrieve the data.
- **format** -- This is the C string that contains one or more of the following items: *Whitespace character, Non-whitespace character and Format specifiers*

  A format specifier follows this prototype: [=%[*][width][modifiers]type=]

argument	Description
*	This is an optional starting asterisk, which indicates that the data is to be read from the stream but ignored, i.e. it is not stored in the corresponding argument.
width	This specifies the maximum number of characters to be read in the current reading operation.
modifiers	Specifies a size different from int (in the case of d, i and n), unsigned int (in the case of o, u and x) or float (in the case of e, f and g) for the data pointed by the corresponding additional argument: h : short int (for d, i and n), or unsigned short int (for o, u and x) l : long int (for d, i and n), or unsigned long int (for o, u and x), or double (for e, f and g) L : long double (for e, f and g)
type	A character specifying the type of data to be read and how it is expected to be read. See next table.

fscanf type specifiers:

type	Qualifying Input	Type of argument
c	Single character: Reads the next character. If a width different from 1 is specified, the function reads width characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.	char *
d	Decimal integer: Number optionally preceded with a + or - sign	int *
e, E, f, g, G	Floating point: Decimal number containing a decimal point, optionally preceded by a + or - sign and optionally followed by the e or E character and a decimal number. Two examples of valid entries are -732.103 and 7.12e4	float *
o	Octal Integer:	int *

s	String of characters. This will read subsequent characters until a whitespace is found (whitespace characters are considered to be blank, newline and tab).	char *
u	Unsigned decimal integer.	unsigned int *
x, X	Hexadecimal Integer	int *

- **other arguments** -- This function expects a sequence of pointers as additional arguments, each one pointing to an object of the type specified by their corresponding %-tag within the format string, in the same order.

  For each format specifier in the format string that retrieves data, an additional argument should be specified. If you want to store the result of a sscanf operation on a regular variable you should precede its identifier with the reference operator, i.e. an ampersand sign (&), like: int n; sscanf (str,"%d",&n);

Return Value

On success, the function returns the number of variables filled. In the case of an input failure before any data could be successfully read, EOF is returned.

Example

The following example shows the usage of sscanf() function.

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

int main()
{
   int day, year;
   char weekday[20], month[20], dtm[100];

   strcpy( dtm, "Saturday March 25 1989" );
   sscanf( dtm, "%s %s %d  %d", weekday, month, &day, &year );

   printf("%s %d, %d = %s\n", month, day, year, weekday );

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
March 25, 1989 = Saturday
```

int fgetc(FILE *stream)

Description

The C library function **int fgetc(FILE *stream)** gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream.

Declaration

Following is the declaration for fgetc() function.

```
int fgetc(FILE *stream)
```

Parameters

- **stream** -- This is the pointer to a FILE object that identifies the stream on which the operation is to be performed.

Return Value

This function returns the character read as an unsigned char cast to an int or EOF on end of file or error.

Example

The following example shows the usage of fgetc() function.

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;
   int n = 0;

   fp = fopen("file.txt","r");
   if(fp == NULL)
   {
      perror("Error in opening file");
      return(-1);
   }
   do
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
```

```
            break ;
         }
         printf("%c", c);
      }while(1);

      fclose(fp);
      return(0);
}
```

Let us assume, we have a text file **file.txt**, which has the following content. This file will be used as an input for our example program:

```
We are in 2012
```

Now, let us compile and run the above program that will produce the following result:

```
We are in 2012
```

char *fgets(char *str, int n, FILE *stream)

Description

The C library function **char *fgets(char *str, int n, FILE *stream)** reads a line from the specified stream and stores it into the string pointed to by **str**. It stops when either **(n-1)** characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.

Declaration

Following is the declaration for fgets() function.

```
char *fgets(char *str, int n, FILE *stream)
```

Parameters

- **str** -- This is the pointer to an array of chars where the string read is stored.
- **n** -- This is the maximum number of characters to be read (including the final null-character). Usually, the length of the array passed as str is used.
- **stream** -- This is the pointer to a FILE object that identifies the stream where characters are read from.

Return Value

On success, the function returns the same str parameter. If the End-of-File is encountered and no characters have been read, the contents of str remain unchanged and a null pointer is returned.

If an error occurs, a null pointer is returned.

Example

The following example shows the usage of fgets() function.

```
#include <stdio.h>

int main()
{
   FILE *fp;
   char str[60];

   /* opening file for reading */
   fp = fopen("file.txt" , "r");
   if(fp == NULL) {
      perror("Error opening file");
      return(-1);
   }
   if( fgets (str, 60, fp)!=NULL ) {
      /* writing content to stdout */
      puts(str);
   }
   fclose(fp);

   return(0);
}
```

Let us assume, we have a text file **file.txt**, which has the following content. This file will be used as an input for our example program:

```
We are in 2012
```

Now, let us compile and run the above program that will produce the following result:

```
We are in 2012
```

int fputc(int char, FILE *stream)

Description

The C library function **int fputc(int char, FILE *stream)** writes a character (an unsigned char) specified by the argument **char** to the specified stream and advances the position indicator for the stream.

Declaration

Following is the declaration for fputc() function.

```
int fputc(int char, FILE *stream)
```

Parameters

- **char** -- This is the character to be written. This is passed as its int promotion.
- **stream** -- This is the pointer to a FILE object that identifies the stream where the character is to be written.

Return Value

If there are no errors, the same character that has been written is returned. If an error occurs, EOF is returned and the error indicator is set.

Example

The following example shows the usage of fputc() function.

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int ch;

   fp = fopen("file.txt", "w+");
   for( ch = 33 ; ch <= 100; ch++ )
   {
      fputc(ch, fp);
   }
   fclose(fp);

   return(0);
}
```

Let us compile and run the above program that will create a file **file.txt** in the current directory, which will have following content:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcd
```

Now let's see the content of the above file using the following program:

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

```
int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

int fputs(const char *str, FILE *stream)

Description

The C library function **int fputs(const char *str, FILE *stream)** writes a string to the specified stream up to but not including the null character.

Declaration

Following is the declaration for fputs() function.

```
int fputs(const char *str, FILE *stream)
```

Parameters

- **str** -- This is an array containing the null-terminated sequence of characters to be written.
- **stream** -- This is the pointer to a FILE object that identifies the stream where the string is to be written.

Return Value

This function returns a non-negative value, or else on error it returns EOF.

Example

The following example shows the usage of fputs() function.

```
#include <stdio.h>
int main ()
{
   FILE *fp;

   fp = fopen("file.txt", "w+");

   fputs("This is c programming.", fp);
   fputs("This is a system programming language.", fp);

   fclose(fp);

   return(0);
}
```

Let us compile and run the above program, this will create a file **file.txt** with the following content:

```
This is c programming.This is a system programming language.
```

Now let's see the content of the above file using the following program:

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
```

```
   fclose(fp);
   return(0);
}
```

int getc(FILE *stream)

Description

The C library function **int getc(FILE *stream)** gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream.

Declaration

Following is the declaration for getc() function.

```
int getc(FILE *stream)
```

Parameters

- **stream** -- This is the pointer to a FILE object that identifies the stream on which the operation is to be performed.

Return Value

This function returns the character read as an unsigned char cast to an int or EOF on end of file or error.

Example

The following example shows the usage of getc() function.

```
#include<stdio.h>

int main()
{
   char c;

   printf("Enter character: ");
   c = getc(stdin);
   printf("Character entered: ");
   putc(c, stdout);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Enter character: a
```

```
Character entered: a
```

int getchar(void)

Description

The C library function **int getchar(void)** gets a character (an unsigned char) from stdin. This is equivalent to **getc** with stdin as its argument.

Declaration

Following is the declaration for getchar() function.

```
int getchar(void)
```

Parameters

- NA

Return Value

This function returns the character read as an unsigned char cast to an int or EOF on end of file or error.

Example

The following example shows the usage of getchar() function.

```
#include <stdio.h>

int main ()
{
   char c;
   printf("Enter character: ");
   c = getchar();

   printf("Character entered: ");
   putchar(c);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Enter character: a
Character entered: a
```

char *gets(char *str)

Description

The C library function **char *gets(char *str)** reads a line from stdin and stores it into the string pointed to by str. It stops when either the newline character is read or when the end-of-file is reached, whichever comes first.

Declaration

Following is the declaration for gets() function.

```
char *gets(char *str)
```

Parameters

- **str** -- This is the pointer to an array of chars where the C string is stored.

Return Value

This function returns str on success, and NULL on error or when end of file occurs, while no characters have been read.

Example

The following example shows the usage of gets() function.

```
#include <stdio.h>

int main()
{
   char str[50];

   printf("Enter a string : ");
   gets(str);

   printf("You entered: %s", str);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Enter a string : tutorialspoint.com
You entered: tutorialspoint.com
```

int putc(int char, FILE *stream)

Description

The C library function **int putc(int char, FILE *stream)** writes a character (an unsigned char) specified by the argument **char** to the specified stream and advances the position indicator for the stream.

Declaration

Following is the declaration for putc() function.

```
int putc(int char, FILE *stream)
```

Parameters

- **char** -- This is the character to be written. The character is passed as its int promotion.
- **stream** -- This is the pointer to a FILE object that identifies the stream where the character is to be written.

Return Value

This function returns the character written as an unsigned char cast to an int or EOF on error.

Example

The following example shows the usage of putc() function.

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int ch;

   fp = fopen("file.txt", "w");
   for( ch = 33 ; ch <= 100; ch++ )
   {
      putc(ch, fp);
   }
   fclose(fp);
   return(0);
}
```

Let us compile and run the above program that will create a file **file.txt** in the current directory which will have following content:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcd
```

Now let's see the content of the above file using the following program:

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

int putchar(int char)

Description

The C library function **int putchar(int char)** writes a character (an unsigned char) specified by the argument char to stdout.

Declaration

Following is the declaration for putchar() function.

```
int putchar(int char)
```

Parameters

- **char** -- This is the character to be written. This is passed as its int promotion.

Return Value

This function returns the character written as an unsigned char cast to an int or EOF on error.

Example

The following example shows the usage of putchar() function.

```
#include <stdio.h>

int main ()
{
   char ch;

   for(ch = 'A' ; ch <= 'Z' ; ch++) {
      putchar(ch);
   }
   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

int puts(const char *str)

Description

The C library function **int puts(const char *str)** writes a string to stdout up to but not including the null character. A newline character is appended to the output.

Declaration

Following is the declaration for puts() function.

```
int puts(const char *str)
```

Parameters

- **str** -- This is the C string to be written.

Return Value

If successful, non-negative value is returned. On error, the function returns EOF.

Example

The following example shows the usage of puts() function.

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

int main()
{
```

```
   char str1[15];
   char str2[15];

   strcpy(str1, "tutorialspoint");
   strcpy(str2, "compileonline");

   puts(str1);
   puts(str2);

   return(0);
}
```

Let us compile and run the above program to produce the following result:

```
tutorialspoint
compileonline
```

int ungetc(int char, FILE *stream)

Description

The C library function **int ungetc(int char, FILE *stream)** pushes the character **char (an unsigned char)** onto the specified **stream** so that the this is available for the next read operation.

Declaration

Following is the declaration for ungetc() function.

```
int ungetc(int char, FILE *stream)
```

Parameters

- **char** -- This is the character to be put back. This is passed as its int promotion.
- **stream** -- This is the pointer to a FILE object that identifies an input stream.

Return Value

If successful, it returns the character that was pushed back otherwise, EOF is returned and the stream remains unchanged.

Example

The following example shows the usage of ungetc() function.

```
#include <stdio.h>
int main ()
{
   FILE *fp;
```

```
   int c;
   char buffer [256];

   fp = fopen("file.txt", "r");
   if( fp == NULL )
   {
      perror("Error in opening file");
      return(-1);
   }
   while(!feof(fp))
   {
      c = getc (fp);
      /* replace ! with + */
      if( c == '!' )
      {
         ungetc ('+', fp);
      }
      else
      {
         ungetc(c, fp);
      }
      fgets(buffer, 255, fp);
      fputs(buffer, stdout);
   }
   return(0);
}
```

Let us assume, we have a text file **file.txt**, which contains the following data. This file will be used as an input for our example program:

```
this is tutorials point
!c standard library
!library functions and macros
```

Now let us compile and run the above program that will produce the following result:

```
this is tutorials point
+c standard library
+library functions and macros
```

void perror(const char *str)

Description

The C library function **void perror(const char *str)** prints a descriptive error message to stderr. First the string **str** is printed, followed by a colon then a space.

Declaration

Following is the declaration for perror() function.

```
void perror(const char *str)
```

Parameters

- **str** -- This is the C string containing a custom message to be printed before the error message itself.

Return Value

This function does not return any value.

Example

The following example shows the usage of perror() function.

```
#include <stdio.h>

int main ()
{
   FILE *fp;

   /* first rename if there is any file */
   rename("file.txt", "newfile.txt");

   /* now let's try to open same file */
   fp = fopen("file.txt", "r");
   if( fp == NULL ) {
      perror("Error: ");
      return(-1);
   }
   fclose(fp);
   return(0);
}
```

Let us compile and run the above program that will produce the following result because we are trying to open a file which does not exist:

```
Error: : No such file or directory
```

13. C Library – <stdlib.h>

Introduction

The **stdlib.h** header defines four variable types, several macros, and various functions for performing general functions.

Library Variables

Following are the variable types defined in the header stdlib.h:

S.N.	Variable & Description
1	**size_t** This is the unsigned integral type and is the result of the **sizeof** keyword.
2	**wchar_t** This is an integer type of the size of a **wide** character constant.
3	**div_t** This is the structure returned by the **div** function.
4	**ldiv_t** This is the structure returned by the **ldiv** function.

Library Macros

Following are the macros defined in the header stdlib.h:

S.N.	Macro & Description
1	**NULL** This macro is the value of a null pointer constant.
2	**EXIT_FAILURE** This is the value for the exit function to return in case of failure.
3	**EXIT_SUCCESS**

	This is the value for the exit function to return in case of success.
4	**RAND_MAX** This macro is the maximum value returned by the **rand** function.
5	**MB_CUR_MAX** This macro is the maximum number of bytes in a multi-byte character set which cannot be larger than MB_LEN_MAX.

Library Functions

Following are the functions defined in the header stdio.h:

S.N.	Function & Description
1	double atof(const char *str) Converts the string pointed to, by the argument *str* to a floating-point number (type double).
2	int atoi(const char *str) Converts the string pointed to, by the argument *str* to an integer (type int).
3	long int atol(const char *str) Converts the string pointed to, by the argument *str* to a long integer (type long int).
4	double strtod(const char *str, char **endptr) Converts the string pointed to, by the argument *str* to a floating-point number (type double).
5	long int strtol(const char *str, char **endptr, int base) Converts the string pointed to, by the argument *str* to a long integer (type long int).
6	unsigned long int strtoul(const char *str, char **endptr, int base) Converts the string pointed to, by the argument *str* to an unsigned long integer (type unsigned long int).
7	void *calloc(size_t nitems, size_t size)

	Allocates the requested memory and returns a pointer to it.
8	void free(void *ptr Deallocates the memory previously allocated by a call to *calloc, malloc,* or*realloc*.
9	void *malloc(size_t size) Allocates the requested memory and returns a pointer to it.
10	void *realloc(void *ptr, size_t size) Attempts to resize the memory block pointed to by ptr that was previously allocated with a call to *malloc* or *calloc*.
11	void abort(void) Causes an abnormal program termination.
12	int atexit(void (*func)(void)) Causes the specified function **func** to be called when the program terminates normally.
13	void exit(int status) Causes the program to terminate normally.
14	char *getenv(const char *name) Searches for the environment string pointed to by name and returns the associated value to the string.
15	int system(const char *string) The command specified by string is passed to the host environment to be executed by the command processor.
16	void *bsearch(const void *key, const void *base, size_t nitems, size_t size, int (*compar)(const void *, const void *)) Performs a binary search.
17	void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*)) Sorts an array.

18	int abs(int x) Returns the absolute value of x.
19	div_t div(int numer, int denom) Divides numer (numerator) by denom (denominator).
20	long int labs(long int x) Returns the absolute value of x.
21	ldiv_t ldiv(long int numer, long int denom) Divides numer (numerator) by denom (denominator).
22	int rand(void) Returns a pseudo-random number in the range of 0 to *RAND_MAX*.
23	void srand(unsigned int seed) This function seeds the random number generator used by the function **rand**.
24	int mblen(const char *str, size_t n) Returns the length of a multibyte character pointed to by the argument *str*.
25	size_t mbstowcs(schar_t *pwcs, const char *str, size_t n) Converts the string of multibyte characters pointed to by the argument *str* to the array pointed to by *pwcs*.
26	int mbtowc(whcar_t *pwc, const char *str, size_t n) Examines the multibyte character pointed to by the argument *str*.
27	size_t wcstombs(char *str, const wchar_t *pwcs, size_t n) Converts the codes stored in the array *pwcs* to multibyte characters and stores them in the string *str*.
28	int wctomb(char *str, wchar_t wchar) Examines the code which corresponds to a multibyte character given by the argument *wchar*.

double atof(const char *str)

Description

The C library function **double atof(const char *str)** converts the string argument **str** to a floating-point number (type double).

Declaration

Following is the declaration for atof() function.

```
double atof(const char *str)
```

Parameters

- **str** -- This is the string having the representation of a floating-point number.

Return Value

This function returns the converted floating point number as a double value. If no valid conversion could be performed, it returns zero (0.0).

Example

The following example shows the usage of atof() function.

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

int main()
{
   float val;
   char str[20];

   strcpy(str, "98993489");
   val = atof(str);
   printf("String value = %s, Float value = %f\n", str, val);

   strcpy(str, "tutorialspoint.com");
   val = atof(str);
   printf("String value = %s, Float value = %f\n", str, val);
   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
String value = 98993489, Float value = 98993488.000000
String value = tutorialspoint.com, Float value = 0.000000
```

int atoi(const char *str)

Description

The C library function **int atoi(const char *str)** converts the string argument **str** to an integer (type int).

Declaration

Following is the declaration for atoi() function.

```
int atoi(const char *str)
```

Parameters

- **str** -- This is the string representation of an integral number.

Return Value

This function returns the converted integral number as an int value. If no valid conversion could be performed, it returns zero.

Example

The following example shows the usage of atoi() function.

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

int main()
{
   int val;
   char str[20];

   strcpy(str, "98993489");
   val = atoi(str);
   printf("String value = %s, Int value = %d\n", str, val);

   strcpy(str, "tutorialspoint.com");
   val = atoi(str);
   printf("String value = %s, Int value = %d\n", str, val);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
String value = 98993489, Int value = 98993489
String value = tutorialspoint.com, Int value = 0
```

long int atol(const char *str)

Description

The C library function **long int atol(const char *str)** converts the string argument **str** to a long integer (type long int).

Declaration

Following is the declaration for atol() function.

```
long int atol(const char *str)
```

Parameters

- **str** -- This is the string containing the representation of an integral number.

Return Value

This function returns the converted integral number as a long int. If no valid conversion could be performed, it returns zero.

Example

The following example shows the usage of atol() function.

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

int main()
{
   long val;
   char str[20];

   strcpy(str, "98993489");
   val = atol(str);
   printf("String value = %s, Long value = %ld\n", str, val);

   strcpy(str, "tutorialspoint.com");
   val = atol(str);
   printf("String value = %s, Long value = %ld\n", str, val);
```

```
   return(0);
}
```

Let us compile and run the above program, this will produce the following result:

```
String value = 98993489, Long value = 98993489
String value = tutorialspoint.com, Long value = 0
```

double strtod(const char *str, char **endptr)

Description

The C library function **double strtod(const char *str, char **endptr)** converts the string pointed to by the argument **str** to a floating-point number (type double). If **endptr** is not NULL, a pointer to the character after the last character used in the conversion is stored in the location referenced by endptr.

Declaration

Following is the declaration for strtod() function.

```
double strtod(const char *str, char **endptr)
```

Parameters

- **str** -- This is the value to be converted to a string.
- **endptr** -- This is the reference to an already allocated object of type char*, whose value is set by the function to the next character in *str* after the numerical value.

Return Value

This function returns the converted floating point number as a double value, else zero value (0.0) is returned.

Example

The following example shows the usage of strtod() function.

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

int main()
{
  char str[30] = "20.30300 This is test";
   char *ptr;
   double ret;

   ret = strtod(str, &ptr);
```

```
   printf("The number(double) is %lf\n", ret);
   printf("String part is |%s|", ptr);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
The number(double) is 20.303000
String part is | This is test|
```

long int strtol(const char *str, char **endptr, int base)

Description

The C library function **long int strtol(const char *str, char **endptr, int base)** converts the initial part of the string in **str** to a **long int** value according to the given **base**, which must be between 2 and 36 inclusive, or be the special value 0.

Declaration

Following is the declaration for strtol() function.

```
long int strtol(const char *str, char **endptr, int base)
```

Parameters

- **str** -- This is the string containing the representation of an integral number.
- **endptr** -- This is the reference to an object of type char*, whose value is set by the function to the next character in *str* after the numerical value.
- **base** -- This is the base, which must be between 2 and 36 inclusive, or be the special value 0.

Return Value

This function returns the converted integral number as a long int value, else zero value is returned.

Example

The following example shows the usage of strtol() function.

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

int main()
{
   char str[30] = "2030300 This is test";
   char *ptr;
```

```
   long ret;

   ret = strtol(str, &ptr, 10);
   printf("The number(unsigned long integer) is %ld\n", ret);
   printf("String part is |%s|", ptr);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
The number(unsigned long integer) is 2030300
String part is | This is test|
```

unsigned long int strtoul(const char *str, char **endptr, int base)

Description

The C library function **unsigned long int strtoul(const char *str, char **endptr, int base)** function converts the initial part of the string in **str** to an unsigned long int value according to the given **base**, which must be between 2 and 36 inclusive, or be the special value 0.

Declaration

Following is the declaration for strtoul() function.

```
unsigned long int strtoul(const char *str, char **endptr, int base)
```

Parameters

- **str** -- This is the string containing the representation of an unsigned integral number.
- **endptr** -- This is the reference to an object of type char*, whose value is set by the function to the next character in str after the numerical value.
- **base** -- This is the base, which must be between 2 and 36 inclusive, or be the special value 0.

Return Value

This function returns the converted integral number as a long int value. If no valid conversion could be performed, a zero value is returned.

Example

The following example shows the usage of strtoul() function.

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

int main()
{
   char str[30] = "2030300 This is test";
   char *ptr;
   long ret;

   ret = strtoul(str, &ptr, 10);
   printf("The number(unsigned long integer) is %lu\n", ret);
   printf("String part is |%s|", ptr);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
The number(unsigned long integer) is 2030300
String part is | This is test|
```

void *calloc(size_t nitems, size_t size)

Description

The C library function **void *calloc(size_t nitems, size_t size)** allocates the requested memory and returns a pointer to it. The difference in **malloc** and **calloc** is that malloc does not set the memory to zero whereas calloc sets allocated memory to zero.

Declaration

Following is the declaration for calloc() function.

```
void *calloc(size_t nitems, size_t size)
```

Parameters

- **nitems** -- This is the number of elements to be allocated.
- **size** -- This is the size of elements.

Return Value

This function returns a pointer to the allocated memory, or NULL if the request fails.

Example

The following example shows the usage of calloc() function.

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

int main()
{
   int i, n;
   int *a;

   printf("Number of elements to be entered:");
   scanf("%d",&n);

   a = (int*)calloc(n, sizeof(int));
   printf("Enter %d numbers:\n",n);
   for( i=0 ; i < n ; i++ )
   {
      scanf("%d",&a[i]);
   }

   printf("The numbers entered are: ");
   for( i=0 ; i < n ; i++ ) {
      printf("%d ",a[i]);
   }

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Number of elements to be entered:3
Enter 3 numbers:
22
55
14
The numbers entered are: 22 55 14
```

void free(void *ptr)

Description

The C library function **void free(void *ptr)** deallocates the memory previously allocated by a call to calloc, malloc, or realloc.

Declaration

Following is the declaration for free() function.

```
void free(void *ptr)
```

Parameters

- **ptr** -- This is the pointer to a memory block previously allocated with malloc, calloc or realloc to be deallocated. If a null pointer is passed as argument, no action occurs.

Return Value

This function does not return any value.

Example

The following example shows the usage of free() function.

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

int main()
{
   char *str;

   /* Initial memory allocation */
   str = (char *) malloc(15);
   strcpy(str, "tutorialspoint");
   printf("String = %s,  Address = %u\n", str, str);

   /* Reallocating memory */
   str = (char *) realloc(str, 25);
   strcat(str, ".com");
   printf("String = %s,  Address = %u\n", str, str);

   /* Deallocate allocated memory */
   free(str);
   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
String = tutorialspoint, Address = 355090448
```

```
String = tutorialspoint.com, Address = 355090448
```

void *malloc(size_t size)

Description

The C library function **void *malloc(size_t size)** allocates the requested memory and returns a pointer to it.

Declaration

Following is the declaration for malloc() function.

```
void *malloc(size_t size)
```

Parameters

- **size** -- This is the size of the memory block, in bytes.

Return Value

This function returns a pointer to the allocated memory, or NULL if the request fails.

Example

The following example shows the usage of malloc() function.

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

int main()
{
   char *str;

   /* Initial memory allocation */
   str = (char *) malloc(15);
   strcpy(str, "tutorialspoint");
   printf("String = %s,  Address = %u\n", str, str);

   /* Reallocating memory */
   str = (char *) realloc(str, 25);
   strcat(str, ".com");
   printf("String = %s,  Address = %u\n", str, str);

   free(str);

   return(0);
```

```
}
```

Let us compile and run the above program that will produce the following result:

```
String = tutorialspoint, Address = 355090448
String = tutorialspoint.com, Address = 355090448
```

void *realloc(void *ptr, size_t size)

Description

The C library function **void *realloc(void *ptr, size_t size)** attempts to resize the memory block pointed to by **ptr** that was previously allocated with a call to **malloc** or **calloc**.

Declaration

Following is the declaration for realloc() function.

```
void *realloc(void *ptr, size_t size)
```

Parameters

- **ptr** -- This is the pointer to a memory block previously allocated with malloc, calloc or realloc to be reallocated. If this is NULL, a new block is allocated and a pointer to it is returned by the function.
- **size** -- This is the new size for the memory block, in bytes. If it is 0 and ptr points to an existing block of memory, the memory block pointed by ptr is deallocated and a NULL pointer is returned.

Return Value

This function returns a pointer to the newly allocated memory, or NULL if the request fails.

Example

The following example shows the usage of realloc() function.

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

int main()
{
   char *str;

   /* Initial memory allocation */
   str = (char *) malloc(15);
   strcpy(str, "tutorialspoint");
   printf("String = %s,  Address = %u\n", str, str);
```

```
   /* Reallocating memory */
   str = (char *) realloc(str, 25);
   strcat(str, ".com");
   printf("String = %s,  Address = %u\n", str, str);

   free(str);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
String = tutorialspoint, Address = 355090448
String = tutorialspoint.com, Address = 355090448
```

void abort(void)

Description

The C library function **void abort(void)** aborts the program execution and comes out directly from the place of the call.

Declaration

Following is the declaration for abort() function.

```
void abort(void)
```

Parameters

- NA

Return Value

This function does not return any value.

Example

The following example shows the usage of abort() function.

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

int main ()
{
   FILE *fp;
```

```
   printf("Going to open nofile.txt\n");
   fp = fopen( "nofile.txt","r" );
   if(fp == NULL)
   {
      printf("Going to abort the program\n");
      abort();
   }
   printf("Going to close nofile.txt\n");
   fclose(fp);

   return(0);
}
```

Let us compile and run the above program that will produce the following result when it tries to open **nofile.txt** file, which does not exist:

```
Going to open nofile.txt
Going to abort the program
Aborted (core dumped)
```

int atexit(void (*func)(void))

Description

The C library function **int atexit(void (*func)(void))** causes the specified function **func** to be called when the program terminates. You can register your termination function anywhere you like, but it will be called at the time of the program termination.

Declaration

Following is the declaration for atexit() function.

```
int atexit(void (*func)(void))
```

Parameters

- **func** -- This is the function to be called at the termination of the program.

Return Value

This function returns a zero value if the function is registered successfully, otherwise a non-zero value is returned if it is failed.

Example

The following example shows the usage of atexit() function.

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

```
#include <stdlib.h>

void functionA ()
{
   printf("This is functionA\n");
}

int main ()
{
   /* register the termination function */
   atexit(functionA );

   printf("Starting  main program...\n");

   printf("Exiting main program...\n");

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Starting main program...
Exiting main program...
This is functionA
```

void exit(int status)

Description

The C library function **void exit(int status)** terminates the calling process immediately. Any open file descriptors belonging to the process are closed and any children of the process are inherited by process 1, init, and the process parent is sent a SIGCHLD signal.

Declaration

Following is the declaration for exit() function.

```
void exit(int status)
```

Parameters

- **status** -- This is the status value returned to the parent process.

Return Value

This function does not return any value.

Example

The following example shows the usage of exit() function.

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

int main ()
{
   printf("Start of the program....\n");

   printf("Exiting the program....\n");
   exit(0);

   printf("End of the program....\n");

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Start of the program....
Exiting the program....
```

char *getenv(const char *name)

Description

The C library function **char *getenv(const char *name)** searches for the environment string pointed to, by **name** and returns the associated value to the string.

Declaration

Following is the declaration for getenv() function.

```
char *getenv(const char *name)
```

Parameters

- **name** -- This is the C string containing the name of the requested variable.

Return Value

This function returns a null-terminated string with the value of the requested environment variable, or NULL if that environment variable does not exist.

Example

The following example shows the usage of getenv() function.

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

int main ()
{
   printf("PATH : %s\n", getenv("PATH"));
   printf("HOME : %s\n", getenv("HOME"));
   printf("ROOT : %s\n", getenv("ROOT"));

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
PATH : /sbin:/usr/sbin:/bin:/usr/bin:/usr/local/bin
HOME : /
ROOT : (null)
```

int system(const char *string)

Description

The C library function **int system(const char *command)** passes the command name or program name specified by **command** to the host environment to be executed by the command processor and returns after the command has been completed.

Declaration

Following is the declaration for system() function.

```
int system(const char *command)
```

Parameters

- **command** -- This is the C string containing the name of the requested variable.

Return Value

The value returned is -1 on error, and the return status of the command otherwise.

Example

The following example shows the usage of system() function to list down all the files and directories in the current directory under unix machine.

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

int main ()
{
   char command[50];

   strcpy( command, "ls -l" );
   system(command);

   return(0);
}
```

Let us compile and run the above program that will produce the following result on my unix machine:

```
drwxr-xr-x 2 apache apache 4096 Aug 22 07:25 hsperfdata_apache
drwxr-xr-x 2 railo railo 4096 Aug 21 18:48 hsperfdata_railo
rw------ 1 apache apache 8 Aug 21 18:48 mod_mono_dashboard_XXGLOBAL_1
rw------ 1 apache apache 8 Aug 21 18:48 mod_mono_dashboard_asp_2
srwx---- 1 apache apache 0 Aug 22 05:28 mod_mono_server_asp
rw------ 1 apache apache 0 Aug 22 05:28 mod_mono_server_asp_1280495620
srwx---- 1 apache apache 0 Aug 21 18:48 mod_mono_server_global
```

The following example shows the usage of system() function to list down all the files and directories in the current directory under windows machine.

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

int main ()
{
   char command[50];

   strcpy( command, "dir" );
   system(command);

   return(0);
}
```

Let us compile and run the above program that will produce the following result on my windows machine:

```
a.txt
amit.doc
sachin
saurav
file.c
```

void *bsearch(const void *key, const void *base, size_t nitems, size_t size, int (*compar)(const void *, const void *))

Description

The C library function void *bsearch(const void *key, const void *base, size_t nitems, size_t size, int (*compar)(const void *, const void *)) function searches an array of **nitems** objects, the initial member of which is pointed to by base, for a member that matches the object pointed to, by key. The size of each member of the array is specified by size.

The contents of the array should be in ascending sorted order according to the comparison function referenced by **compar**.

Declaration

Following is the declaration for bsearch() function.

```
void *bsearch(const void *key, const void *base, size_t nitems, size_t size,
int (*compar)(const void *, const void *))
```

Parameters

- **key** -- This is the pointer to the object that serves as key for the search, type-casted as a void*.
- **base** -- This is the pointer to the first object of the array where the search is performed, type-casted as a void*.
- **nitems** -- This is the number of elements in the array pointed by base.
- **size** -- This is the size in bytes of each element in the array.
- **compar** -- This is the function that compares two elements.

Return Value

This function returns a pointer to an entry in the array that matches the search key. If key is not found, a NULL pointer is returned.

Example

The following example shows the usage of bsearch() function.

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

```
#include <stdlib.h>


int cmpfunc(const void * a, const void * b)
{
   return ( *(int*)a - *(int*)b );
}

int values[] = { 5, 20, 29, 32, 63 };

int main ()
{
   int *item;
   int key = 32;

   /* using bsearch() to find value 32 in the array */
   item = (int*) bsearch (&key, values, 5, sizeof (int), cmpfunc);
   if( item != NULL )
   {
      printf("Found item = %d\n", *item);
   }
   else
   {
      printf("Item = %d could not be found\n", *item);
   }

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Found item = 32
```

void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))

Description

The C library function void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*)) sorts an array.

Declaration

Following is the declaration for qsort() function.

```
void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *,
const void*))
```

Parameters

- **base** -- This is the pointer to the first element of the array to be sorted.
- **nitems** -- This is the number of elements in the array pointed by base.
- **size** -- This is the size in bytes of each element in the array.
- **compar** -- This is the function that compares two elements.

Return Value

This function does not return any value.

Example

The following example shows the usage of qsort() function.

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

int values[] = { 88, 56, 100, 2, 25 };

int cmpfunc (const void * a, const void * b)
{
   return ( *(int*)a - *(int*)b );
}

int main()
{
   int n;

   printf("Before sorting the list is: \n");
   for( n = 0 ; n < 5; n++ ) {
      printf("%d ", values[n]);
   }

   qsort(values, 5, sizeof(int), cmpfunc);

   printf("\nAfter sorting the list is: \n");
   for( n = 0 ; n < 5; n++ ) {
```

```
      printf("%d ", values[n]);
   }

  return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Before sorting the list is:
88 56 100 2 25
After sorting the list is:
2 25 56 88 100
```

int abs(int x)

Description

The C library function **int abs(int x)** returns the absolute value of **int x**.

Declaration

Following is the declaration for abs() function.

```
int abs(int x)
```

Parameters

- **x** -- This is the integral value.

Return Value

This function returns the absolute value of x.

Example

The following example shows the usage of abs() function.

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

int main ()
{
   int a, b;

   a = abs(5);
   printf("value of a = %d\n", a);

   b = abs(-10);
```

```
   printf("value of b = %d\n", b);

   return(0);
}
```

Let us compile and run the above program, this will produce the following result:

```
value of a = 5
value of b = 10
```

div_t div(int numer, int denom)

Description

The C library function div_t div(int numer, int denom) divides numer (numerator) by denom (denominator).

Declaration

Following is the declaration for div() function.

```
div_t div(int numer, int denom)
```

Parameters

- **numer** -- This is the numerator.
- **denom** -- This is the denominator.

Return Value

This function returns the value in a structure defined in <cstdlib>, which has two members. For div_t: *int quot; int rem;*

Example

The following example shows the usage of div() function.

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

int main()
{
   div_t output;

   output = div(27, 4);
   printf("Quotient part of (27/ 4) = %d\n", output.quot);
   printf("Remainder part of (27/4) = %d\n", output.rem);
```

```
   output = div(27, 3);
   printf("Quotient part of (27/ 3) = %d\n", output.quot);
   printf("Remainder part of (27/3) = %d\n", output.rem);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Quotient part of (27/ 4) = 6
Remainder part of (27/4) = 3
Quotient part of (27/ 3) = 9
Remainder part of (27/3) = 0
```

long int labs(long int x)

Description

The C library function **long int labs(long int x)** returns the absolute value of **x**.

Declaration

Following is the declaration for labs() function.

```
long int labs(long int x)
```

Parameters

- **x** -- This is the integral value.

Return Value

This function returns the absolute value of **x**.

Example

The following example shows the usage of labs() function.

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

int main ()
{
   long int a,b;

   a = labs(65987L);
   printf("Value of a = %ld\n", a);
```

```
   b = labs(-1005090L);
   printf("Value of b = %ld\n", b);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Value of a = 65987
Value of b = 1005090
```

ldiv_t ldiv(long int numer, long int denom)

Description

The C library function div_t div(long int numer, long int denom) divides numer (numerator) by denom (denominator).

Declaration

Following is the declaration for ldiv() function.

```
div_t div(long int numer, long int denom)
```

Parameters

- **numer** -- This is the numerator.
- **denom** -- This is the denominator.

Return Value

This function returns the value in a structure defined in <cstdlib>, which has two members. For ldiv_t: *long quot; long rem;*

Example

The following example shows the usage of ldiv() function.

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

int main ()
{
   ldiv_t output;

   output = ldiv(100000L, 30000L);

   printf("Quotient = %ld\n", output.quot);
```

```
   printf("Remainder = %ld\n", output.rem);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Quotient = 3
Remainder = 10000
```

int rand(void)

Description

The C library function **int rand(void)** returns a pseudo-random number in the range of 0 to *RAND_MAX*.

RAND_MAX is a constant whose default value may vary between implementations but it is granted to be at least 32767.

Declaration

Following is the declaration for rand() function.

```
int rand(void)
```

Parameters

- NA

Return Value

This function returns an integer value between 0 and RAND_MAX.

Example

The following example shows the usage of rand() function.

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

int main()
{
   int i, n;
   time_t t;

   n = 5;
```

```
   /* Intializes random number generator */
   srand((unsigned) time(&t));

   /* Print 5 random numbers from 0 to 49 */
   for( i = 0 ; i < n ; i++ ) {
      printf("%d\n", rand() % 50);
   }

  return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
38
45
29
29
47
```

void srand(unsigned int seed)

Description

The C library function **void srand(unsigned int seed)** seeds the random number generator used by the function **rand**.

Declaration

Following is the declaration for srand() function.

```
void srand(unsigned int seed)
```

Parameters

- **seed** -- This is an integer value to be used as seed by the pseudo-random number generator algorithm.

Return Value

This function does not return any value.

Example

The following example shows the usage of srand() function.

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

```
int main()
{
   int i, n;
   time_t t;

   n = 5;

   /* Intializes random number generator */
   srand((unsigned) time(&t));

   /* Print 5 random numbers from 0 to 50 */
   for( i = 0 ; i < n ; i++ ) {
      printf("%d\n", rand() % 50);
   }

  return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
38
45
29
29
47
```

int mblen(const char *str, size_t n)

Description

The C library function **int mblen(const char *str, size_t n)** returns the length of a multi-byte character pointed to, by the argument **str**.

Declaration

Following is the declaration for mblen() function.

```
int mblen(const char *str, size_t n)
```

Parameters

- **str** -- This is the pointer to the first byte of a multibyte character.
- **n** -- This is the maximum number of bytes to be checked for character length.

Return Value

The mblen() function returns the number of bytes passed from the multi-byte sequence starting at str, if a non-null wide character was recognized. It returns 0, if a null wide character was recognized. It returns -1, if an invalid multi-byte sequence was encountered or if it could not parse a complete multi-byte character.

Example

The following example shows the usage of mblen() function.

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

int main()
{
   int len;
   char *pmbnull  = NULL;
   char *pmb = (char *)malloc( MB_CUR_MAX );
   wchar_t *pwc = L"Hi";
   wchar_t *pwcs = (wchar_t *)malloc( sizeof( wchar_t ));

   printf("Converting to multibyte string\n");
   len = wcstombs( pmb, pwc, MB_CUR_MAX);
   printf("Characters converted %d\n", len);
   printf("Hex value of first multibyte character: %#.4x\n", pmb);

   len = mblen( pmb, MB_CUR_MAX );
   printf( "Length in bytes of multibyte character %x: %u\n", pmb, len );

   pmb = NULL;

   len = mblen( pmb, MB_CUR_MAX );
   printf( "Length in bytes of multibyte character %x: %u\n", pmb, len );

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Converting to multibyte string
Characters converted 1
```

```
Hex value of first multibyte character: 0x168c6010
Length in bytes of multibyte character 168c6010: 1
Length in bytes of multibyte character 0: 0
```

size_t mbstowcs(schar_t *pwcs, const char *str, size_t n)

Description

The C library function **size_t mbstowcs(schar_t *pwcs, const char *str, size_t n)** converts the string of multi-byte characters pointed to, by the argument **str** to the array pointed to by **pwcs**.

Declaration

Following is the declaration for mbstowcs() function.

```
size_t mbstowcs(schar_t *pwcs, const char *str, size_t n)
```

Parameters

- **pwcs** -- This is the pointer to an array of wchar_t elements that is long enough to store a wide string max characters long.
- **str** -- This is the C multibyte character string to be interpreted.
- **n** -- This is the maximum number of wchar_t characters to be interpreted.

Return Value

This function returns the number of characters translated, excluding the ending null-character. If an invalid multi-byte character is encountered, a -1 value is returned.

Example

The following example shows the usage of mbstowcs() function.

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

int main()
{
   int len;
   char *pmbnull  = NULL;
   char *pmb = (char *)malloc( MB_CUR_MAX );
   wchar_t *pwc = L"Hi";
   wchar_t *pwcs = (wchar_t *)malloc( sizeof( wchar_t ));

   printf("Converting to multibyte string\n");
   len = wcstombs( pmb, pwc, MB_CUR_MAX);
```

```
   printf("Characters converted %d\n", len);
   printf("Hex value of first multibyte character: %#.4x\n", pmb);

   printf("Converting back to Wide-Character string\n");
   len = mbstowcs( pwcs, pmb, MB_CUR_MAX);
   printf("Characters converted %d\n", len);
   printf("Hex value of first wide character %#.4x\n\n", pwcs);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Converting to multibyte string
Characters converted 1
Hex value of first multibyte character: 0x19a60010
Converting back to Wide-Character string
Characters converted 1
Hex value of first wide character 0x19a60030
```

int mbtowc(whcar_t *pwc, const char *str, size_t n)

Description

The C library function **int mbtowc(whcar_t *pwc, const char *str, size_t n)** converts a multibyte sequence to a wide character.

Declaration

Following is the declaration for mbtowc() function.

```
int mbtowc(whcar_t *pwc, const char *str, size_t n)
```

Parameters

- **pwc** -- This is the pointer to an object of type wchar_t.
- **str** -- This is the pointer to the first byte of a multi-byte character.
- **n** -- This is the maximum number of bytes to be checked for character length.

Return Value

- If str is not NULL, the mbtowc() function returns the number of consumed bytes starting at str, or 0 if **s** points to a null byte, or -1 upon failure.
- If str is NULL, the mbtowc() function returns non-zero if the encoding has non-trivial shift state, or zero if the encoding is stateless.

Example

The following example shows the usage of mbtowc() function.

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

int main()
{
   char *str = "This is tutorialspoint.com";
   wchar_t mb[100];
   int len;

   len = mblen(NULL, MB_CUR_MAX);

   mbtowc(mb, str, len*strlen(str) );

   wprintf(L"%ls \n", mb );

   return(0);
}
```

Let us compile and run the above program that will produce the following result which will be in multi-byte, a kind of binary output.

```
???
```

size_t wcstombs(char *str, const wchar_t *pwcs, size_t n)

Description

The C library function **size_t wcstombs(char *str, const wchar_t *pwcs, size_t n)** converts the wide-character string **pwcs** to a multibyte string starting at **str**. At most **n** bytes are written to **str**.

Declaration

Following is the declaration for wcstombs() function.

```
size_t wcstombs(char *str, const wchar_t *pwcs, size_t n)
```

Parameters

- **str** -- This is the pointer to an array of char elements at least n bytes long.
- **pwcs** -- This is wide-character string to be converted.
- **n** -- This is the maximum number of bytes to be written to str.

Return Value

This function returns the number of bytes (not characters) converted and written to str, excluding the ending null-character. If an invalid multibyte character is encountered, -1 value is returned.

Example

The following example shows the usage of wcstombs() function.

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

#define BUFFER_SIZE 50

int main()
{
   size_t ret;
   char *MB = (char *)malloc( BUFFER_SIZE );
   wchar_t *WC = L"http://www.tutorialspoint.com";

   /* converting wide-character string */
   ret = wcstombs(MB, WC, BUFFER_SIZE);

   printf("Characters converted = %u\n", ret);
   printf("Multibyte character = %s\n\n", MB);
   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Characters converted = 29
Multibyte character = http://www.tutorialspoint.com
```

int wctomb(char *str, wchar_t wchar)

Description

The C library function **int wctomb(char *str, wchar_t wchar)** function converts the wide character **wchar** to its multibyte representation and stores it at the beginning of the character array pointed to by **str**.

Declaration

Following is the declaration for wctomb() function.

```
int wctomb(char *str, wchar_t wchar)
```

Parameters

- **str** -- This is the pointer to an array large enough to hold a multibyte character,
- **wchar** -- This is the wide character of type wchar_t.

Return Value

- If str is not NULL, the wctomb() function returns the number of bytes that have been written to the byte array at str. If wchar cannot be represented as a multibyte sequence, -1 is returned.
- If str is NULL, the wctomb() function returns non-zero if the encoding has non-trivial shift state, or zero if the encoding is stateless.

Example

The following example shows the usage of wctomb() function.

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

int main()
{
   int i;
   wchar_t wc = L'a';
   char *pmbnull = NULL;
   char *pmb = (char *)malloc(sizeof( char ));

   printf("Converting wide character:\n");
   i = wctomb( pmb, wc );
   printf("Characters converted: %u\n", i);
   printf("Multibyte character: %.1s\n", pmb);

   printf("Trying to convert when target is NULL:\n");
   i = wctomb( pmbnull, wc );
   printf("Characters converted: %u\n", i);
   /* this will not print any value */
   printf("Multibyte character: %.1s\n", pmbnull);

   return(0);
```

```
}
```

Let us compile and run the above program that will produce the following result:

```
Converting wide character:
Characters converted: 1
Multibyte character: a
Trying to convert when target is NULL:
Characters converted: 0
Multibyte character:
```

14. C Library – <string.h>

Introduction

The **string.h** header defines one variable type, one macro, and various functions for manipulating arrays of characters.

Library Variables

Following is the variable type defined in the header string.h:

S.N.	Variable & Description
1	**size_t** This is the unsigned integral type and is the result of the **sizeof** keyword.

Library Macros

Following is the macro defined in the header string.h:

S.N.	Macro & Description
1	**NULL** This macro is the value of a null pointer constant.

Library Functions

Following are the functions defined in the header string.h:

S.N.	Function & Description
1	void *memchr(const void *str, int c, size_t n) Searches for the first occurrence of the character c (an unsigned char) in the first n bytes of the string pointed to, by the argument *str*.
2	int memcmp(const void *str1, const void *str2, size_t n) Compares the first n bytes of *str1* and *str2*.
3	void *memcpy(void *dest, const void *src, size_t n)

	Copies n characters from src to *dest*.
4	void *memmove(void *dest, const void *src, size_t n) Another function to copy n characters from *str2* to *str1*.
5	void *memset(void *str, int c, size_t n) Copies the character c (an unsigned char) to the first n characters of the string pointed to, by the argument *str*.
6	char *strcat(char *dest, const char *src) Appends the string pointed to, by *src* to the end of the string pointed to by *dest*.
7	char *strncat(char *dest, const char *src, size_t n) Appends the string pointed to, by *src* to the end of the string pointed to, by *dest*up to n characters long.
8	char *strchr(const char *str, int c) Searches for the first occurrence of the character c (an unsigned char) in the string pointed to, by the argument *str*.
9	int strcmp(const char *str1, const char *str2) Compares the string pointed to, by *str1* to the string pointed to by *str2*.
10	int strncmp(const char *str1, const char *str2, size_t n) Compares at most the first n bytes of *str1* and *str2*.
11	int strcoll(const char *str1, const char *str2) Compares string *str1* to *str2*. The result is dependent on the LC_COLLATE setting of the location.
12	char *strcpy(char *dest, const char *src) Copies the string pointed to, by *src* to *dest*.
13	char *strncpy(char *dest, const char *src, size_t n) Copies up to n characters from the string pointed to, by *src* to *dest*.
14	size_t strcspn(const char *str1, const char *str2) Calculates the length of the initial segment of str1 which consists entirely of

	characters not in str2.
15	char *strerror(int errnum) Searches an internal array for the error number errnum and returns a pointer to an error message string.
16	size_t strlen(const char *str) Computes the length of the string str up to but not including the terminating null character.
17	char *strpbrk(const char *str1, const char *str2) Finds the first character in the string *str1* that matches any character specified in *str2*.
18	char *strrchr(const char *str, int c) Searches for the last occurrence of the character c (an unsigned char) in the string pointed to by the argument *str*.
19	size_t strspn(const char *str1, const char *str2) Calculates the length of the initial segment of *str1* which consists entirely of characters in *str2*.
20	char *strstr(const char *haystack, const char *needle) Finds the first occurrence of the entire string *needle* (not including the terminating null character) which appears in the string *haystack*.
21	char *strtok(char *str, const char *delim) Breaks string *str* into a series of tokens separated by *delim*.
22	size_t strxfrm(char *dest, const char *src, size_t n) Transforms the first **n** characters of the string **src** into corrent locale and places them in the string **dest**.

void *memchr(const void *str, int c, size_t n)

Description

The C library function **void *memchr(const void *str, int c, size_t n)** searches for the first occurrence of the character **c** (an unsigned char) in the first **n** bytes of the string pointed to, by the argument **str**.

Declaration

Following is the declaration for memchr() function.

```
void *memchr(const void *str, int c, size_t n)
```

Parameters

- **str** -- This is the pointer to the block of memory where the search is performed.
- **c** -- This is the value to be passed as an int, but the function performs a byte per byte search using the unsigned char conversion of this value.
- **n** -- This is the number of bytes to be analyzed.

Return Value

This function returns a pointer to the matching byte or NULL if the character does not occur in the given memory area.

Example

The following example shows the usage of memchr() function.

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

int main ()
{
   const char str[] = "http://www.tutorialspoint.com";
   const char ch = '.';
   char *ret;

   ret = memchr(str, ch, strlen(str));

   printf("String after |%c| is - |%s|\n", ch, ret);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
String after |.| is - |.tutorialspoint.com|
```

int memcmp(const void *str1, const void *str2, size_t n)

Description

The C library function **int memcmp(const void *str1, const void *str2, size_t n))** compares the first **n** bytes of memory area **str1** and memory area **str2**.

Declaration

Following is the declaration for memcmp() function.

```
int memcmp(const void *str1, const void *str2, size_t n)
```

Parameters

- **str1** -- This is the pointer to a block of memory.
- **str2** -- This is the pointer to a block of memory.
- **n** -- This is the number of bytes to be compared.

Return Value

- if Return value is < 0 then it indicates str1 is less than str2.
- if Return value is > 0 then it indicates str2 is less than str1.
- if Return value is = 0 then it indicates str1 is equal to str2.

Example

The following example shows the usage of memcmp() function.

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

int main ()
{
   char str1[15];
   char str2[15];
   int ret;

   memcpy(str1, "abcdef", 6);
   memcpy(str2, "ABCDEF", 6);

   ret = memcmp(str1, str2, 5);

   if(ret > 0)
   {
      printf("str2 is less than str1");
   }
   else if(ret < 0)
   {
      printf("str1 is less than str2");
   }
```

```
   else
   {
      printf("str1 is equal to str2");
   }

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
str2 is less than str1
```

void *memcpy(void *dest, const void *src, size_t n)

Description

The C library function **void *memcpy(void *str1, const void *str2, size_t n)** copies **n** characters from memory area **str2** to memory area **str1**.

Declaration

Following is the declaration for memcpy() function.

```
void *memcpy(void *str1, const void *str2, size_t n)
```

Parameters

- **str1** -- This is pointer to the destination array where the content is to be copied, type-casted to a pointer of type void*.
- **str2** -- This is pointer to the source of data to be copied, type-casted to a pointer of type void*.
- **n** -- This is the number of bytes to be copied.

Return Value

This function returns a pointer to destination, which is str1.

Example

The following example shows the usage of memcpy() function.

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

int main ()
{
   const char src[50] = "http://www.tutorialspoint.com";
   char dest[50];
```

```
   printf("Before memcpy dest = %s\n", dest);
   memcpy(dest, src, strlen(src)+1);
   printf("After memcpy dest = %s\n", dest);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Before memcpy dest =
After memcpy dest = http://www.tutorialspoint.com
```

void *memmove(void *dest, const void *src, size_t n)

Description

The C library function **void *memmove(void *str1, const void *str2, size_t n)** copies **n** characters from **str2** to **str1**, but for overlapping memory blocks, memmove() is a safer approach than memcpy().

Declaration

Following is the declaration for memmove() function.

```
void *memmove(void *str1, const void *str2, size_t n)
```

Parameters

- **str1** -- This is a pointer to the destination array where the content is to be copied, type-casted to a pointer of type void*.
- **str2** -- This is a pointer to the source of data to be copied, type-casted to a pointer of type void*.
- **n** -- This is the number of bytes to be copied.

Return Value

This function returns a pointer to the destination, which is str1.

Example

The following example shows the usage of memmove() function.

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

int main ()
{
   const char dest[] = "oldstring";
   const char src[]  = "newstring";
```

```
   printf("Before memmove dest = %s, src = %s\n", dest, src);
   memmove(dest, src, 9);
   printf("After memmove dest = %s, src = %s\n", dest, src);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Before memmove dest = oldstring, src = newstring
After memmove dest = newstring, src = newstring
```

void *memset(void *str, int c, size_t n)

Description

The C library function **void *memset(void *str, int c, size_t n)** copies the character **c** (an unsigned char) to the first **n** characters of the string pointed to, by the argument **str**.

Declaration

Following is the declaration for memset() function.

```
void *memset(void *str, int c, size_t n)
```

Parameters

- **str** -- This is a pointer to the block of memory to fill.
- **c** -- This is the value to be set. The value is passed as an int, but the function fills the block of memory using the unsigned char conversion of this value.
- **n** -- This is the number of bytes to be set to the value.

Return Value

This function returns a pointer to the memory area str.

Example

The following example shows the usage of memset() function.

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

int main ()
{
   char str[50];
```

```
   strcpy(str,"This is string.h library function");
   puts(str);

   memset(str,'$',7);
   puts(str);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
This is string.h library function
$$$$$$$ string.h library function
```

char *strcat(char *dest, const char *src)

Description

The C library function **char *strcat(char *dest, const char *src)** appends the string pointed to by **src** to the end of the string pointed to by **dest**.

Declaration

Following is the declaration for strcat() function.

```
char *strcat(char *dest, const char *src)
```

Parameters

- **dest** -- This is pointer to the destination array, which should contain a C string, and should be large enough to contain the concatenated resulting string.
- **src** -- This is the string to be appended. This should not overlap the destination.

Return Value

This function returns a pointer to the resulting string dest.

Example

The following example shows the usage of strcat() function.

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

int main ()
{
   char src[50], dest[50];
```

```
   strcpy(src,  "This is source");
   strcpy(dest, "This is destination");

   strcat(dest, src);

   printf("Final destination string : |%s|", dest);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Final destination string : |This is destinationThis is source|
```

char *strncat(char *dest, const char *src, size_t n)

Description

The C library function **char *strncat(char *dest, const char *src, size_t n)** appends the string pointed to by **src** to the end of the string pointed to by **dest** up to **n** characters long.

Declaration

Following is the declaration for strncat() function.

```
char *strncat(char *dest, const char *src, size_t n)
```

Parameters

- **dest** -- This is pointer to the destination array, which should contain a C string, and should be large enough to contain the concatenated resulting string which includes the additional null-character.
- **src** -- This is the string to be appended.
- **n** -- This is the maximum number of characters to be appended.

Return Value

This function returns a pointer to the resulting string dest.

Example

The following example shows the usage of strncat() function.

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

int main ()
```

```
{
   char src[50], dest[50];

   strcpy(src,  "This is source");
   strcpy(dest, "This is destination");

   strncat(dest, src, 15);

   printf("Final destination string : |%s|", dest);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Final destination string : |This is destinationThis is source|
```

char *strchr(const char *str, int c)

Description

The C library function **char *strchr(const char *str, int c)** searches for the first occurrence of the character **c** (an unsigned char) in the string pointed to by the argument **str**.

Declaration

Following is the declaration for strchr() function.

```
char *strchr(const char *str, int c)
```

Parameters

- **str** -- This is the C string to be scanned.
- **c** -- This is the character to be searched in str.

Return Value

This returns a pointer to the first occurrence of the character c in the string str, or NULL if the character is not found.

Example

The following example shows the usage of strchr() function.

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

int main ()
{
   const char str[] = "http://www.tutorialspoint.com";
   const char ch = '.';
   char *ret;

   ret = strchr(str, ch);

   printf("String after |%c| is - |%s|\n", ch, ret);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
String after |.| is - |.tutorialspoint.com|
```

int strcmp(const char *str1, const char *str2)

Description

The C library function **int strcmp(const char *str1, const char *str2)** compares the string pointed to, by **str1** to the string pointed to by **str2**.

Declaration

Following is the declaration for strcmp() function.

```
int strcmp(const char *str1, const char *str2)
```

Parameters

- **str1** -- This is the first string to be compared.
- **str2** -- This is the second string to be compared.

Return Value

This function return values that are as follows:

- if Return value is < 0 then it indicates str1 is less than str2.
- if Return value is > 0 then it indicates str2 is less than str1.
- if Return value is = 0 then it indicates str1 is equal to str2.

Example

The following example shows the usage of strncmp() function.

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

int main ()
{
   char str1[15];
   char str2[15];
   int ret;


   strcpy(str1, "abcdef");
   strcpy(str2, "ABCDEF");

   ret = strcmp(str1, str2);

   if(ret < 0)
   {
      printf("str1 is less than str2");
   }
   else if(ret > 0)
   {
      printf("str2 is less than str1");
   }
   else
   {
      printf("str1 is equal to str2");
   }

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
str2 is less than str1
```

int strncmp(const char *str1, const char *str2, size_t n)

Description

The C library function **int strncmp(const char *str1, const char *str2, size_t n)** compares at most the first **n** bytes of **str1** and **str2**.

Declaration

Following is the declaration for strncmp() function.

```
int strncmp(const char *str1, const char *str2, size_t n)
```

Parameters

- **str1** -- This is the first string to be compared.
- **str2** -- This is the second string to be compared.
- **n** -- The maximum number of characters to be compared.

Return Value

This function return values that are as follows:

- if Return value is < 0 then it indicates str1 is less than str2.
- if Return value is > 0 then it indicates str2 is less than str1.
- if Return value is = 0 then it indicates str1 is equal to str2.

Example

The following example shows the usage of strncmp() function.

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

int main ()
{
   char str1[15];
   char str2[15];
   int ret;


   strcpy(str1, "abcdef");
   strcpy(str2, "ABCDEF");

   ret = strncmp(str1, str2, 4);

   if(ret < 0)
   {
      printf("str1 is less than str2");
   }
```

```
   else if(ret > 0)
   {
      printf("str2 is less than str1");
   }
   else
   {
      printf("str1 is equal to str2");
   }

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
str2 is less than str1
```

int strcoll(const char *str1, const char *str2)

Description

The C library function **int strcoll(const char *str1, const char *str2)** compares string **str1** to **str2**. The result is dependent on the LC_COLLATE setting of the location.

Declaration

Following is the declaration for strcoll() function.

```
int strcoll(const char *str1, const char *str2)
```

Parameters

- **str1** -- This is the first string to be compared.
- **str2** -- This is the second string to be compared.

Return Value

This function return values that are as follows:

- if Return value is < 0 then it indicates str1 is less than str2.
- if Return value is > 0 then it indicates str2 is less than str1.
- if Return value is = 0 then it indicates str1 is equal to str2.

Example

The following example shows the usage of strcoll() function.

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

int main ()
{
   char str1[15];
   char str2[15];
   int ret;


   strcpy(str1, "abc");
   strcpy(str2, "ABC");

   ret = strcoll(str1, str2);

   if(ret > 0)
   {
      printf("str1 is less than str2");
   }
   else if(ret < 0)
   {
      printf("str2 is less than str1");
   }
   else
   {
      printf("str1 is equal to str2");
   }

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
str1 is less than str2
```

char *strcpy(char *dest, const char *src)

Description

The C library function **char *strcpy(char *dest, const char *src)** copies the string pointed to, by **src** to **dest**.

Declaration

Following is the declaration for strcpy() function.

```
char *strcpy(char *dest, const char *src)
```

Parameters

- **dest** -- This is the pointer to the destination array where the content is to be copied.
- **src** -- This is the string to be copied.

Return Value

This returns a pointer to the destination string dest.

Example

The following example shows the usage of strcpy() function.

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

int main()
{
   char src[40];
   char dest[100];

   memset(dest, '\0', sizeof(dest));
   strcpy(src, "This is tutorialspoint.com");
   strcpy(dest, src);

   printf("Final copied string : %s\n", dest);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Final copied string : This is tutorialspoint.com
```

char *strncpy(char *dest, const char *src, size_t n)

Description

The C library function **char *strncpy(char *dest, const char *src, size_t n)** copies up to **n** characters from the string pointed to, by **src** to **dest**. In a case where the length of src is less than that of n, the remainder of dest will be padded with null bytes.

Declaration

Following is the declaration for strncpy() function.

```
char *strncpy(char *dest, const char *src, size_t n)
```

Parameters

- **dest** -- This is the pointer to the destination array where the content is to be copied.
- **src** -- This is the string to be copied.
- **n** -- The number of characters to be copied from source.

Return Value

This function returns the final copy of the copied string.

Example

The following example shows the usage of strncpy() function. Here we have used function memset() to clear the memory location.

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

int main()
{
   char src[40];
   char dest[12];

   memset(dest, '\0', sizeof(dest));
   strcpy(src, "This is tutorialspoint.com");
   strncpy(dest, src, 10);

   printf("Final copied string : %s\n", dest);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Final copied string : This is tu
```

size_t strcspn(const char *str1, const char *str2)

Description

The C library function **size_t strcspn(const char *str1, const char *str2)** calculates the length of the initial segment of **str1**, which consists entirely of characters not in **str2**.

Declaration

Following is the declaration for strcspn() function.

```
size_t strcspn(const char *str1, const char *str2)
```

Parameters

- **str1** -- This is the main C string to be scanned.
- **str2** -- This is the string containing a list of characters to match in str1.

Return Value

This function returns the number of characters in the initial segment of string str1, which are not in the string str2.

Example

The following example shows the usage of strcspn() function.

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


int main ()
{
   int len;
   const char str1[] = "ABCDEF4960910";
   const char str2[] = "013";

   len = strcspn(str1, str2);

   printf("First matched character is at %d\n", len + 1);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
First matched character is at 10
```

char *strerror(int errnum)

Description

The C library function **char *strerror(int errnum)** searches an internal array for the error number **errnum** and returns a pointer to an error message string. The error strings produced by **strerror** depend on the developing platform and compiler.

Declaration

Following is the declaration for strerror() function.

```
char *strerror(int errnum)
```

Parameters

- **errnum** -- This is the error number, usually **errno**.

Return Value

This function returns a pointer to the error string describing error errnum.

Example

The following example shows the usage of strerror() function.

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

int main ()
{
   FILE *fp;

   fp = fopen("file.txt","r");
   if( fp == NULL )
   {
      printf("Error: %s\n", strerror(errno));
   }

  return(0);
}
```

Let us compile and run the above program that will produce the following result because we are trying to open a file which does not exist:

```
Error: No such file or directory
```

size_t strlen(const char *str)

Description

The C library function **size_t strlen(const char *str)** computes the length of the string **str** up to, but not including the terminating null character.

Declaration

Following is the declaration for strlen() function.

```
size_t strlen(const char *str)
```

Parameters

- **str** -- This is the string whose length is to be found.

Return Value

This function returns the length of string.

Example

The following example shows the usage of strlen() function.

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

int main ()
{
   char str[50];
   int len;

   strcpy(str, "This is tutorialspoint.com");

   len = strlen(str);
   printf("Length of |%s| is |%d|\n", str, len);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Length of |This is tutorialspoint.com| is |26|
```

char *strpbrk(const char *str1, const char *str2)

Description

The C library function **char *strpbrk(const char *str1, const char *str2)** finds the first character in the string **str1** that matches any character specified in **str2**. This does not include the terminating null-characters.

Declaration

Following is the declaration for strpbrk() function.

```
char *strpbrk(const char *str1, const char *str2)
```

Parameters

- **str1** -- This is the C string to be scanned.
- **str2** -- This is the C string containing the characters to match.

Return Value

This function returns a pointer to the character in str1 that matches one of the characters in str2, or NULL if no such character is found.

Example

The following example shows the usage of strpbrk() function.

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

int main ()
{
   const char str1[] = "abcde2fghi3jk4l";
   const char str2[] = "34";
   char *ret;

   ret = strpbrk(str1, str2);
   if(ret)
   {
      printf("First matching character: %c\n", *ret);
   }
   else
   {
      printf("Character not found");
   }

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
First matching character: 3
```

char *strrchr(const char *str, int c)

Description

The C library function **char *strrchr(const char *str, int c)** searches for the last occurrence of the character **c** (an unsigned char) in the string pointed to, by the argument **str**.

Declaration

Following is the declaration for strrchr() function.

```
char *strrchr(const char *str, int c)
```

Parameters

- **str** -- This is the C string.
- **c** -- This is the character to be located. It is passed as its int promotion, but it is internally converted back to char.

Return Value

This function returns a pointer to the last occurrence of character in str. If the value is not found, the function returns a null pointer.

Example

The following example shows the usage of strrchr() function.

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

int main ()
{
   int len;
   const char str[] = "http://www.tutorialspoint.com";
   const char ch = '.';
   char *ret;

   ret = strrchr(str, ch);

   printf("String after |%c| is - |%s|\n", ch, ret);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
String after |.| is - |.com|
```

size_t strspn(const char *str1, const char *str2)

Description

The C library function **size_t strspn(const char *str1, const char *str2)** calculates the length of the initial segment of **str1** which consists entirely of characters in **str2**.

Declaration

Following is the declaration for strspn() function.

```
size_t strspn(const char *str1, const char *str2)
```

Parameters

- **str1** -- This is the main C string to be scanned.
- **str2** -- This is the string containing the list of characters to match in str1.

Return Value

This function returns the number of characters in the initial segment of str1 which consist only of characters from str2.

Example

The following example shows the usage of strspn() function.

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

int main ()
{
   int len;
   const char str1[] = "ABCDEFG019874";
   const char str2[] = "ABCD";

   len = strspn(str1, str2);

   printf("Length of initial segment matching %d\n", len );

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Length of initial segment matching 4
```

char *strstr(const char *haystack, const char *needle)

Description

The C library function **char *strstr(const char *haystack, const char *needle)** finds the first occurrence of the substring **needle** in the string **haystack**. The terminating '\0' characters are not compared.

Declaration

Following is the declaration for strstr() function.

```
char *strstr(const char *haystack, const char *needle)
```

Parameters

- **haystack** -- This is the main C string to be scanned.
- **needle** -- This is the small string to be searched within haystack string.

Return Value

This function returns a pointer to the first occurrence in haystack of any of the entire sequence of characters specified in needle, or a null pointer if the sequence is not present in haystack.

Example

The following example shows the usage of strstr() function.

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


int main()
{
   const char haystack[20] = "TutorialsPoint";
   const char needle[10] = "Point";
   char *ret;

   ret = strstr(haystack, needle);

   printf("The substring is: %s\n", ret);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
The substring is: Point
```

char *strtok(char *str, const char *delim)

Description

The C library function **char *strtok(char *str, const char *delim)** breaks string **str** into a series of tokens using the delimiter **delim**.

Declaration

Following is the declaration for strtok() function.

```
char *strtok(char *str, const char *delim)
```

Parameters

- **str** -- The contents of this string are modified and broken into smaller strings (tokens).
- **delim** -- This is the C string containing the delimiters. These may vary from one call to another.

Return Value

This function returns a pointer to the last token found in the string. A null pointer is returned if there are no tokens left to retrieve.

Example

The following example shows the usage of strtok() function.

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

int main()
{
   const char str[80] = "This is - www.tutorialspoint.com - website";
   const char s[2] = "-";
   char *token;

   /* get the first token */
   token = strtok(str, s);

   /* walk through other tokens */
   while( token != NULL )
   {
      printf( " %s\n", token );
```

```
      token = strtok(NULL, s);
   }

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
This is
www.tutorialspoint.com
website
```

size_t strxfrm(char *dest, const char *src, size_t n)

Description

The C library function **size_t strxfrm(char *dest, const char *src, size_t n)** transforms the first **n** characters of the string **src** into current locale and place them in the string **dest**.

Declaration

Following is the declaration for strxfrm() function.

```
size_t strxfrm(char *dest, const char *src, size_t n)
```

Parameters

- **dest** -- This is the pointer to the destination array where the content is to be copied. It can be a null pointer if the argument for n is zero.
- **src** -- This is the C string to be transformed into current locale.
- **n** -- The maximum number of characters to be copied to str1.

Return Value

This function returns the length of the transformed string, not including the terminating null-character.

Example

The following example shows the usage of strxfrm() function.

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

```
int main()
{
   char dest[20];
   char src[20];
   int len;

   strcpy(src, "Tutorials Point");
   len = strxfrm(dest, src, 20);

   printf("Length of string |%s| is: |%d|", dest, len);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Length of string |Tutorials Point| is: |15|
```

15. C Library – <time.h>

Introduction

The **time.h** header defines four variable types, two macro and various functions for manipulating date and time.

Library Variables

Following are the variable types defined in the header time.h:

S.N.	Variable & Description
1	**size_t** This is the unsigned integral type and is the result of the **sizeof** keyword.
2	**clock_t** This is a type suitable for storing the processor time.
3	**time_t is** This is a type suitable for storing the calendar time.
4	**struct tm** This is a structure used to hold the time and date.

The tm structure has the following definition:

```
struct tm {
   int tm_sec;         /* seconds,  range 0 to 59          */
   int tm_min;         /* minutes, range 0 to 59           */
   int tm_hour;        /* hours, range 0 to 23             */
   int tm_mday;        /* day of the month, range 1 to 31  */
   int tm_mon;         /* month, range 0 to 11             */
   int tm_year;        /* The number of years since 1900   */
   int tm_wday;        /* day of the week, range 0 to 6    */
   int tm_yday;        /* day in the year, range 0 to 365  */
   int tm_isdst;       /* daylight saving time             */
};
```

Library Macros

Following are the macros defined in the header time.h:

S.N.	Macro & Description
1	**NULL** This macro is the value of a null pointer constant.
2	**CLOCKS_PER_SEC** This macro represents the number of processor clocks per second.

Library Functions

Following are the functions defined in the header time.h:

S.N.	Function & Description
1	char *asctime(const struct tm *timeptr) Returns a pointer to a string which represents the day and time of the structure timeptr.
2	clock_t clock(void) Returns the processor clock time used since the beginning of an implementation defined era (normally the beginning of the program).
3	char *ctime(const time_t *timer) Returns a string representing the localtime based on the argument timer.
4	double difftime(time_t time1, time_t time2) Returns the difference of seconds between time1 and time2 (time1-time2).
5	struct tm *gmtime(const time_t *timer) The value of timer is broken up into the structure tm and expressed in Coordinated Universal Time (UTC), also known as Greenwich Mean Time (GMT).
6	struct tm *localtime(const time_t *timer) The value of timer is broken up into the structure tm and expressed in the local time zone.
7	time_t mktime(struct tm *timeptr) Converts the structure pointed to by timeptr into a time_t value according to

	the local time zone.
8	size_t strftime(char *str, size_t maxsize, const char *format, const struct tm *timeptr) Formats the time represented in the structure timeptr according to the formatting rules defined in format and stored into str.
9	time_t time(time_t *timer) Calculates the current calender time and encodes it into time_t format.

char *asctime(const struct tm *timeptr)

Description

The C library function **char *asctime(const struct tm *timeptr)** returns a pointer to a string which represents the day and time of the structure **struct timeptr**.

Declaration

Following is the declaration for asctime() function.

```
char *asctime(const struct tm *timeptr)
```

Parameters

The **timeptr** is a pointer to tm structure that contains a calendar time broken down into its components as shown below:

```
struct tm {
   int tm_sec;         /* seconds,  range 0 to 59          */
   int tm_min;         /* minutes, range 0 to 59           */
   int tm_hour;        /* hours, range 0 to 23             */
   int tm_mday;        /* day of the month, range 1 to 31  */
   int tm_mon;         /* month, range 0 to 11             */
   int tm_year;        /* The number of years since 1900   */
   int tm_wday;        /* day of the week, range 0 to 6    */
   int tm_yday;        /* day in the year, range 0 to 365  */
   int tm_isdst;       /* daylight saving time             */
};
```

Return Value

This function returns a C string containing the date and time information in a human-readable format **Www Mmm dd hh:mm:ss yyyy**, where *Www* is the weekday, *Mmm* the month in letters, *dd* the day of the month, *hh:mm:ss* the time, and *yyyy* the year.

Example

The following example shows the usage of asctime() function.

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

int main()
{
   struct tm t;

   t.tm_sec    = 10;
   t.tm_min    = 10;
   t.tm_hour   = 6;
   t.tm_mday   = 25;
   t.tm_mon    = 2;
   t.tm_year   = 89;
   t.tm_wday   = 6;

   puts(asctime(&t));

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Sat Mar 25 06:10:10 1989
```

clock_t clock(void)

Description

The C library function **clock_t clock(void)** returns the number of clock ticks elapsed since the program was launched. To get the number of seconds used by the CPU, you will need to divide by CLOCKS_PER_SEC.

On a 32 bit system where CLOCKS_PER_SEC equals 1000000 this function will return the same value approximately every 72 minutes.

Declaration

Following is the declaration for clock() function.

```
clock_t clock(void)
```

Parameters

- NA

Return Value

This function returns the number of clock ticks elapsed since the start of the program. On failure, the function returns a value of -1.

Example

The following example shows the usage of clock() function.

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

int main()
{
   clock_t start_t, end_t, total_t;
   int i;

   start_t = clock();
   printf("Starting of the program, start_t = %ld\n", start_t);

   printf("Going to scan a big loop, start_t = %ld\n", start_t);
   for(i=0; i< 10000000; i++)
   {
   }
   end_t = clock();
   printf("End of the big loop, end_t = %ld\n", end_t);

   total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
   printf("Total time taken by CPU: %f\n", total_t  );
   printf("Exiting of the program...\n");

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Starting of the program, start_t = 0
Going to scan a big loop, start_t = 0
End of the big loop, end_t = 20000
Total time taken by CPU: 0.000000
```

```
Exiting of the program...
```

char *ctime(const time_t *timer)

Description

The C library function **char *ctime(const time_t *timer)** returns a string representing the localtime based on the argument **timer**.

The returned string has the following format: **Www Mmm dd hh:mm:ss yyyy**,where *Www* is the weekday, *Mmm* the month in letters, *dd* the day of the month, *hh:mm:ss* the time, and *yyyy* the year.

Declaration

Following is the declaration for ctime() function.

```
char *ctime(const time_t *timer)
```

Parameters

- **timer** -- This is the pointer to a time_t object that contains a calendar time.

Return Value

This function returns a C string containing the date and time information in a human-readable format.

Example

The following example shows the usage of ctime() function.

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

int main ()
{
   time_t curtime;

   time(&curtime);

   printf("Current time = %s", ctime(&curtime));

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Current time = Mon Aug 13 08:23:14 2012
```

double difftime(time_t time1, time_t time2)

Description

The C library function **double difftime(time_t time1, time_t time2)** returns the difference of seconds between **time1** and **time2** i.e. **(time1 - time2)**. The two times are specified in calendar time, which represents the time elapsed since the Epoch (00:00:00 on January 1, 1970, Coordinated Universal Time (UTC)).

Declaration

Following is the declaration for difftime() function.

```
double difftime(time_t time1, time_t time2)
```

Parameters

- **time1** -- This is the time_t object for end time.
- **time2** -- This is the time_t object for start time.

Return Value

This function returns the difference of two times (time2 - time1) as a double value.

Example

The following example shows the usage of difftime() function.

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

int main ()
{
   time_t start_t, end_t;
   double diff_t;

   printf("Starting of the program...\n");
   time(&start_t);

   printf("Sleeping for 5 seconds...\n");
   sleep(5);

   time(&end_t);
   diff_t = difftime(end_t, start_t);

   printf("Execution time = %f\n", diff_t);
   printf("Exiting of the program...\n");
```

```
   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Starting of the program...
Sleeping for 5 seconds...
Execution time = 5.000000
Exiting of the program...
```

struct tm *gmtime(const time_t *timer)

Description

The C library function **struct tm *gmtime(const time_t *timer)** uses the value pointed by timer to fill a **tm** structure with the values that represent the corresponding time, expressed in Coordinated Universal Time (UTC) or GMT timezone.

Declaration

Following is the declaration for gmtime() function.

```
struct tm *gmtime(const time_t *timer)
```

Parameters

- **timeptr** -- This is the pointer to a time_t value representing a calendar time.

Return Value

This function returns pointer to a tm structure with the time information filled in. Below is the detail of timeptr structure:

```
struct tm {
   int tm_sec;         /* seconds,  range 0 to 59          */
   int tm_min;         /* minutes, range 0 to 59           */
   int tm_hour;        /* hours, range 0 to 23             */
   int tm_mday;        /* day of the month, range 1 to 31  */
   int tm_mon;         /* month, range 0 to 11             */
   int tm_year;        /* The number of years since 1900   */
   int tm_wday;        /* day of the week, range 0 to 6    */
   int tm_yday;        /* day in the year, range 0 to 365  */
   int tm_isdst;       /* daylight saving time             */
};
```

Example

The following example shows the usage of gmtime() function.

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

#define BST (+1)
#define CCT (+8)

int main ()
{

   time_t rawtime;
   struct tm *info;

   time(&rawtime);
   /* Get GMT time */
   info = gmtime(&rawtime );

   printf("Current world clock:\n");
   printf("London : %2d:%02d\n", (info->tm_hour+BST)%24, info->tm_min);
   printf("China  : %2d:%02d\n", (info->tm_hour+CCT)%24, info->tm_min);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Current world clock:
London : 14:10
China : 21:10
```

struct tm *localtime(const time_t *timer)

Description

The C library function **struct tm *localtime(const time_t *timer)** uses the time pointed by **timer** to fill a **tm** structure with the values that represent the corresponding local time. The value of **timer** is broken up into the structure **tm** and expressed in the local time zone.

Declaration

Following is the declaration for localtime() function.

```
struct tm *localtime(const time_t *timer)
```

Parameters

- **timer** -- This is the pointer to a time_t value representing a calendar time.

Return Value

This function returns a pointer to a **tm** structure with the time information filled in. Following is the tm structure information:

```
struct tm {
   int tm_sec;         /* seconds,  range 0 to 59          */
   int tm_min;         /* minutes, range 0 to 59           */
   int tm_hour;        /* hours, range 0 to 23             */
   int tm_mday;        /* day of the month, range 1 to 31  */
   int tm_mon;         /* month, range 0 to 11             */
   int tm_year;        /* The number of years since 1900   */
   int tm_wday;        /* day of the week, range 0 to 6    */
   int tm_yday;        /* day in the year, range 0 to 365  */
   int tm_isdst;       /* daylight saving time             */
};
```

Example

The following example shows the usage of localtime() function.

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

int main ()
{
   time_t rawtime;
   struct tm *info;
   char buffer[80];

   time( &rawtime );

   info = localtime( &rawtime );
   printf("Current local time and date: %s", asctime(info));

   return(0);
```

```
}
```

Let us compile and run the above program that will produce the following result:

```
Current local time and date: Thu Aug 23 09:12:05 2012
```

time_t mktime(struct tm *timeptr)

Description

The C library function **time_t mktime(struct tm *timeptr)** converts the structure pointed to by **timeptr** into a time_t value according to the local time zone.

Declaration

Following is the declaration for mktime() function.

```
time_t mktime(struct tm *timeptr)
```

Parameters

- **timeptr** -- This is the pointer to a time_t value representing a calendar time, broken down into its components. Below is the detail of timeptr structure

```
struct tm {
   int tm_sec;         /* seconds,  range 0 to 59          */
   int tm_min;         /* minutes, range 0 to 59           */
   int tm_hour;        /* hours, range 0 to 23             */
   int tm_mday;        /* day of the month, range 1 to 31  */
   int tm_mon;         /* month, range 0 to 11             */
   int tm_year;        /* The number of years since 1900   */
   int tm_wday;        /* day of the week, range 0 to 6    */
   int tm_yday;        /* day in the year, range 0 to 365  */
   int tm_isdst;       /* daylight saving time             */
};
```

Return Value

This function returns a time_t value corresponding to the calendar time passed as argument. On error, a -1 value is returned.

Example

The following example shows the usage of mktime() function.

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

int main ()
{
   int ret;
   struct tm info;
   char buffer[80];

   info.tm_year = 2001 - 1900;
   info.tm_mon = 7 - 1;
   info.tm_mday = 4;
   info.tm_hour = 0;
   info.tm_min = 0;
   info.tm_sec = 1;
   info.tm_isdst = -1;

   ret = mktime(&info);
   if( ret == -1 )
   {
       printf("Error: unable to make time using mktime\n");
   }
   else
   {
      strftime(buffer, sizeof(buffer), "%c", &info );
      printf(buffer);
   }

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Wed Jul 4 00:00:01 2001
```

size_t strftime(char *str, size_t maxsize, const char *format, const struct tm *timeptr)

Description

The C library function **size_t strftime(char *str, size_t maxsize, const char *format, const struct tm *timeptr)** formats the time represented in the structure **timeptr** according to the formatting rules defined in **format** and stored into **str**.

Declaration

Following is the declaration for strftime() function.

```
size_t strftime(char *str, size_t maxsize, const char *format, const struct tm
*timeptr)
```

Parameters

- **str** -- This is the pointer to the destination array where the resulting C string is copied.
- **maxsize** -- This is the maximum number of characters to be copied to str.
- **format** -- This is the C string containing any combination of regular characters and special format specifiers. These format specifiers are replaced by the function to the corresponding values to represent the time specified in tm. The format specifiers are:

Specifier	Replaced By	Example
%a	Abbreviated weekday name	Sun
%A	Full weekday name	Sunday
%b	Abbreviated month name	Mar
%B	Full month name	March
%c	Date and time representation	Sun Aug 19 02:56:02 2012
%d	Day of the month (01-31)	19
%H	Hour in 24h format (00-23)	14
%I	Hour in 12h format (01-12)	05
%j	Day of the year (001-366)	231

%m	Month as a decimal number (01-12)	08
%M	Minute (00-59)	55
%p	AM or PM designation	PM
%S	Second (00-61)	02
%U	Week number with the first Sunday as the first day of week one (00-53)	33
%w	Weekday as a decimal number with Sunday as 0 (0-6)	4
%W	Week number with the first Monday as the first day of week one (00-53)	34
%x	Date representation	08/19/12
%X	Time representation	02:50:06
%y	Year, last two digits (00-99)	01
%Y	Year	2012
%Z	Timezone name or abbreviation	CDT
%%	A % sign	%

- **timeptr** -- This is the pointer to a tm structure that contains a calendar time broken down into its components as shown below:

```
struct tm {
   int tm_sec;         /* seconds,  range 0 to 59          */
   int tm_min;         /* minutes, range 0 to 59           */
   int tm_hour;        /* hours, range 0 to 23             */
   int tm_mday;        /* day of the month, range 1 to 31  */
   int tm_mon;         /* month, range 0 to 11             */
   int tm_year;        /* The number of years since 1900   */
   int tm_wday;        /* day of the week, range 0 to 6    */
   int tm_yday;        /* day in the year, range 0 to 365  */
```

```
   int tm_isdst;       /* daylight saving time            */
};
```

Return Value

If the resulting C string fits in less than size characters (which includes the terminating null-character), the total number of characters copied to str (not including the terminating null-character) is returned otherwise, it returns zero.

Example

The following example shows the usage of strftime() function.

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

int main ()
{
   time_t rawtime;
   struct tm *info;
   char buffer[80];

   time( &rawtime );

   info = localtime( &rawtime );

   strftime(buffer,80,"%x - %I:%M%p", info);
   printf("Formatted date & time : |%s|\n", buffer );

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Formatted date & time : |08/23/12 - 12:40AM|
```

time_t time(time_t *timer)

Description

The C library function **time_t time(time_t *seconds)** returns the time since the Epoch (00:00:00 UTC, January 1, 1970), measured in seconds. If **seconds** is not NULL, the return value is also stored in variable **seconds**.

Declaration

Following is the declaration for time() function.

```
time_t time(time_t *t)
```

Parameters

- **seconds** -- This is the pointer to an object of type time_t, where the seconds value will be stored.

Return Value

The current calendar time as a time_t object.

Example

The following example shows the usage of time() function.

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

int main ()
{
  time_t seconds;

  seconds = time(NULL);
  printf("Hours since January 1, 1970 = %ld\n", seconds/3600);

  return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Hours since January 1, 1970 = 393923
```