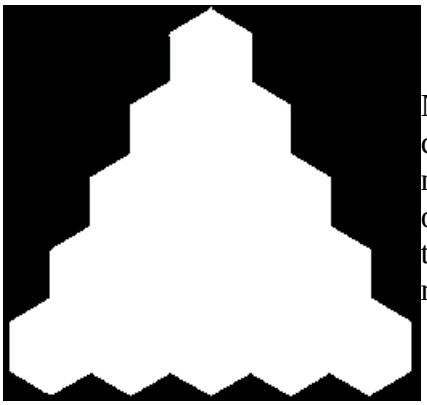


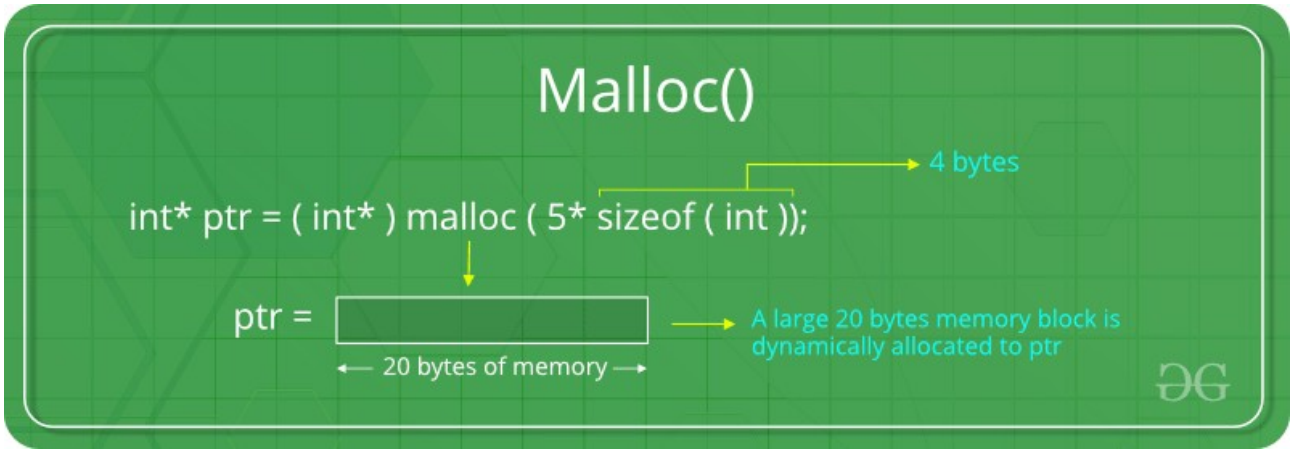
Note that .exe files created after compilation are binary files and they need not to be keep and let them occupy the storage. hence delete them because it will be automatically recreated after compiling again

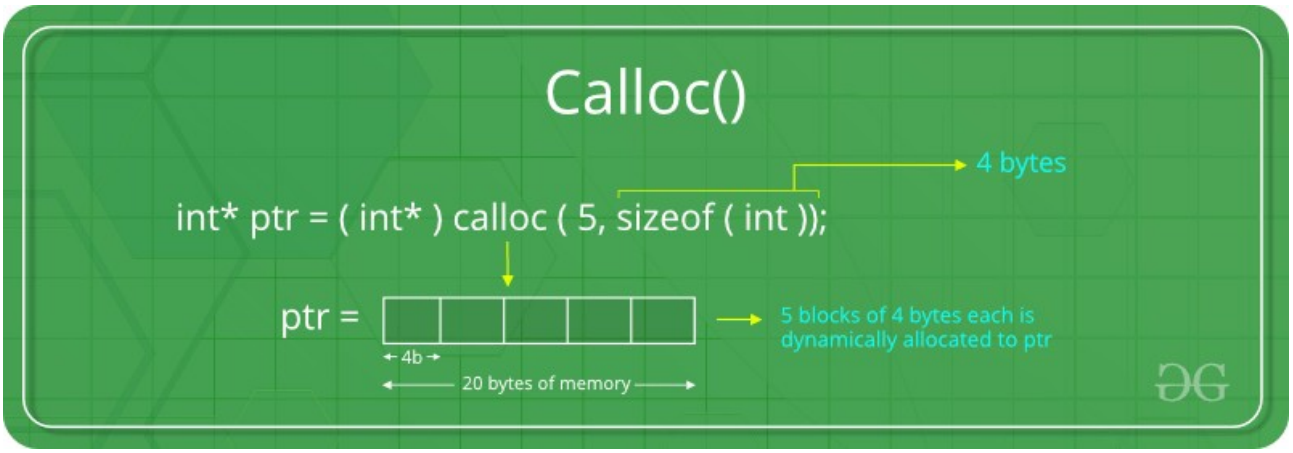
https://cplusplus.com/reference/cstdlib/bsearch/

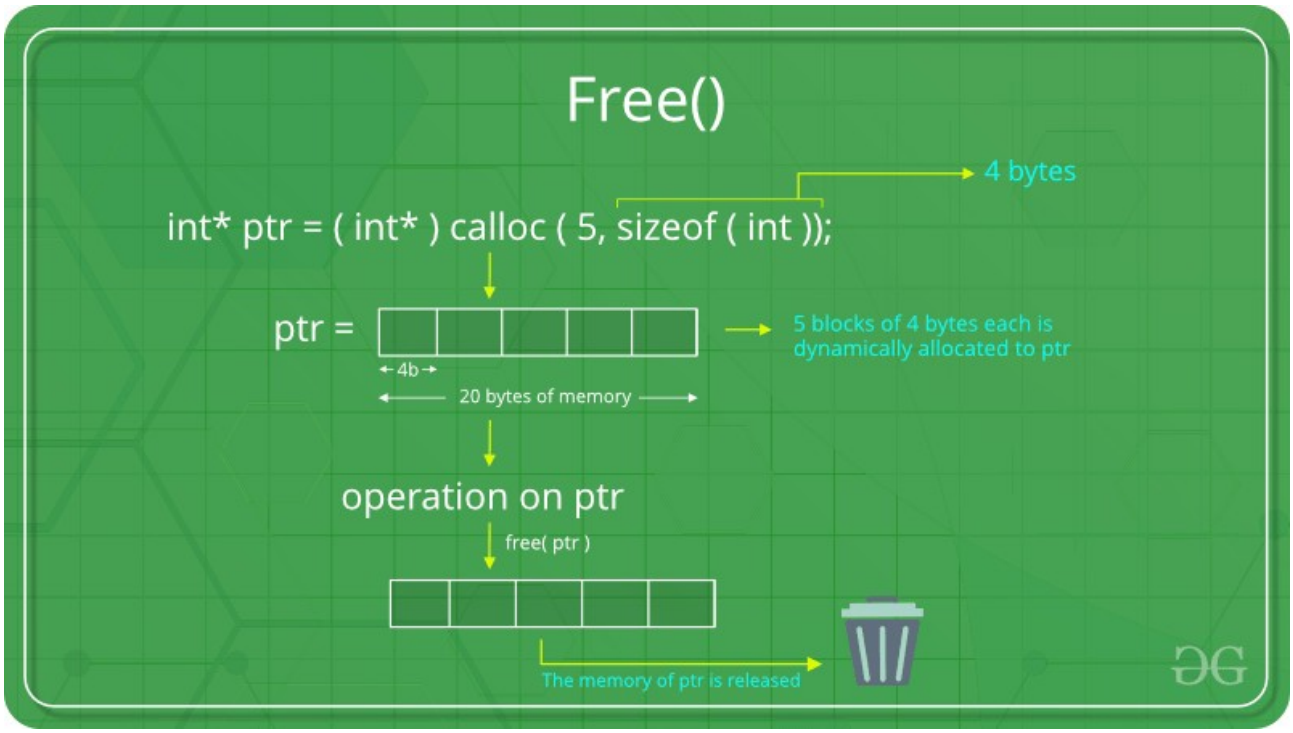
All functions in all the libraries

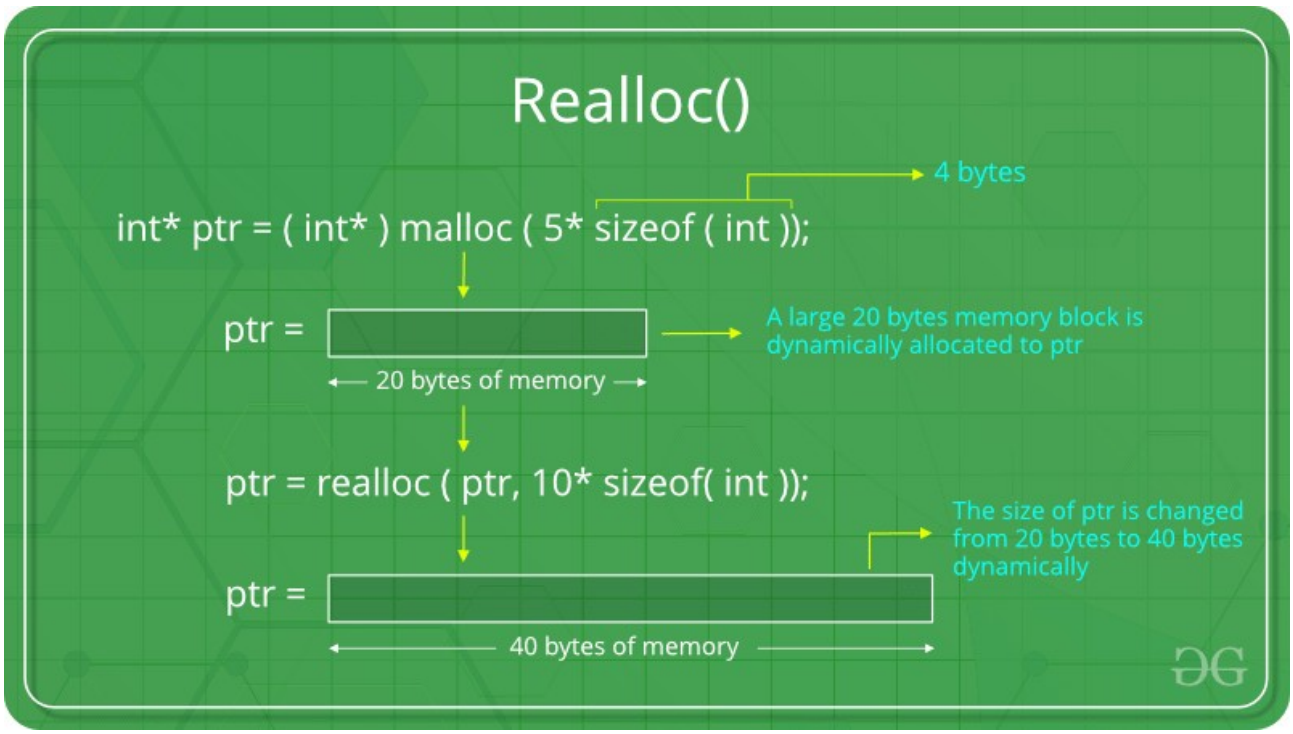
Pascal's Triangle

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5   10  10  5   1
    1   6   15  20  15  6   1
  1   7   21  35  35  21  7   1
```




Calloc()
4 bytes
int* ptr = (int*) calloc (5, sizeof (int));
ptr =
5 blocks of 4 bytes each is dynamically allocated to ptr
4b
20 bytes of memory


Free()
4 bytes
int* ptr = (int*) calloc (5, sizeof (int));
ptr =
5 blocks of 4 bytes each is dynamically allocated to ptr
4b
20 bytes of memory
operation on ptr
free(ptr)
The memory of ptr is released



It also contains CLOCKS_PER_SEC macro which holds the number of times does the system clock ticks per second.

time.h header file contains following data types.

clock_t
time_t
struct tm

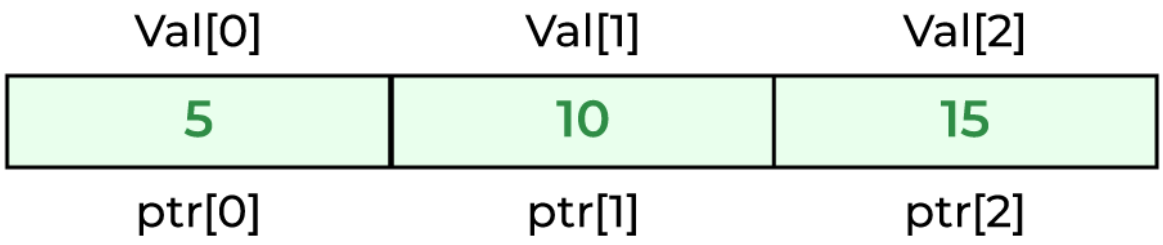
Pre-defined Functions in time.h

S.No	Function Name	Explanation
1.	**asctime()**	This function returns the date and time in the format day month date hours:minutes:seconds year. Eg: Sat Jul 27 11:26:03 2019. asctime() function returns a string by taking struct tm variable as a parameter.
2.	**clock()**	This function returns the processor time consumed by a program

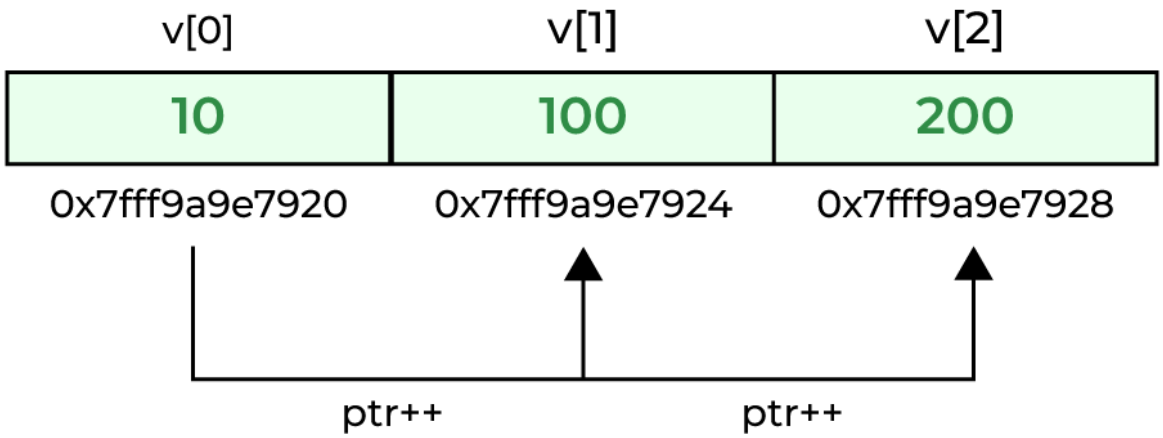
S.No	Function Name	Explanation
3.	**ctime()**	This function returns the date and time in the format day month date hours:minutes:seconds year Eg: Sat Jul 27 11:26:03 2019 Format is same as **asctime()** time is printed based on the pointer returned by Calendar Time
4.	**difftime()**	This function returns the difference between the times provided. `difftime(end, start))`
5.	**gmtime()**	This function prints the UTC (Coordinated Universal Time) Time and date. Format for both gmtime() and asctime() is same .All coded syntax are also same only instead of localtime , `gmtime` is used.
6.	mktime()	This function returns the calendar-time equivalent using struct tm.
7.	**time()**	This function returns the calendar-time equivalent using data-type time_t.
8.	**strftime()**	This function helps to format the string returned by other time functions using different format specifiers

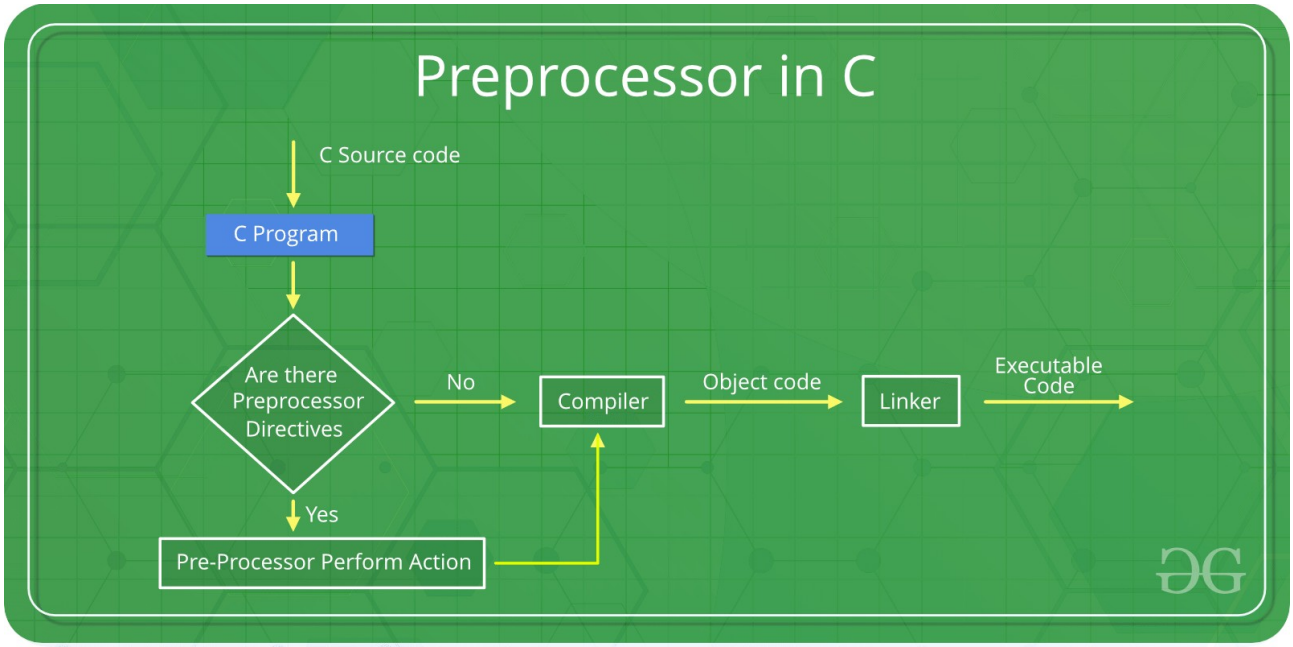
C Pointers and Arrays

Accessing Array Elements using Pointer with Array Subscript



if we have an array named val then val and &val[0] can be used interchangeably.





You can see the intermediate steps in the above diagram. The source code written by programmers is first stored in a file, let the name be “program.c“. This file is then processed by preprocessors and an expanded source code file is generated named “**program.i**”. This expanded file is compiled by the compiler and an object code file is generated named “**program.obj**”. Finally, the linker links this object code file to the object code of the library functions to generate the executable file “program.exe”.

List of preprocessor directives in C

The following table lists all the preprocessor directives in C: These can be used in main or a function.

Preprocessor Directives	Description
#define	Used to define a macro
#undef	Used to undefine a macro
#include	Used to include a file in the source code program

#ifdef	Used to include a section of code if a certain macro is defined by #define
	`#ifdef macro_name`
	`// Code to be executed if macro_name is defined`

#ifndef	Used to include a section of code if a certain macro is not defined by #define
	`#ifndef macro_name`
	`// Code to be executed if macro_name is not defined`

#if	Check for the specified condition
	`#if constant_expr`
	`// Code to be executed if constant_expression is true`

#else	Alternate code that executes when #if fails
	`#else`
	`// Code to be excuted if none of the above conditions are true`

#elif	Combines else and if for another condition check
	`#elif another_constant_expr`
	`// Code to be excuted if another_constant_expression is true`

Preprocessor Directives	Description
#endif	Used to mark the end of #if, #ifdef, and #ifndef `#endif`

There are 4 Main Types of Preprocessor Directives:

1.Macros

2.File Inclusion

3.Conditional Compilation

4.Other directives

In C, Macros are pieces of code in a program that is given some name. Whenever this name is encountered by the compiler, the compiler replaces the name with the actual piece of code. The ‘#define’ directive is used to define a macro.

```
#define token value
```

Predefined Macros in C

__**LINE**__ **Macro**: __LINE__ macro contains the current line number of the program in the compilation. It gives the line number where it is called. It is used in generating log statements, error messages, throwing exceptions and debugging codes. Whenever the compiler finds an error in compilation it first generates the line number at which error occurred using __LINE__ and prints error message along with line number so that user can easily fix that error easily.

__**FILE**__ **Macro**: __FILE__ macro holds the file name of the currently executing program in the computer. It is also used in debugging, generating error reports and log messages.

__**DATE**__ **Macro**: __DATE__ macro gives the date at which source code of this program is converted into object code.
Date is in the format *mmm dd yyyy*. mmm is the abbreviated month name.

__**TIME**__ **Macro**: __DATE__ macro gives the time at which program was compiled.

__**STDC__ Macro**: __STDC__ Macro is used to confirm the compiler standard. Generally it holds the value 1 which means that the compiler conforms to ISO Standard C.

__**STDC__HOSTED Macro**: This macro holds the value 1 if the compiler's target is a hosted environment. A hosted environment is a facility in which a third-party holds the compilation data and runs the programs on its own computers. Generally, the value is set to 1.

__**STDC_VERSION**__: This macro holds the C Standard's version number in the form *yyyymmL* where yyyy and mm are the year and month of the Standard version. This signifies which version of the C Standard the compiler conforms to.

other not work

File Inclusion

This type of preprocessor directive tells the compiler to include a file in the source code program. The **#include preprocessor directive** is used to include the header files in the C program.

There are two types of files that can be included by the user in the program:

1)Standard Header Files

The standard header files contain definitions of pre-defined functions like printf(), scanf(), etc

Different functions are declared in different header files.

Syntax

```
#include <file_name>
```

where *file_name* is the name of the header file to be included. The '**<' and '>' brackets** tell the compiler to look for the file in the s**tandard directory.**

2)User-defined Header Files

When a program becomes very large, it is a good practice to divide it into smaller files and include them whenever needed. These types of files are user-defined header files.

Syntax

```
#include "filename"
```

The **double quotes (" ")** tell the compiler to search for the header file in the **source file's directory.**

Conditional Compilation

a type of directive that helps to compile a specific portion of the program or to skip the compilation of some specific part of the program based on some conditions. There are the following preprocessor directives that are used to insert conditional code:

1. **#if Directive**
2. **#ifdef Directive**
3. **#ifndef Directive**
4. **#else Directive**
5. **#elif Directive**
6. **#endif Directive**

#endif directive is used to close off the #if, #ifdef, and #ifndef opening directives which means the preprocessing of these directives is completed.

difference between **Compile time**, **Load time** and **Execution time**

- **Compile time**. The compiler translates symbolic addresses to absolute addresses. If you know at compile time where the process will reside in memory, then absolute code can be generated (Static).
- **Load time**. The compiler translates symbolic addresses to relative (relocatable) addresses. The loader translates these to absolute addresses. If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code (Static).
- **Execution time**. If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. The absolute addresses are generated by hardware. Most general-purpose OSs use this method (Dynamic).

#pragma Directive

This directive is a special purpose directive and is used to turn on or off some features. These types of directives are compiler-specific, i.e., they vary from compiler to compiler.

Syntax

```
#pragma directive
```

1.**#pragma startup:** These directives help us to specify the functions that are needed to run before program startup (before the control passes to main()).

2.**#pragma exit**: These directives help us to specify the functions that are needed to run just before the program exit (just before the control returns from main()).

pragma program will not work with GCC compilers.

#pragma warn Directive

This directive is used to hide the warning message which is displayed during compilation. We can hide the warnings as shown below:

•**#pragma warn -rvl**: This directive hides those warnings which are raised when a function that is supposed to return a value does not return a value.

•**#pragma warn -par**: This directive hides those warnings which are raised when a function does not use the parameters passed to it.

•**#pragma warn -rch**: This directive hides those warnings which are raised when a code is unreachable. For example, any code written after the *return* statement in a function is unreachable.

File Handing in C

Syntax of fopen()

```
FILE* fopen(const char *file_name, const char *access_mode);
```

Opening Modes	Description
r	Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the first character in it. If the file cannot be opened fopen( ) returns NULL.
rb	Open for reading in binary mode. If the file does not exist, fopen( ) returns NULL.
w	Open for writing in text mode. If the file exists, its contents are overwritten. If the file doesn't exist, a new file is created. Returns NULL, if unable to open the file.
wb	Open for writing in binary mode. If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the last character in it. It opens only in the append mode. If the file doesn't exist, a new file is created. Returns NULL, if unable to open the file.
ab	Open for append in binary mode. Data is added to the end of the file. If the file does not exist, it will be created.
r+	Searches file. It is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the first character in it. Returns NULL, if unable to open the file. opens a file in both read and write mode
rb+	Open for both reading and writing in binary mode. If the file does not exist, fopen( ) returns NULL.
w+	Searches file. If the file exists, its contents are overwritten. If the file doesn't exist a new file is created. Returns NULL, if unable to open the file. `r+` - opens a file in both read and write mode
wb+	Open for both reading and writing in binary mode. If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the last character in it. It opens the file in

Opening Modes	Description
	both reading and append mode. If the file doesn't exist, a new file is created. Returns NULL, if unable to open the file.
ab+	Open for both reading and appending in binary mode. If the file does not exist, it will be created.

File operation	Declaration & Description
fopen() - To open a file	Declaration: FILE *fopen (const char *filename, const char *mode) fopen() function is used to open a file to perform operations such as reading, writing etc. In a C program, we declare a file pointer and use fopen() as below. fopen() function creates a new file if the mentioned file name does not exist. FILE *fp; fp=fopen ("filename", "'mode"); Where, fp – file pointer to the data type "FILE". filename – the actual file name with full path of the file. mode – refers to the operation that will be performed on the file. Example: r, w, a, r+, w+ and a+. Please refer below the description for these mode of operations.
fclose() - To close a file	Declaration: int fclose(FILE *fp); fclose() function closes the file that is being pointed by file pointer fp. In a C program, we close a file as below. fclose (fp);
fgets() - To read a file	Declaration: char *fgets(char *string, int n, FILE *fp) fgets function is used to read a file line by line. In a C program, we use fgets function as below. fgets (buffer, size, fp); where, buffer – buffer to put the data in. size – size of the buffer fp – file pointer
fprintf() - To write into a file	Declaration: int fprintf(FILE *fp, const char *format, ...);fprintf() function writes string into a file pointed by fp. In a C program, we write string into a file as below. fprintf (fp, "some data"); or fprintf (fp, "text %d", variable_name);

C provides a number of build-in function to perform basic file operations:

- `fopen()` - create a new file or open a existing file
- `fclose()` - close a file
- `getc()` - reads a character from a file

- `putc()` - writes a character to a file
- `fscanf()` - reads a set of data from a file
- `fprintf()` - writes a set of data to a file
- `getw()` - reads a integer from a file
- `putw()` - writes a integer to a file
- `fseek()` - set the position to desire point
- `ftell()` - gives current position in the file
- `rewind()` - set the position to the beginning point

Function	Description

fscanf()	Use formatted string and variable arguments list to take input from a file. It returns zero or EOF, if unsuccessful. Otherwise, it returns the number of items successfully assigned.
	`fscanf(fptr, "%s %s %s %d", str1, str2, str3, &year);`
	`char c = fgetc(fptr);`

fgets()	Input the whole line from the file.
fgetc()	Reads a single character from the file.
fgetw()	Reads a number from a file.

fread()	Reads the specified bytes of data from a binary file.
	`size_t fread(void * buffer, size_t size, size_t count, FILE * stream);`
	buffer: It refers to the pointer to the buffer memory block where the data read will be stored.
	•**size:** It refers to the size of each element in bytes.
	•**count:** It refers to the count of elements to be read.
	•**stream:** It refers to the pointer to the file stream.
	`struct threeNum num;//alias created`
	`FILE *fptr;`

Function	Description
	`fread(&num, sizeof(struct threeNum), 1, fptr);`

The **getc()** and some other file reading functions return EOF (End Of File) when they reach the end of the file while reading. EOF indicates the end of the file and its value is implementation-defined.
Note: *One thing to note here is that after reading a particular part of the file, the file pointer will be automatically moved to the end of the last read character.*

C getchar is a standard library function that takes a single input character from standard input. The major difference between getchar and getc is that getc can take input from any no of input streams but getchar can take input from a single standard input stream.
•It is defined inside the <stdio.h> header file.

- •Just like getchar, there is also a function called putchar that prints only one character to the standard output stream.

Syntax of getchar() in C

```
int getchar(void);
```

getchar() function does not take any parameters.

- •The input from the standard input is read as an unsigned char and then it is typecast and returned as an integer value(int).
- •EOF is returned in two cases:
  - ❍When the file end is reached
  - ❍When there is an error during the execution

function

<cstdlib>

qsort

```
void qsort (void* base, size_t num, size_t size,
           int (*compar)(const void*,const void*));
```

Sort elements of array

Sorts the *num* elements of the array pointed to by *base*, each element *size* bytes long, using the *compar* function to determine the order.

The sorting algorithm used by this function compares pairs of elements by calling the specified *compar* function with pointers to them as argument.

The function does not return any value, but modifies the content of the array pointed to by *base* reordering its elements as defined by *compar*.

The order of equivalent elements is undefined.

Parameters

`base`
Pointer to the first object of the array to be sorted, converted to a `void*`.

`num`
Number of elements in the array pointed to by *base*.
`size_t` is an unsigned integral type.

`size`
Size in bytes of each element in the array.
`size_t` is an unsigned integral type.

`compar`
Pointer to a function that compares two elements.
This function is called repeatedly by `qsort` to compare two elements. It shall follow the following prototype:

```
int compar (const void* p1, const void* p2);
```

Taking two pointers as arguments (both converted to `const void*`). The function defines the order of the elements by returning (in a stable and transitive manner):

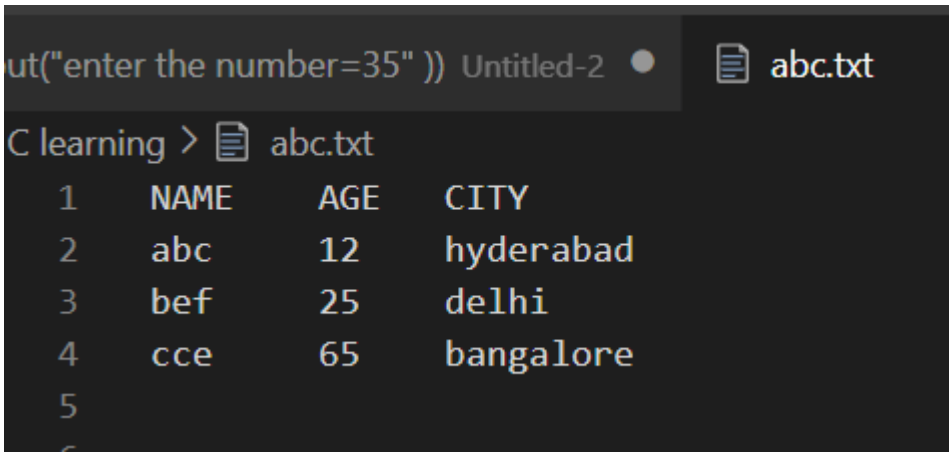
return value	meaning
<0	The element pointed to by *p1* goes before the element pointed to by *p2*
0	The element pointed to by *p1* is equivalent to the element pointed to by *p2*
>0	The element pointed to by *p1* goes after the element pointed to by *p2*

For types that can be compared using regular relational operators, a general *compar* function may look like:

```
int compareMyType (const void * a, const void * b)
{
```

```
  if ( *(MyType*)a <  *(MyType*)b ) return -1;
  if ( *(MyType*)a == *(MyType*)b ) return 0;
  if ( *(MyType*)a >  *(MyType*)b ) return 1;
}
```

abc.txt



array.c

```
#include <stdio.h>

// for effective code it is not required to use switch as variable declaration is not allowed in swich and as iD , 2D and 3D have different variables requirement hence use Array function and write all conditions there , At last final code is given

/*
int Array(int num)
{
  if (num == 1)
  {
    printf("1D Array ");

    /*
      int size;
    // int OneD[] = {};
    printf("\n1D Array is a single row of elements.\n Enter the size of 1D : ");
    scanf("%d", &size);//if this size input is taken in the switch case then it would create error for printing the array hence where size is declared take the input there.
    int OneD[size];
    printf("Enter the elements of 1D Array:\n");
    for (int i = 0; i < size; i++)
    {
      scanf("%d", &OneD[i]);
      }
    */

/*
}
else if (num == 2)
{
 printf("2D Array ");
}
else if (num == 3)
{
 printf("3D Array ");
}
else
{
 printf("Invalid Input ");
}
}
```

```
int main()
{
int num;
printf("Enter the number of dimensions: ");
scanf("%d", &num);
Array(num);

// if(num==1){

int size;
// int OneD[] = {};
printf("\n1D Array is a single row of elements.\n Enter the size of 1D : ");
scanf("%d", &size); // if this size input is taken in the switch case then it would create error for printing the array hence where size is declared take the input there.
int OneD[size];
// }


switch (num)
{

case 1:

//  int OneD[] = {};//error: a label can only be part of a statement and a declaration is not a statement
// int size;//error: a label can only be part of a statement and a declaration is not a statement

// int OneD[size];
// int OneD[size]={};//error

// int OneD[];//error
// int OneD[10]; // no error

printf("Enter elements in 1D :\n");

for (int i = 0; i < size; i++)
{//size will be undefined if it will be put in if condition in main

  // int element;
  printf("element at index %d :", i);
  // scanf("%d", &element);
  // OneD[i] = element;
  scanf("%d", &OneD[i]);
  // continue;
}
```

```
 printf("elements in !D array are: ");

 for (int i = 0; i < size; i++)
 // for (int i=0; i < length; i++)
 { // not executes at this case statement as size and OneD[] is declared outside
switch
   printf("%d \t ", OneD[i]);
 }

 break;

case 2:
case 3:
dafault:
 printf("Not valid dimension:");
}


return 0;
}

*/



int arrayInC(int num)
{

  if (num == 1)
  {
    printf("ready to create 1D array\n");
    int size;
    printf("Enter size of 1D array:");
    scanf("%d", &size);
    int OneD[size];
    for (int i = 0; i < size; i++)
    {
      printf("Enter the element at index %d : ", i);
      scanf("%d", &OneD[i]);
    }
    printf("1D array:\n");
    for (int i = 0; i < size; i++)
    {
      printf("%d \t", OneD[i]);
    }
  }
  else if (num == 2)
  {
    printf("ready to create 2D array\n");
    int rows, cols;
    printf("Enter number of rows in 2D :");
```

```
    scanf("%d", &rows); // scanf if present in the next line line the next print
will be executed only after enter in scanf hence , new line is automatically
created after scanf, hence not needed to use \n , it will create extra space
    printf("Enter number of columns in 2D :");
    scanf("%d", &cols);
    int TwoD[rows][cols];
    for (int i = 0; i < rows; i++)
    {
      for (int j = 0; j < cols; j++)
      {
        printf("Enter the element at row %d and column %d , TwoD[%d][%d] : ", i, j,
i, j);
        scanf("%d", &TwoD[i][j]); // if no data is entered then it will take the
addresses as the value and execute all the code to the last
      }
    }
    printf("2D array:\n");
    for (int i = 0; i < rows; i++)
    {
      for (int j = 0; j < cols; j++)
      {
        printf("%d \t", TwoD[i][j]);
      }
      printf("\n");
    }

    /*   In case when no input is given
    Enter number of rows in 2D :3
    Enter number of columns in 2D :2
    Enter the element at row 0 and column 0 , TwoD[0][0]
    : cd "c:\Sudhadocuments\C files\C learning"
    Enter the element at row 0 and column 1 , TwoD[0][1]
    : Enter the element at row 1 and column 0 , TwoD[1][0]
    : Enter the element at row 1 and column 1 , TwoD[1][1]
    : Enter the element at row 2 and column 0 , TwoD[2][0]
    : Enter the element at row 2 and column 1 , TwoD[2][1]
    : 2D array:
    32      0       4200031         4210837         6356648         0
    */

    /*
    // final output
    Enter the dimension of Array :2
    ready to create 2D array
    Enter number of rows in 2D :3
    Enter number of columns in 2D :2
    Enter the element at row 0 and column 0 , TwoD[0][0] : 1
    Enter the element at row 0 and column 1 , TwoD[0][1] : 2
    Enter the element at row 1 and column 0 , TwoD[1][0] : 3
    Enter the element at row 1 and column 1 , TwoD[1][1] : 4
    Enter the element at row 2 and column 0 , TwoD[2][0] : 5
```

```
    Enter the element at row 2 and column 1 , TwoD[2][1] : 6
    2D array:
    1       2
    3       4
    5       6
    */
  }

  else if (num == 3)
  {
    printf("ready to create 3D array\n");
    int rows, cols, layers;
    printf("Enter number of rows in 3D :");
    scanf("%d", &rows);
    printf("Enter number of columns in 3D :");
    scanf("%d", &cols);
    printf("Enter number of layers in 3D :");
    scanf("%d", &layers);

    int ThreeD[layers][rows][cols];

    for (int k = 0; k < layers; k++)
    {
      for (int i = 0; i < rows; i++)
      {
        for (int j = 0; j < cols; j++)
        {
          printf("Enter the element at layer %d , row %d and column %d , ThreeD[%d]
[%d][%d] : ", k, i, j, k, i, j);
          scanf("%d", &ThreeD[k][i][j]);
        }
      }
    }
    printf("3D array:\n");
    for (int k = 0; k < layers; k++)
    {
      for (int i = 0; i < rows; i++)
      {
        for (int j = 0; j < cols; j++)
        {
          printf("%d \t", ThreeD[k][i][j]);
        }
        printf("\n");
      }
      printf("\n");
    }

    /* Output
    Enter the dimension of Array :3
    ready to create 3D array
    Enter number of rows in 3D :4
```

```
    Enter number of columns in 3D :2
    Enter number of layers in 3D :2
    Enter the element at layer 0 , row 0 and column 0 , ThreeD[0][0][0] : 2
    Enter the element at layer 0 , row 0 and column 1 , ThreeD[0][0][1] : 3
    Enter the element at layer 0 , row 1 and column 0 , ThreeD[0][1][0] : 3
    Enter the element at layer 0 , row 1 and column 1 , ThreeD[0][1][1] : 2
    Enter the element at layer 0 , row 2 and column 0 , ThreeD[0][2][0] : 3
    Enter the element at layer 0 , row 2 and column 1 , ThreeD[0][2][1] : 2
    Enter the element at layer 0 , row 3 and column 0 , ThreeD[0][3][0] : 3
    Enter the element at layer 0 , row 3 and column 1 , ThreeD[0][3][1] : 2
    Enter the element at layer 1 , row 0 and column 0 , ThreeD[1][0][0] : 3
    Enter the element at layer 1 , row 0 and column 1 , ThreeD[1][0][1] : 2
    Enter the element at layer 1 , row 1 and column 0 , ThreeD[1][1][0] : 3
    Enter the element at layer 1 , row 1 and column 1 , ThreeD[1][1][1] : 2
    Enter the element at layer 1 , row 2 and column 0 , ThreeD[1][2][0] : 3
    Enter the element at layer 1 , row 2 and column 1 , ThreeD[1][2][1] : 2
    Enter the element at layer 1 , row 3 and column 0 , ThreeD[1][3][0] : 3
    Enter the element at layer 1 , row 3 and column 1 , ThreeD[1][3][1] : 2
    3D array:
    2       3
    3       2
    3       2
    3       2

    3       2
    3       2
    3       2
    3       2
    */
  }

  else
  {
    printf("Not valid dimension");
  }
}



int main()
{
  int num;
  printf("Enter the dimension of Array :");
  scanf("%d", &num);
  arrayInC(num);

  // Array reversal
  int arrToRev[] = {2, 5, 3, 4, 3, 43};
  arrayRev(arrToRev);

  return 0;
```

```
}
```

atoi_Custom.c

```c
// C program to Implement Custom atoi()
// ascii to integer

#include <stdio.h>

int atoi_Conversion(const char* strg)
{

  // Initialize res to 0
  int res = 0;
  int i = 0;

  // Iterate through the string strg and compute res
  while (strg[i] != '\0') {
    printf("i=%d res = %d\n",i,res);
    res = res * 10 + (strg[i] - '0');//10 is used make initial res at one digit place right appending the new in right place //res * 10  + (strg[i] - '0') is used to convert the character to integer implicitly as res * 10 is integer and (strg[i] - '0') is string and as int is first hence whole will be converted into integer.

    i++;
  }
  return res;
}

int main()
{
  // const char strg[] = "12345";
  const char strg[] = "5654";
  int value = atoi_Conversion(strg);

  // print the Converted Value
  printf("String to be Converted: %s\n", strg);
  printf("Converted to Integer: %d\n", value);
  return 0;
}
```

atoi.c

```
/*
In C, atoi stands for ASCII To Integer. The atoi() is a library function in C that
converts the numbers in string form to their integer value. To put it simply, the
atoi() function accepts a string (which represents an integer) as a parameter and
yields an integer value in return.

C atoi() function is defined inside <stdlib.h> header file.

Syntax of atoi() Function
int atoi(const char *strg);
Parameters
strg: The string that is to be converted to an integer.

Note: The atoi function takes in a string as a constant (Unchangeable) and gives
back the converted integer without modifying the original string.

Return Value
The atoi() function returns the following value:

An equivalent integer value by interpreting the input string as a number only if
the input string str is Valid.
If the conversion is not valid, the function returns 0.

Note: In the case of an overflow the returned value is undefined.


*/

// C program to illustrate the use of atoi()

#include <stdio.h>
#include <stdlib.h>
#include <string.h> //to use strcpy

int main()
{

  // string to be converted
  char strToConvert[] = "908475966";

  // converting string using atoi()
  int ConvertedStr = atoi(strToConvert);

  // printing the Converted String
  printf("String to be Converted: %s\n", strToConvert);
  printf("Converted to Integer: %d\n", ConvertedStr);
```

```
  /*
  Explanation: The string “99898989” is converted to the corresponding integer
value, and both the original string and the returned integer value are printed.

  String to be Converted: 908475966
Converted to Integer: 908475966

  */

  //    if the input string contains non-numeric characters so the function returns
0.

  // string to be converted
  char strToConvertStr[] = "SudhaforShanaya";

  // converting string using atoi()
  int ConvertedStrStr = atoi(strToConvertStr);

  // printing the Converted String
  printf("String SudhaforShanaya to be Converted: %s\n", strToConvertStr);
  printf("Converted to Integer: %d\n", ConvertedStrStr);

  /*
  String SudhaforShanaya to be Converted: SudhaforShanaya
Converted to Integer: 0
  */

  //    The atoi() function does not recognize decimal points or exponents it just
takes the integer ignoring the decimals


  int res_val;
  char inp_str[30];

  // Initialize the input string
  strcpy(inp_str, "12.56");

  // Converting string to integer using atoi()
  res_val = atoi(inp_str);

  // print result
  printf("Input String = %s\nResulting Integer = %d\n",
      inp_str, res_val);

  /*
  Input String = 12.56
  Resulting Integer = 12
  */

  //  Passing partially valid string results in conversion of only integer part.
(If non-numerical values are at the beginning then it returns 0)
```

```
  // Initializing the input string
  strcpy(inp_str, "1234adsnds");

  // Convert string to integer using atoi() and store the
  // result in result_value
  res_val = atoi(inp_str);

  printf("Input String = %s\nResulting Integer = %d\n",
      inp_str, res_val);

  /*
  Input String = 1234adsnds
  Resulting Integer = 1234
  */

  //  Passing string beginning with the character '+' to atoi() then '+' is ignored
and only integer value is returned.

  // Initializing the input string
  strcpy(inp_str, "+23234");

  // Convert string to integer using atoi() and store the
  // result in result_value
  res_val = atoi(inp_str);

  printf("Input String = %s\nResulting Integer = %d\n",
      inp_str, res_val);

  /*
  Input String = +23234
  Resulting Integer = 23234
  */

  return 0;
}
```

C learning\Automated_Bill_Generator.c

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

// this is a string
char * replaceWord(const char * str, const char * oldWord, const char * newWord)
{
  char * resultString;//final return value will be this and it gets printed in output using %s as pointer is used
  int i, count = 0;
  int newWordLength = strlen(newWord);
  int oldWordLength = strlen(oldWord);


  // Lets count the number of times old word occurs in the string
  for (i = 0; str[i] !='\0'; i++)//this will keep the string of fgets content

  { //indexing works here as str[i] is an array of char
    if (strstr(&str[i], oldWord) == &str[i]) //char *strstr (const char *s1, const char *s2);//as strstr takes pointer parameter hence address is passed to  //&str[i] gives the mains string //basically strstr returns the pointer and if this point is also pointed by &str[i] then it it gets into if condition satisfied
    {
      count++;

      // Jump over this word
      i = i + oldWordLength - 1; //1 is subtracted as strlen  counts from 1 to n and index takes from 0
    }
  }

  // Making a new string to fit in the replaced words
  resultString = (char *)malloc(i + count * (newWordLength - oldWordLength) + 1);//i stores the point where the old word ended and from there the difference size is also created //inshort resultString stores whole new file size from start to end of file //if newwordlength is less than oldlength then (newWordLength - oldWordLength) give -ve value and size will be decreased in that case

  i = 0;
  while (*str)//str is the file content from fgets
  {
    // Compare the substring with result
    if(strstr(str, oldWord) == str)
    {
      strcpy(&resultString[i], newWord);
```

```
      i += newWordLength;//i is taken from 0 of the file//here i is responsible for
writing the content in the actual bill generated

      str += oldWordLength;//str pointer will move end of oldword
    }
    else{//when nothing has to substitute
      resultString[i] = *str;//it is just traversing
      i += 1;//i is taken from 0 of the file
      str +=1;
    }
  }
  resultString[i] = '\0';
  return resultString; //it is a kind of pointer result hence it gets stored in
pointer newStr created below

}

int main()
{
  FILE * ptr = NULL;
  FILE * ptr2 = NULL;
  ptr = fopen("bill.txt", "r");
  ptr2 = fopen("genBill.txt", "w");
  char str [200];
  fgets(str, 200, ptr);
  printf("The given bill template was:\n %s\n\n", str);

  // Call the replaceWord function and generate newStr
  char * newStr;//as function defined replaceWord is also pointer
  newStr = replaceWord(str, "{{item}}", "body massager");
  newStr = replaceWord(newStr, "{{outlet}}", "phenoxi Shanaya Bazaar");
  newStr = replaceWord(newStr, "{{name}}", "Sudha");
  printf("The actual bill generated is:\n %s\n\n", newStr);

  printf("The generated bill has been written to the file genBill.txt\n");


  fprintf(ptr2, "%s", newStr);

  fclose(ptr);
  fclose(ptr2);
  return 0;
}
```

C learning\automated_Receipt_Generator.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>

int main()
{



  FILE *ptr;
  ptr = fopen("message.txt", "w+");//it overwrites
  if (ptr == NULL)
  {
    printf("Unable to access file");
    return 1;
  }
  else
  {
    printf("File opened successfully\n\n");
   fputs("Thanks {{name}} for purchasing {{item}} from our outlet {{outlet}}.
Please visit our outlet {{outlet}} for any kind of problems. We plan to serve you
again soon.",ptr);


  }

int str[10];
  fseek(ptr,7,SEEK_SET);
 // fgets(str, 10, ptr);
//  printf("%s",str);
  printf("\n\n");
  fputs("  Sudha  ", ptr);
    fclose(ptr);

   ptr = fopen("message.txt", "r+");
  if (fgets(str, 180, ptr) != NULL) {
    puts(str);
  }


  fclose(ptr);
```

```
// The strchr() function returns a pointer to the first occurrence of c that is converted to a character in string. The function returns NULL if the specified character is not found.

// In C/C++, std::strstr() is a predefined function used for string matching. <string.h> is the header file required for string functions. This function takes two strings s1 and s2 as arguments and finds the first occurrence of the string s2 in the string s1. The process of matching does not include the terminating null-characters(‘\0’), but function stops there.

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

// Parameters
// s1: This is the main string to be examined.
// s2: This is the sub-string to be searched in string.
// Return Value
// This function returns a pointer point to the first character of the found s2 in s1 otherwise a null pointer if s2 is not present in s1.
// If s2 points to an empty string, s1 is returned.



/*
const char *filename = "message.txt"; // Change to your file name
  const char *word_to_find = "name";   // Change to the word you want to find

  #define BUFFER_SIZE 1024

  char buffer[BUFFER_SIZE];
  int fd;
  ssize_t bytes_read;

  // Open the file for reading
  fd = open(filename, O_RDONLY);
  if (fd == -1) {
    perror("Error opening file");
    return EXIT_FAILURE;
  }

  // Read the file in chunks
  while ((bytes_read = read(fd, buffer, sizeof(buffer) - 1)) > 0) {
    buffer[bytes_read] = '\0'; // Null-terminate the buffer

    // Search for the word in the buffer
    if (strstr(buffer, word_to_find) != NULL) {
      printf("Word '%s' found in the file.\nand buffer is :%d\nBytes read :%d",
word_to_find,buffer,bytes_read);
      break; // Exit the loop if the word is found

    }
  }
```

```
  if (bytes_read == -1) {
    perror("Error reading file");
  } else if (bytes_read == 0) {
    printf("Word '%s' not found in the file.\n", word_to_find);


    /*
    Word 'name' found in the file.
and buffer is :6355728
Bytes read :161
    */
 /*  }  */
/*
  // Close the file
  close(fd);
  return EXIT_SUCCESS;
*/

/*
Explaination:
Include Headers: The necessary headers are included for file handling and string manipulation.
Define Constants: Constants for the buffer size and the filename are defined.
Open the File: The file is opened using open(). Error handling is included to check if the file was opened successfully.
Read the File: A loop reads the file in chunks of BUFFER_SIZE bytes using read(). The buffer is null-terminated after reading to treat it as a string.
Search for the Word: The strstr() function is used to search for the specified word in the buffer. If found, a message is printed, and the loop breaks.
Handle End of File: If the end of the file is reached without finding the word, a message is printed.
Close the File: Finally, the file is closed using close().
*/

  printf("");
  printf("\n");

  return 0;
}
```

C learning\bill.txt

Thanks {{name}} for purchasing {{item}} from our outlet {{outlet}}. Please visit our outlet {{outlet}} for any kind of problems. We plan to serve you again soon.

C learning\binary_Write_Read.c

```c
// C program to write to a binary file
#include <stdio.h>
#include <stdlib.h>

// Struct declared
struct Num {
  int n1, n2;
};

// Driver code
int main()
{
   // variables declared
  int n;
  struct Num obj;

   // File Pointers declared
  FILE* fptr;

   // Failure Condition
  if ((fptr = fopen("tempT.bin", "wb")) == NULL) { //this binary will not be understandable by you as it is stored in binary form but if you type something directly in binary then it is understandable
    printf("Error! opening file");

    // if file pointer returns NULL program
    // will exit
    exit(1);
  }


  for (n = 1; n < 10; n++) {
    obj.n1 = n;
    obj.n2 = 12 + n;

   // Data written
    fwrite(&obj, sizeof(struct Num), 1, fptr);
  }

   // File closed
```

```
  fclose(fptr);

  printf("Data in written in Binary File\n\n");



// C Program to Read from a binary file using fread()
#include <stdio.h>
#include <stdlib.h>
struct Num
{
  int n1, n2;
};



 if ((fptr = fopen("temp.bin", "rb")) == NULL)
 {
  printf("Error! opening file");

  // If file pointer will return NULL
  // Program will exit.
  exit(1);
 }

 // else it will return a pointer
 // to the file.
 for (int n = 1; n < 10; ++n)
 {
  fread(&obj, sizeof(struct Num), 1, fptr);
  printf("n1: %d\tn2: %d\n", obj.n1, obj.n2);

  /*
  Data in written in Binary File

n1: 1   n2: 13
n1: 2   n2: 14
n1: 3   n2: 15
n1: 4   n2: 16
n1: 5   n2: 17
n1: 6   n2: 18
n1: 7   n2: 19
n1: 8   n2: 20
n1: 9   n2: 21


but if you have saved the bin file by typing youself then this will not give the
required result as it think whatever written in it is in binary and it will try to
convert it into normal text that is already written normal , hence it halts
  */
 }
```

```
 fclose(fptr);




  return 0;
}
```

C learning\binTry.c

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

struct threeNum{
  int n1, n2, n3;
};

int main(){
  int n;
  struct threeNum num;
  FILE *fptr;
  if ((fptr = fopen("program.bin","rb")) == NULL){
    printf("Error! opening file");
    // Program exits if the file pointer returns NULL.
    exit(1);
    }
   for(n = 1; n <= 5; ++n){

    //  fread(&num, sizeof(struct threeNum), 1, fptr);

    printf("&num %d is %s\n",n,&num);//&num 1 is ¢→@: //&num 2 is n1  n2  n3  ûsu☻://NOTE:`ûsu☻ is terminator//&num is pointing to the structure defination address at first then in next loop , it will refer to the 0th index value in binary file and so on increase due to fread updation
    printf("&num %d is %d\n",n,&num);//&num 1 is 6356748: //it will always point to the base address of num as %d is used in every loop

    fread(&num, sizeof(struct threeNum), 1, fptr);//here 1 is the count for number of 12 bytes to be read from file pointed by the pointer//we are updating the values in the base address directly , you can think of it as scanf used &a to put the value at the address of a //it will read only 12 bytes in an iteration
    //  NOTE: here we are putting whole 12 bytes to &num not individual n1, n2, n3 parts hence num.n1 will give the address at that point
    printf("sizeof(struct threeNum) is %d\n",sizeof(struct threeNum));//sizeof(struct threeNum) is 12 //as it contains three integer and each is of 4 bytes hence 3*4=12 bytes
   printf("&num %d is %s:\n",n,&num);//&num 1 is n1  n2  n3 //&num will give value here as it is in binary file hence &num which gives address will give the value due to be in binary file
   printf("&num %d is %d:\n",n,&num);//&num 1 is 6356748://it will point to the addressa and will be same always as base address is not changed , we only changed the value at this address

    printf("n1: %d\tn2: %d\tn3: %d\n", num.n1, num.n2, num.n3);//n1: 538980718 n2: 538980974   n3: 168637294
    printf("n1: %d\tn2: %d\tn3: %d\n", &num.n1, &num.n2, &num.n3);// n1: 6356748 n2: 6356752     n3: 6356756
```

```
     printf("n1: %d\tn2: %d\tn3: %d\n", *&num.n1, *&num.n2, *&num.n3);////n1: 538980718   n2: 538980974   n3: 168637294  //same as num.n1

    //  printf("n1: %s\tn2: %s\tn3: %s\n",&num[0],&num[1],&num[2]);//it will give error in usinf [] anywhere

   }
   fclose(fptr);
   return 0;
}

//Code submitted by Susobhan Akhuli
```

C learning\commandLine_Arg.c

// Run this in terminal open from file loaction directly or from vscode terminal that we use to see the output but the ther you have to give lines by yourself to see the arguements but in case of no arguements it will show the output with direct run command

/*
// Introduction to Command Line Arguments in C

The command line is a text interface for your computer that allows you to enter commands for immediate execution. A command-line can perform almost everything that a graphical user interface can. Many tasks can be performed more rapidly and are easier to automate.

Let's say we have a weather forecasting application developed in any language. When we run the program, it will display a graphical user interface (GUI) where you can enter the city name and hit the ENTER button to know about the current weather. But if you don't have that GUI, which means you can't click on any buttons, that is where command-line arguments come into play, where we pass the parameters in the terminal box to do any actions.

For example:

weather "Delhi" and hit the ENTER key, this will show you the current weather.

For example:

You can navigate around your computer's files and directories using the command line.
The command line can be scripted to automate complex tasks, like the example given below:
If a user wants to put 50+ files' data into a file, this is a highly time-consuming task. Copying data from 50+ files, on the other hand, can be done in less than a minute with a single command at the command line. And many more..

Syntax:
cd “Directory name”, like cd desktop

What are Command-Line Arguments?

Command-line arguments are simple parameters that are given on the system's command line, and the values of these arguments are passed on to your program during program execution. When a program starts execution without user interaction, command-line arguments are used to pass values or files to it.

What are Command-Line Arguments in C?

When the main function of a program contains arguments, then these arguments are known as Command Line Arguments.

The main function can be created with two methods: first with no parameters (void) and second with two parameters. The parameters are argc and argv, where argc is an integer and the argv is a list of command line arguments.

argc denotes the number of arguments given, while argv[] is a pointer array pointing to each parameter passed to the program. If no argument is given, the value of argc will be 1.
The value of argc should be non-negative.
Syntax:
* Main function without arguments:
  int main()

* Main function with arguments:
  int main(int argc, char* argv[])

// Properties of Command Line Arguments in C

Command line arguments in C are passed to the main function as argc and argv.
Command line arguments are used to control the program from the outside.

argv[argc] is a Null pointer.

The name of the program is stored in argv[0],
 the first command-line parameter in argv[1], and the last argument in argv[n].

argc (ARGument Count) is an integer variable that stores the number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, the value of argc would be 2 (one for argument and one for program name)
The value of argc should be non-negative.
argv (ARGument Vector) is an array of character pointers listing all the arguments.
If argc is greater than zero, the array elements from argv[0] to argv[argc-1] will contain pointers to strings.
argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments.

Command-line arguments are useful when you want to control your program from outside rather than hard coding the values inside the code.
To allow the usage of standard input and output so that we can utilize the shell to chain commands.
To override defaults and have more direct control over the application. This is helpful in testing since it allows test scripts to run the application.

```
What are command-line arguments?

They are user inputs that are passed to the program via standard input.
They are program settings that are stored in a configuration file.
They are parameters passed to a program through the system's command line during
program execution.
They are options that can be set in the program's properties window.


*/

#include <stdio.h>
int main(int argc, char *argv[]) {
  int i;
  printf("\nProgram name: %s", argv[0]);

  // if (argc < 2) {//means when no aguement is passed and argc 1 will store the
program name

  //or

  if (argc == 1){
    printf("\nNo Extra Command Line Argument Passed "
        "Other Than Program Name");


    printf("\n\nNo argument passed through command line!");
  }

  /*
   else {
    printf("\nArgument supplied: \n");
    for (i = 1; i < argc; i++){
      printf("%s\t", argv[i]);

      //OR

      // printf("%s\n", argv[i]);
    }

  }
*/
  // or instead of else you can have used if again
if (argc >= 2) {
    printf("\nNumber Of Arguments Passed: %d", argc);
    printf("\n----Following Are The Command Line "
        "Arguments Passed----");
    for (int i = 0; i < argc; i++)
      printf("\nargv[%d]: %s", i, argv[i]);
  }
```

```
}

/*
// Output in Different Scenarios


// 1)Without Argument: When we run the above code and don’t pass any argument,
let’s see the output our code generates.

PS C:\Sudhadocuments\C files\C learning> gcc commandLine_Arg.c

PS C:\Sudhadocuments\C files\C learning> .\commandLine_Arg.exe

Program name:C:\Sudhadocuments\C files\C learning\commandLine_Arg.exe

No argument passed through command line!

// 2)Pass single argument: When we pass the single argument separated by space but
inside the double quotes or single quotes. Let’s see the output.


Properties of Command Line Arguments in C
They are passed to the main() function.
They are parameters/arguments supplied to the program when it is invoked.
They are used to control programs from outside instead of hard coding those values
inside the code.
argv[argc] is a NULL pointer.
argv[0] holds the name of the program.
argv[1] points to the first command line argument and argv[argc-1] points to the
last argument.
Note: You pass all the command line arguments separated by a space, but if the
argument itself has a space, then you can pass such arguments by putting them
inside double quotes “” or single quotes ”.




PS C:\Sudhadocuments\C files\C learning> .\commandLine_Arg.exe "Hello Sudha"

Program name:C:\Sudhadocuments\C files\C learning\commandLine_Arg.exe
Argument supplied: Hello Sudha
PS C:\Sudhadocuments\C files\C learning>

// 3)Pass more than single argument: When we run the program by passing more than
single argument, say three arguments. Let’s see the output:
PS C:\Sudhadocuments\C files\C learning> .\commandLine_Arg.exe Hello Shanaya how
are you

Program name:C:\Sudhadocuments\C files\C learning\commandLine_Arg.exe
```

```
Argument supplied: Hello        Shanaya how     are     you


*/
```

C learning\comparator_function_qsort.c

```
/*
// Comparator Function in qsort()
The key point about qsort() is the comparator() function. The comparator function
takes two arguments and contains logic to decide their relative order in the sorted
output. The idea is to provide flexibility so that qsort() can be used for any type
(including user-defined types) and can be used to obtain any desired order
(increasing, decreasing, or any other).

The comparator function takes two pointers as arguments (both type-casted to const
void*) and defines the order of the elements by returning (in a stable and
transitive manner

// Prototype of comparator() function
int comparator(const void* p1, const void* p2);

// Parameters
p1 and p2: These are the pointer to the elements to be compared.
// Return Value
The comparator function should only return the following values:

( <0 ): Less than zero, if the element pointed by p1 goes before the element
pointed by p2.
( 0 ): Zero, if the element pointed by p1 is equivalent to the element pointed by
p2.
( >0 ): Greater than zero, if the element pointed by p1 goes after the element
pointed by p2.
*/



// C program to illustrate the use of comparator function in
// qsort()
#include <stdio.h>
#include <stdlib.h>

/*
struct Student
{
  int age, marks;
  char name[20];
};
```

```
//  to sort the students based on marks in ascending order. The comparator function
will look like this:

int comparator(const void* p, const void* q)
{
  int l = ((struct Student*)p)->marks;
  int r = ((struct Student*)q)->marks;
 return (l - r);
}

*/



// This function is used in qsort to decide the relative
// order of elements at addresses p and q.
int comparator(const void* p, const void* q)
{
  // Get the values at given addresses
  int l = *(const int*)p;
  int r = *(const int*)q;
 // both odd, put the greater of two first.
  if ((l & 1) && (r & 1))
    return (r - l);

  // both even, put the smaller of two first
  if (!(l & 1) && !(r & 1))
    return (l - r);

  // l is even, put r first
  if (!(l & 1))
    return 1;

  // l is odd, put l first
  return -1;
}


// A utility function to print an array
void printArr(int arr[], int n)
{
  int i;
  for (i = 0; i < n; ++i)
    printf("%d ", arr[i]);
}

// Driver program to test above function
int main()
{
  int arr[] = { 1, 6, 5, 2, 3, 9, 4, 7, 8 };
```

```c
  int size = sizeof(arr) / sizeof(arr[0]);
  qsort((void*)arr, size, sizeof(arr[0]), comparator);

  printf("Output array is\n");
  printArr(arr, size);

  /*
Output array is
9 7 5 3 1 2 4 6 8
  */

  return 0;
}
```

C learning\desktop.ini

```
[LocalizedFileNames]
hello.c=@hello,0
```

C learning\dynamic_memory.c

```c
#include <stdio.h>
#include <stdlib.h>
// <stdlib.h> is used for dynamic memory allocation

int main()
{

  // This pointer will hold the base address of the block created //* ptr hold value but ptr holds address
  int *ptr;
  int *ptr2;
  int n, i;

  // Get the number of elements for the array
  printf("Enter number of elements:");
  scanf("%d", &n);
  printf("Entered number of elements: %d\n", n);

  // Dynamically allocate memory using malloc()
  ptr = (int *)malloc(n * sizeof(int)); // ptr holds address //as int* ptr declared hence (int*)malloc declared to keep the datatype of pointer same

  ptr2 = (int *)calloc(n, sizeof(int));
  ptr2 = realloc(ptr2, 2 * n); // if entered number of elements is 8 then it will create 8*2 elements size now

  // Get the new size for the array
  n = 10;
  printf("\n\nEnter the new size of the array: %d\n", n);
  // Dynamically re-allocate memory using realloc()
  ptr = (int *)realloc(ptr, n * sizeof(int)); // here it is just written like malloc but instead of n , ptr used which represents the old size and n * sizeof(int) represents the new size

  if (ptr == NULL)
  {
    printf("Reallocation Failed\n");
    exit(0);
  }

  // Check if the memory has been successfully
  // allocated by malloc or not
  if (ptr == NULL || ptr2 == NULL)
  {

    printf("Memory not allocated.\n");
    exit(0);
```

```
  }
  else
  {

    // Memory has been successfully allocated
    printf("Memory successfully allocated using malloc or calloc.\n");

    printf("&ptr %d \n", &ptr); //&ptr 6356760  //here the base address has only the difference of 4 that is size of int*
    printf("&ptr %x \n", &ptr); // 60ff18  //here the base address has only the difference of 4 that is size //&ptr 60ff18

    // Get the base address of the block created

    printf("*&ptr %x \n", *&ptr); //*&ptr ff19e8
    printf("*&ptr %d \n", *&ptr); //*&ptr 16718312//here we observe that base address of first block is  8133096 and if number of elements entered is n then it will allocate 4*8 bytes more ==32 bytes and as this very first address is assigned then we only need to assign for 7 other elements and becomes 16718312 + (7*4) ==16718340
    // OR
    printf("(unsigned int)ptr %x \n", (unsigned int)ptr); //(unsigned int)ptr ff19e8//same as *&ptr

    printf("&ptr2 %d \n", &ptr2);   //&ptr2 6356756
    printf("&ptr2 %x \n", &ptr2);   //&ptr2 60ff14
    printf("*&ptr2 %x \n", *&ptr2); //*&ptr2 ff1a68
    printf("*&ptr2 %d \n", *&ptr2); //*&ptr2 16718440 //after leaving 100bytes , this new allocation started //here 100 bytes is leaved so that in case of reallocation ,it can be extended from old but not exactly , realloc starts from garbage value

    // the difference of 4 in the addresses is because int* ptr; is of 4 bytes only

    // Get the elements of the array
    for (i = 0; i < n; ++i)
    {
      ptr[i] = i + 1;  // we are putting the values in array
      ptr2[i] = i + 1; // we are putting the values in array
    }

    // Print the elements of the array
    printf("The elements of the array are: ");
    for (i = 0; i < n; ++i)
    {
      printf("%d ", ptr[i]);


      printf("&ptr %d \n", &ptr[i]);

      /*
```

```
&ptr 16718312
&ptr 16718316
&ptr 16718320
&ptr 16718324
&ptr 16718328
&ptr 16718332
&ptr 16718336
&ptr 16718340
```

```
      */
      printf("&ptr2 %d \n", &ptr2[i]);
      /*
```

```
&ptr2 16718440
&ptr2 16718444
&ptr2 16718448
&ptr2 16718452
&ptr2 16718456
&ptr2 16718460
&ptr2 16718464
&ptr2 16718468
```

```
      */

      printf("*&ptr %d \n", *&ptr[i]);//it returns the value
      /*
```

```
1 *&ptr 1
2 *&ptr 2
3 *&ptr 3
4 *&ptr 4
5 *&ptr 5
6 *&ptr 6
7 *&ptr 7
8 *&ptr 8
      */
    }

    for (i = 0; i < 2 * n; ++i)
    { // to check for realloc as size chacged to 2n
      printf("&ptr2[%d] %d \n", i, &ptr2[i]);
      /*
```

```
&ptr2[0] 15800936
&ptr2[1] 15800940
&ptr2[2] 15800944
&ptr2[3] 15800948
&ptr2[4] 15800952
&ptr2[5] 15800956
&ptr2[6] 15800960
&ptr2[7] 15800964
&ptr2[8] 15800968
&ptr2[9] 15800972
&ptr2[10] 15800976
```

```
&ptr2[11] 15800980
&ptr2[12] 15800984
&ptr2[13] 15800988
&ptr2[14] 15800992
&ptr2[15] 15800996
      */
    }

    // Print the elements of the array
    printf("\nThe elements of the array are: ");
    for (i = 0; i < n; ++i)
    {

      printf("%d ", ptr2[i]);
    }



free(ptr);//here as else part creates hence free here in else part



 // Free the memory

  free(ptr);

  printf("Malloc Memory successfully freed.\n");

  // Free the memory
  free(ptr2);
  printf("Calloc Memory successfully freed.\n");

  }

  /*
  Enter number of elements:8
  Entered number of elements: 8
  Memory successfully allocated using malloc.
  The elements of the array are: 1 2 3 4 5 6 7 8
  */



  return 0;

  /*
    // C malloc() method
  // The “malloc” or “memory allocation” method in C is used to dynamically
allocate a single large block of memory with the specified size. It returns a
pointer of type void which can be cast into a pointer of any form. It doesn’t
```

Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

```
ptr = (cast-type*) malloc(byte-size)
For Example:
ptr = (int*) malloc(100 * sizeof(int));
```

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory(total 400 bytes).

```
// If space is insufficient, allocation fails and returns a NULL pointer.
```

C calloc() method

“calloc” or “contiguous allocation” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:

It initializes each block with a default value ‘0’.

It has two parameters or arguments as compare to malloc().

```
ptr = (cast-type*)calloc(n, element-size);
here, n is the no. of elements and element-size is the size of each element.

ptr = (float*) calloc(25, sizeof(float));
This statement allocates contiguous space in memory for 25 elements each with the size of the float.

// If space is insufficient, allocation fails and returns a NULL pointer.
```

```
// C free() method
// The “free” method in C is used to deallocate the memory that was previously allocated using
malloc() or calloc(). It takes a pointer to the memory block to be deallocated as an argument.The memory allocated using functions malloc() and calloc() is not de-allocated on their own.
// Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

// For Example:
// free(ptr);
```

```
// C realloc() method
“realloc” or “re-allocation” method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to dynamically re-allocate memory. re-allocation of memory maintains the
```

```
already present value and new blocks will be initialized with the default garbage
value.

  ptr = realloc(ptr, newSize);
  where ptr is reallocated with new size 'newSize'.


  */
}
```

C learning\Employees_Manager.c

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int chars, i = 0;
  char a, b;
  char *ptr;
  while (i < 3) //we let to enter 3 employees only

  {
    /*Syntax ch = getchar( ); where ch is a char Var. When this function is executed, the computer will wait for a key to be pressed and assigns the value to the variable when the “enter” key pressed. putchar( ) function is used to display one character at a time on the monitor.*/

    printf("Employee %d: Enter the number of characters in your Employee Id\n", i + 1);
    scanf("%d", &chars);
    getchar();
    printf("Enter the value of a\n");
    scanf("%c", &a);
    getchar();
    printf("Enter the value of b\n");
    scanf("%c", &b);
    ptr = (char *)malloc((chars + 1) * sizeof(char));//+1 is used to store the terminator \0
    printf("Enter your Employee Id\n");
    scanf("%s", ptr);
    printf("Your Employee Id is %s\n", ptr);
    free(ptr);
    i = i + 1;
  }

  return 0;

  /*Problem Statement:-
Suppose ABC is a private limited company which manages the employee records of other companies. Employee id can be of any length, and it can contain any character. The following are the task you have to perform for three employees.

Take the length of an employee id as an input and store it in an integer length variable
Take an employee id as an input and display it on the screen.
Save the employee id in a character array, which is allocated dynamically.
Create only one character array dynamically.
```

```
Program Example:-
Employee1;
Enter the no. of characters in your employee id: 45
//dynamically allocate the character array
Take input from the user: //employee_id

Employee2;
Enter the no. of characters in your employee id: 5
//dynamically allocate the character array
Take input from the user: //employee_id

Employee3;
Enter the no. of characters in your employee id: 9
//dynamically allocate the character array
Take input from the user: //employee_id

You have to reallocate the memory in such a way so that your program is robust and
uses less memory.


*/

  // don't write anything even comments below main() otherwise it will be executing
infinite times without termination and give abrupt result
}
```

C learning\entering_Marks_Realloc.c

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int index = 0, i = 0, n, *marks; // this marks pointer hold the base address of
the block created
  int ans;
  marks = (int*)malloc(sizeof(int)); // dynamically allocate memory using malloc
  // check if the memory is successfully allocated by
  // malloc or not?
  if (marks == NULL) {
    printf("memory cannot be allocated");
  }
  else {
    // memory has successfully allocated
    printf("Memory has been successfully allocated by "
        "using malloc\n");
    printf("\n marks = %pc\n",marks);// marks = 007619E8c // print the base or
beginning address of allocated memory //%p format specifier is used to write the
address in  007619E8c form
    do {
      printf("\n Enter Marks\n");
      scanf("%d", &marks[index]); // Get the marks
      printf("would you like to add more(1/0): ");
      scanf("%d", &ans);

      if (ans == 1) {
        index++;
        marks = (int*)realloc(marks,
          (index + 1)* sizeof(int)); // Dynamically reallocate// memory by using
realloc

          // check if the memory is successfully allocated by realloc or not?
        if (marks == NULL) {
          printf("memory cannot be allocated");
        }
        else {
          printf("Memory has been successfully "
              "reallocated using realloc:\n");
          printf(
            "\n base address of marks are:%pc",
            marks); ////print the base or
                ///beginning address of
                ///allocated memory
        }
      }
    } while (ans == 1);
    // print the marks of the students
```

```
    for (i = 0; i <= index; i++) {
      printf("marks of students %d are: %d\n ", i,
          marks[i]);
    }
    free(marks);
  }
  return 0;
}
```

C learning\factorial.c

```c
#include <stdio.h>

int fact(int num){
  if (num==0){
    return 1; //base case
  }
  else{
    return num*fact(num-1);
  }
}

int main(){
int num;
printf("Enter the number to find it's factorial : ");
scanf("%d", &num);
fact(num);
printf("Factorial of %d is %d", num, fact(num));
}
```

C learning\fflush.c

```c
// C program to illustrate situation
// where flush(stdin) is required only
// in certain compilers.
#include <stdio.h>
#include<stdlib.h>
int main()
{
  // char str[20];
  char str[50];
  int i;
  for (i=0; i<2; i++)
  {
    printf("Enter a string %d:\n",i);
    scanf("%[^\n]s", str);
    printf("%s\n", str);
    // fflush(stdin);

  }


  /*
  The code above takes only single input and gives the same result for the second input. Reason is because as the string is already stored in the buffer i.e. stream is not cleared yet as it was expecting string with spaces or new line. So, to handle this situation fflush(stdin) is used.



  Output :
  Enter a string 0:
Hello I am Sudha //(it let you to write only index 0 part and lower will be taken automatically with the buffer)
Hello I am Sudha //result of printf
Enter a string 1:
Hello I am Sudha(taken automatically from buffer , you have not typed it infect it doesnot let you to type)
PS C:\Sudhadocuments\C files\C learning>
  */



// C program to illustrate flush(stdin)
// This program works as expected only
// in certain compilers like Microsoft
// visual studio.

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

```
  {
    printf("Enter a string %d:\n",i);
    scanf("%[^\n]s", str); //this will take from above buffer hence doesnot let you
to type in i= 0 here
    printf("%s\n", str);

    // used to clear the buffer
    // and accept the next string
    fflush(stdin); //it let you to type in next i=1
  }


/*
Enter a string 0:
hey, I am doing great(typed by you)
hey, I am doing great
Enter a string 1:
hey, I am doing great

Enter a string 0:
hey, I am doing great
Enter a string 1:
I am happy to be here
I am happy to be here
PS C:\Sudhadocuments\C files\C learning>
*/


  return 0;
}

/*
fflush() is typically used for output stream only. Its purpose is to clear (or
flush) the output buffer and move the buffered data to console (in case of stdout)
or disk (in case of file output stream).


Below is its syntax.

fflush(FILE *ostream);

ostream points to an output stream
or an update stream in which the
most recent operation was not input,
the fflush function causes any
unwritten data for that stream to
be delivered to the host environment
to be written to the file; otherwise,
the behavior is undefined.
```

```
// using fflush for input stream like stdin

While taking an input string with spaces, the buffer does not get cleared for the
next input and considers the previous input for the same. To solve this problem
fflush(stdin) is. used to clear the stream/buffer.



Although using “fflush(stdin)” after “scanf()” statement also clears the input
buffer in certain compilers, it is not recommended to use it as it is undefined
behavior by the language standards. In C and C++, we have different methods to
clear the buffer


*/
```

C learning\fibonacci.c

```c
#include <stdio.h>

/*
int fab(int a, int b)
{
  int c;
  // if (a == 0 && b == 1)
  // {
  //     if (a == 0)
  //     {
  //         return a;
  //     }
  //     else if (b == 1)
  //     {
  //         return 1;
  //     }
  // }
  // else
  // {

    c = a + b;


    return c;



  // }


}

int main()
{
  int size;
  printf("Enter the numbers of the fabonacii to get : ");
  scanf("%d", &size);
  int a = 0, b = 1 ,c;
printf("%d\t%d\t",a,b);

  for (int i = 0; i < size-2; i++) //or
  //for(int i=2;i<size;++i)//loop starts from 2 because 0 and 1 are already printed
  {
    // printf("%d\t", fab(a, b));
    // fab(a, b); // not working properly

    c=a+b; //works properly
```

```
    printf("%d\t", fab(a, b));
    a = b;
    // b = c;//works properly
    b=fab(a, b);//not works properly//it takes 3 as like it is recalling it hence
it updates b as 1 and 1 created earlierthen again 1 , it is not like setting ther
value hence shows error.
    printf("\nfab(a, b):%d\n",fab(a, b));

  }


// int i=0;
//     while(i < size-2){
// printf("%d\t", fab(a, b));
//         a = b;
//         b = fab(a, b);
//         i++;
//     }

  return 0;
}


*/

// final code

/*
//without recursion
int main(){
  int a=0,b=1,c,size;
  printf("Enter the numbers of the fabonacii to get : ");
  scanf("%d",&size);
  printf("%d\t%d\t",a,b);
  for(int i=0;i<size-2;i++){
    c=a+b;
    printf("%d\t",c);
    a=b;
    b=c;
  }

return 0;
}

*/

/*
// with recursion and breakdown
int fab(int num)
{ // 5
  static int n = 0, m = 1, p; //only for this fab defined
```

```c
  if (num > 0)
  {
    p = n + m;
    printf("%d\t", p);
    n = m;
    m = p;
    fab(num-1);//4
  }

}

int main()
{
  int a = 0, b = 1, c, size;
  printf("Enter the numbers of the fabonacii to get : ");
  scanf("%d", &size); // 7
  printf("%d\t%d\t", a, b);

  fab(size - 2); // 5
}
*/

// completely with recursion even first two element and find the element at certain
position

int fab(int num)
{ // 7

  if (num == 1 || num == 2)
  {
    // printf("%d\t", num - 1);
    return num - 1;
  }
  else // 3
  {

    // printf("%d\t", fab(num-1)+fab(num-2));
    return fab(num - 1) + fab(num - 2);
  }
}

int main()
{
  int a = 0, b = 1, c, position;
  printf("Enter the element position of the fabonacii to get : ");
  scanf("%d", &position); // 7
  // fab(position);          // 7
  printf("%d", fab(position));

return 0;
```

```
}
```

C learning\file_Size.c

```
// C program to find the size of file
#include <stdio.h>

long int findSize(char file_name[])
{
  // opening the file in read mode
  FILE *fp = fopen(file_name, "r");

  // checking if the file exist or not
  if (fp == NULL)
  {
    printf("File Not Found!\n");
    return -1;
  }

  fseek(fp, 0L, SEEK_END); // fp will go to the end of file in one go and this file
position will be tracked by ftell //don't think as one by one go in each loop
because it is stupid to think such

  // 0L is a long integer value with all the bits set to zero - that's generally
the definition of 0 .

  /*
  fseek() is used to move the file pointer associated with a given file to a
specific position.

  Syntax of fseek()
  The fseek() syntax is:

  int fseek(FILE *pointer, long int offset, int position);


  Parameters
  pointer: It is the pointer to a FILE object that identifies the stream.

  offset: It is the number of bytes to offset from the position

  // as offset is purakarna or complete , pay karna//offset is also to make the
effect of something less strong or noticeable
  // In programming, an offset is a value or position that can be added or
subtracted from a starting point to access data or perform calculations. Offsets
```

```
are commonly used in computer engineering and low-level programming, such as
assembly language

  position: It is the position from where the offset is added. Position defines the
point with respect to which the file pointer needs to be moved. It has three
values:

  SEEK_END: It denotes the end of the file.
  SEEK_SET: It denotes starting of the file.
  SEEK_CUR: It denotes the file pointer’s current position.

  Return Value

  It returns zero if successful, or else it returns a non-zero value.
  */

  // calculating the size of the file
  long int res = ftell(fp); // it is used to obtain the current file position of a
stream. It returns a long integer value representing the current position in the
file, measured in bytes from the beginning. This function is particularly useful
for determining the position to later return to a specific location in a file.

  printf("res  : %ld\n", res); // res  : 20 if new line is present even when no
character is there in new line //if no new line and only 10, 20, 30, 40, 50 present
in single line then it takes 18 bytes

  // closing the file
  fclose(fp);

  return res;
}

// Driver code
int main()
{
  // char file_name[] = { "a.txt" };
  char file_name[] = {"input.bin"};

  // all charaters other than alphabets and decimal will be ignored in any file
like here input.bin is the file name and array will take only index one value

  // input.bin contains

  /*
  10, 20, 30, 40, 50

  */

  /*

  NOTE:
```

```
  Input :  file_name = "a.txt"
       Let "a.txt" contains "geeks"
  Output : 6 Bytes
  There are 5 bytes for 5 characters then an extra
  byte for end of file.

  Input :  file_name = "a.txt"
       Let "a.txt" contains "geeks for geeks"
  Output : 16 Bytes
  // space will take 1 byte//hence 5+1+3+1+5+1=16

  // but this end of file is not consider when using fseek current position, it is
counted when use sizeof()

  */

  printf("the sizeof( space) %d bytes\n\n",sizeof("  "));//the sizeof( space): 2
bytes //3 bytes if 2 spaces //it is because it is also considering an extra byte
for end of file.


  long int res = findSize(file_name);
  if (res != -1)
    printf("Size of the file is %ld bytes \n", res); // //Size of the file is 20
bytes
  return 0;



  /*
  NOTE: space takes 1 byte because ecah character will take one space as it is
converted into string
  new line after enter created will take 2 bytes
  10 will take 2 bytes that is 1 B for 1 B for 0

  */
}
```

C learning\fileIO.c

```c
// C program to Open a File, Write in it, And Close the File
#include <stdio.h>
#include <string.h>
#include <stdlib.h> //contains exit(0)

// NOTE: all the unwanted output will be just because of the file which is being read or write hence other things like size should be as much as needed is good but it would not create unwanted output even if size is given more than need

// try to keeo the fread in block so that it donot affect the lower outputs

// all the unwanted output is because of size mismatched
// the file that has to perform I/O should have proper size defined and known
// all the characters will be taken as 1 byte even white space and end of line will take 2 bytes




// text file

// "C:\Windows\WinSxS\x86_netfx4-attributionfile_b03f5f7f11d50a3a_4.0.15912.0_none_bfd0663f65a3dd07\ThirdPartyNotices.txt"
// binary file

// To read all values of n1, n2 & n3 of the bin, we are using fscanf()

struct threeNum
{
  int n1, n2, n3;
};

int main()
{

  // Declare the file pointer
  FILE *filePointer;

  // Get the data to be written in file

  // char dataToBeWritten[50] = "Sudha is very passionate about her work .She is a determined person having a great vision";

  /*
  fileIO.c: In function 'main':
fileIO.c:20:32: warning: initializer-string for array of chars is too long
```

```
   char dataToBeWritten[50] = "Sudha is very passionate about her work .She is a
determined person having a great vision";

^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~
  */

  //    increase the size of dataToBeWritten[50]

  char dataToBeWritten[] = "//Sudha is very passionate about her work .She is a
determined person having a great vision"; // since it's c file hence we added / to
comment

  // Open the existing file GfgTest.c using fopen()
  // in write mode using "w" attribute
  filePointer = fopen("FileTryInFileIO.c", "w"); // it removes all the existing
content but thereafter every fput will append the data

  // Check if this filePointer is null
  // which maybe if the file does not exist but as it is in write mode it will
create the file//it is for write pointer
  if (filePointer == NULL)
  {
    printf("FileTryInFileIO.c file failed to open.");
  }
  else
  {

    printf("The FileTryInFileIO.c is now opened.\n");

    // Write the dataToBeWritten into the file
    if (strlen(dataToBeWritten) > 0)
    {

      // writing in the file using fputs()
      fputs(dataToBeWritten, filePointer);   // overwrite all the content of
initial file as it is opened in write mode
      fputs("\n", filePointer);              // it is like append a \n in code not
overwrite , it is for terminating
      fputs("dataToBeWritten", filePointer); // it is like append a \n in code not
overwrite , it is for terminating
      fputs(dataToBeWritten, filePointer);   // it is like append a \n in code not
overwrite , it is for terminating
      // fputs(, filePointer); //it is like append a \n in code not overwrite , it
is for terminating
    }

    // Closing the file using fclose()
    fclose(filePointer);

    printf("Data successfully written in file "
```

```
        "FileTryInFileIO.c\n");
    printf("The FileTryInFileIO.c is now closed.\n");

    /*
    The file is now opened.
    Data successfully written in file FileTryInFileIO.c
    The file is now closed.
    */
  }

  // This program will create a file named FileTryInFileIO.c if not exist in the
same directory as the source file which will contain the following text:

  printf("\n\n");

  FILE *fileReadPointer;
  fileReadPointer = fopen("FileReadTryInFileIO.c", "r");

  char str[80];



  // checking if the file is opened successfully

  if (fileReadPointer == NULL)
  {
    printf("The file is not opened. The program will "
        "now exit.");
    exit(0); // Exit Success: Exit Success is indicated by exit(0) statement which
means successful termination of the program, i.e. program has been executed without
any error or interrupt.

    // The file is not opened. The program will now exit.

    // it satisfy if condition when file doesnot exist because read mode can't
create a file when in doesnot exists that write mode does

    // the rest of the codes will not execute if it gets into if condition and exit
the program

    // we get the below statement when this file donot exist
    // The file is not opened. The program will now exit.
  }
  else
  {
    printf("The file is opened in read mode\n");


    // The file is opened in read mode//as we have created the file that has to be
read
```

```
    if(fgets(str, 80, fileReadPointer) != NULL)//fgets is used in if to test if
string is present in it
 {
  puts(str);//it will print first 80 bytes
 }
 fclose(fileReadPointer);
  }

  printf("\n\n");

  int a;
  printf("enter value to read :");
  scanf("%*s %d", &a); // The %*s in scanf is used to ignore some input as
required. In this case, it ignores the input until the next space or newline.
Similarly, if you write %*d it will ignore integers until the next space or
newline.
  printf("Input value read : a=%d\n", a);

  /* Here 5434343534 was ignores and only 4553445 is taken only because it gets
into new line  after 5434343534 and scanf() stops reading when it encounters a new
line.


  enter value to read :5434343534
4553445
Input value read : a=4553445

// OR with space

enter value to read :fdfsdf 2
Input value read : a=2

//OR even if string is not enter , it will take any non whitespace as string and
wait for next string to print

enter value to read :1 2
Input value read : a=2



  */

  printf("\n");

  // int fscanf(FILE *ptr, const char *format, ...)

  FILE *ptr = fopen("abc.txt", "r");
  if (ptr == NULL)
  {
    printf("no such file.");
    return 0;
```

```
  }

  /* Assuming that abc.txt has content in below
    format
    NAME    AGE   CITY
    abc     12    hyderabad
    bef     25    delhi
    cce     65    bangalore */

  char buf[100]; // it will give banglore again and again due to it's size and it
will print for every repaint hence limit the length of buf //

  // char buf[50];

  // printf("fscanf  : %d \n",fscanf(ptr, "%*s %*s %s ", buf));//fscanf  : 1 //it
will put the value into buf once and now ptr will point to next line each time
fscanf(ptr, "%*s %*s %s ", buf) is shown , even for printing//fscanf(ptr, "%*s %*s
%s ", buf) returned 1 means it has successfully assigned a value

  // Note: don't print any pointer or any character thet will later on store the
data from anoter file by reading otherwise if print is used then it will print it
repeadly after each repaint in below codes.

  // printf("buf outside the while loop : %s\n", buf);//buf outside the while
loop : √

  while (fscanf(ptr, "%*s %*s %s ", buf) == 1) // this while loop will take the %s
once beacuse it has used the expression and buf always updates from the last
point //hence if once updated to city then next time it check for hydrabad
//whatever fsacnf get , it is storing in buf and printin it

  // while (fscanf(ptr, "%*s %*s %s ", buf) != 1) //it will keep printing city
  { // this condition will check for all the time till fscanf get a string  as
fscanf returns 1 when it encounter the data // as we have learned earlier %*s %*s
is used to ignore string array two times , one string array finishes when any space
is encountered and till new character is not encounted (it do not take any space as
charater and it keep ignoring as nothing , but string ignore is activated only when
it encounters character ) .After ignoring two strings in a line it stores the third
string in buf variable, buf is declared char datatype to store %s value //

    // printf("fscanf  : %d \n",fscanf(ptr, "%*s %*s %s ", buf));//fscanf  : 1
//delhi as fscanf used 3rd time and as this will return one because it has assigned
delhi to buf now, hence it will restart the while loop and ther in condition it
will loose bangalore and next time fscanf  : -1 and then printing  fscanf(ptr, "%*s
%*s %s ", buf) after -1 will give banglore because it updates from the last
point //for hyderabad
    printf("%s\n", buf);
    // fclose(ptr);//this will not let the while loop to execute for second
iteration hence it will print delhi again and again
```

```
    // as we already know when while loop is applied directly on some address data like by using pointer then it do not need any increment or decrement for next iteration of while loop
  }

  // must to close the file otherwise
  fclose(ptr);

  printf("\n\n");

  // https://www.reddit.com/r/C_Programming/comments/10crrku/need_help_with_binary_files_structures_and_fread/

  // check above link to handle binary file

  // fread() function to read the content of binary file into an array.

  FILE *fileT;
  // int buffer[5];//it gives axtra v symbol
  int buffer[0]; // it works also

  // Open the binary file for reading
  fileT = fopen("input.bin", "rb");
  if (fileT == NULL)
  {

    perror("Error opening file"); // The perror() function prints an error message to stderr . If string is not NULL and does not point to a null character, the string pointed to by string is printed to the standard error stream, followed by a colon and a space.

    // Error opening file: No such file or directory //this output we will get and that's too like normal output black gray not in red color

    return 1; // exit the int main() hence below codes will not execute
  }

  // Read the integers from the file into the buffer

  // fread(buffer, sizeof(int), 5, fileT);
  // fread(buffer, 1, 5, fileT);//Element 1: 10, 2lore
  // fread(buffer, 2, 5, fileT);//Element 1: 10, 20, 30`
  // fread(buffer, 18, 1, fileT);
  // fread(buffer, 3, 5, fileT);//Element 5: 10, 20, 30, 40,u
  // fread(buffer, 2, 6, fileT);
  // fread(buffer, 3, 6, fileT);
  // fread(buffer, 4, 6, fileT);//Element 5: 10, 20, 30, 40, 50

  // only buffer and &buffer will give the same result because direct address is affected or element buffer is affected and all the no updates will be made because
```

```
whole file is read at once hence bytes wise updation is not required hence  buffer and &buffer will give same result
  // fread(&buffer, sizeof(int),5, file);
  // Print the integers that were read

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

    fread(buffer, sizeof(int), 5, fileT);//Element 1: 10, 20, 30, 40, 50 // put fread in for loop or in any type of block otherwu=ise it will affect the other codes also
    /*
Element 1: 10, 20, 30, 40, 50
Element 2:
Element 3:
Element 4:
Element 5:

as the fread inside the for loop hence when it finishes reading 20 bytes as ((sizeof(int)=4 )*5=20),thereafter it is left with nothing to read
    */
    // printf("Element %d: %d\n", i + 1, buffer[i]); //it gives address
    /*
    Element 1: 539766833
    Element 2: 539766834
    Element 3: 539766835
    Element 4: 539766836
    Element 5: 2109493
    */

    // printf("Element %d: %s\n", i + 1, buffer[i]);//halt here as %s used

    // printf("Element %s: %d\n", i + 1, buffer[i]);//halt here as %s used

    // printf("Element %d: %d\n", i + 1, &buffer); // 6356508 is address of buffer and always

    // printf("Element %d: %s\n", i + 1, &buffer); // 6356508 is address of buffer and always if %d used //Element 1: 10, 20, 30, 40, 50  , if %s is used

    // printf("Element %d: %d\n", i + 1, buffer); // 6356508 is address buffer always
    printf("Element %d: %s\n", i + 1, buffer); // Element 1: 10, 20, 30, 40, 50 , when %s is used
    // fclose(file);
  }

  // Close the file

  fclose(fileT);
```

```
  /*
Element 1: 539766833
Element 2: 539766834
Element 3: 539766835
Element 4: 539766836
Element 5: 16265269
  */

  // binTry.c file contains the detailed explaination of below code

  // To read all values of n1, n2 & n3 of the bin, we are using fscanf()
  printf("\n\n");

  // struct threeNum num;
  struct threeNum *num;

  printf("sizeof(struct threeNum) : %d bytes\n", sizeof(struct threeNum)); //
sizeof(struct threeNum) : 12 bytes //this size of because of 4 int defined in it

  FILE *fptr;

  fptr = fopen("program.bin", "rb");

  printf("fopen('program.bin', 'rb) :%s \n\n", fptr = fopen("program.bin",
"rb")); // fopen('program.bin', 'rb) :
  // it gives nothing

  if ((fptr = fopen("program.bin", "rb")) == NULL)
  // if (fptr== NULL)//also works
  {
    printf("Error! opening file"); // Error! opening file
    // Program exits if the file pointer returns NULL.

    exit(1); // 1 or EXIT_FAILURE: The statements exit(1) and exit(EXIT_FAILURE)
mean that the program terminated abruptly with an error and no below codes will
execute but all the above code till this will be executed
  }

  for (int n = 1; n <= 5; ++n) // executes 4 times
  {
    // fread(&num, sizeof(struct threeNum), 1, fptr);
    // fread(&num, sizeof(struct threeNum), 0, fptr);//&num : ╨+■
    // fread(&num, 12, 1, fptr);//&num : ╨+■
    /*

&num : n1  n2  n3
30, 40, 50
&num : 1   5   6
30, 40, 50
&num : 2   10  11
30, 40, 50
```

```
&num : 3   15  16
30, 40, 50
&num : 4   20  21
30, 40, 50

*/
    // fread(&num, 4, 1, fptr);//
    fread(&num, 12, 1, fptr); //

    // here 1 is the count for number of 12 bytes(2 bytes for line terminator at the end and oter each character + space all takes 1 byte) to be read from file pointed by the pointer  // here as we need updation hence &num is used otherwise num would not keep the last value

    // printf("n1: %d\tn2: %d\tn3: %d\n", num.n1, num.n2, num.n3);//it gives only address not the values

    printf("&num : %s \n", &num); // return 12 bytes at a time //above reault would also have given the same output only the problem was because of fread outside the for loop above
    /*
&num : n1  n2  n3

&num : 1   5   6

&num : 2   10  11

&num : 3   15  16

&num : 4   20  21
    */

    // printf("&num : %d \n", &num); //return 12 bytes at a time //it gives addresses of num

    /*
    &num : 6356516
    &num : 6356516
    &num : 6356516
    &num : 6356516
    */

    // fread(num, sizeof(struct threeNum), 1, fptr);//halt and exit

    // printf("num : %s \n\n", num); // num : 6356516, it will terminate after printing this address only once
  }

  fclose(fptr);

  /*
```

```
  n1: 538980718   n2: 846077984   n3: 538976288
  n1: 168637294   n2: 538976305   n3: 538981664
  n1: 221650976   n2: 538980874   n3: 808525856
  n1: 538976288   n2: 168636721   n3: 538976307
  */

  printf("End\n");

  return 0;
}
```

C learning\FileReadTryInFileIO.c

```
// Heleroefjeiregjfdvldsvdvvfbgfbfd
// vdfgn
// vdfgn
// vdfgn
// vdfgn
// vdfgn
```

C learning\FileTryInFileIO.c

```
//Sudha is very passionate about her work .She is a determined person having a
great vision
dataToBeWritten//Sudha is very passionate about her work .She is a determined
person having a great vision
```

C learning\freadTry.c

```c
#include <stdio.h>

typedef struct {
  int id;
  float value;
} Record;

void writeRecords(const char *filename) { //writeRecords(filename);
  FILE *file = fopen(filename, "wb");
  Record records[] = {
    {1, 10.5},
    {2, 20.5},
    {3, 30.5}  // Last record
  };

  fwrite(records, sizeof(Record), 3, file);
  fclose(file);
}

 /*
void readRecords(const char *filename) {
  FILE *file = fopen(filename, "rb");
  Record record;

  while (fread(&record, sizeof(Record), 1, file) == 1) {
    printf("ID: %d, Value: %.2f\n", record.id, record.value);
  }

  fclose(file);
}
*/


void readRecords(const char *filename) {
  FILE *file = fopen(filename, "rb");
  Record record;

  // Read until end of file
  while (fread(&record, sizeof(Record), 1, file) == 1) {
    printf("ID: %d, Value: %.2f\n", record.id, record.value);
  }

  fclose(file);
}
```

```c
int main() {

  const char *filename = "data.bin";
  writeRecords(filename);
  readRecords(filename);

/*
ID: 1, Value: 10.50
ID: 2, Value: 20.50
ID: 3, Value: 30.50
*/


  return 0;
}
```

C learning\fseek_rewind.c

```c
// C program to implement
// fseek
#include <stdio.h>
#include <stdlib.h>

// Driver code
int main()
{
   // string declared
  char str[80];

   // File Pointer declared
  FILE* ptr;

   // File Opened
  ptr = fopen("Hello.txt", "w+");//write + read

   // Puts data in File
   fputs("Welcome to Sudha's place", ptr);

   // fseek function used
  fseek(ptr, 11, SEEK_SET);//it points to seek set thet is 11th position

   // puts Shanaya's palce in place of at position defined
   // After 11 elements
  fputs("Shanaya's palce  ", ptr);
  fclose(ptr);

   // Reading the file
  ptr = fopen("Hello.txt", "r+");
  if (fgets(str, 80, ptr) != NULL) {
    puts(str);
  }



// rewind() in C

//    rewind() function sets the file pointer to the beginning of the file.
// void rewind(FILE *stream);
 // Rewind function called  Sets ptr to beginning //it is used after fgets which put the pointer at some other position as per thr bytes to be read , like fgets(str, 80, ptr) puts the ptr after 80 th position hence next time when fget is used then it will start for that position  but if we want the fgets from start then we need to use rewind

  rewind(ptr);
```

```c
  if (fgets(str, 200, ptr) != NULL) {
    puts(str);//Welcome to Shanaya's palce
  }



   // Close the opened file
  fclose(ptr);

  /*
  Welcome to Shanaya's palce
  */







  return 0;
}
```

C learning\function_Pointer.c

```
/*
// Function Pointer in C

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

Following are some interesting facts about function pointers.


1) Unlike normal pointers, a function pointer points to code, not data. Typically a
function pointer stores the start of executable code.

2) Unlike normal pointers, we do not allocate de-allocate memory using function
pointers.




*/

#include <stdio.h>

#include <stdlib.h>//qsort is function in stdlib.h library

#include <stdbool.h>//search function in it

// A normal function with an int parameter and void return type
void fun(int a) { printf("Value of a is %d\n", a); } // Value of a is 10



// for 4th and 5the property
void add(int a, int b)
{
  printf("Addition is %d\n", a + b);
}
void subtract(int a, int b)
{
  printf("Subtraction is %d\n", a - b);
}
void multiply(int a, int b)
{
  printf("Multiplication is %d\n", a * b);
}
```

```c
// for property 6
// A simple C program to show function pointers as parameter

// Two simple functions
void fun1() { printf("Fun1\n"); }
void fun2() { printf("Fun2\n"); }

// A function that receives a simple function
// as parameter and calls the function
void wrapper(void (*fun)()) { fun(); }



//// An example for qsort and comparator

// A sample comparator function that is used for sorting an integer array in
ascending order.
// To sort any array for any other data type and/or criteria, all we need to do is
write more compare functions. And we can use the same qsort()


int compare(const void* a, const void* b)
{
  return (*(int*)a - *(int*)b);
}



// A compare function that is used for searching an integer array

bool compareT(const void* a, const void* b)
{
  return (*(int*)a == *(int*)b);
}

// General purpose search() function that can be used
// for searching an element *x in an array arr[] of
// arr_size. Note that void pointers are used so that
// the function can be called by passing a pointer of
// any type. ele_size is size of an array element
int search(void* arr, int arr_size, int ele_size, void* x,
      bool compareT(const void*, const void*))
{
  // Since char takes one byte, we can use char pointer
  // for any type/ To get pointer arithmetic correct,
  // we need to multiply index with size of an array
  // element ele_size
  char* ptr = (char*)arr;

  int i;
  for (i = 0; i < arr_size; i++)
```

```
    if (compare(ptr + i * ele_size, x))
      return i;

  // If element not found
  return -1;
}









int main()
{
  // fun_ptr is a pointer to function fun()
  void (*fun_ptr)(int) = &fun;

  // If we remove bracket, then the expression “void (*fun_ptr)(int)” becomes “void
*fun_ptr(int)” which is declaration of a function that returns void pointer.

  /* The above line is equivalent of following two
  void (*fun_ptr)(int);
  fun_ptr = &fun;
  */

  // Invoking fun() using fun_ptr
  (*fun_ptr)(10); // Value of a is 10

  //  This point in particular is very useful in C. In C, we can use function
pointers to avoid code redundancy. For example a simple qsort() function can be
used to sort arrays in ascending order or descending or by any other order in case
of array of structures. Not only this, with function pointers and void pointers, it
is possible to use qsort for any data type.

  // Comparator function of qsort() in C
  // the function uses QuickSort algorithm to sort the given array. Following is
the prototype of qsort()
/*
void qsort (void* base, size_t num, size_t size, int (*comparator)(const
void*,const void*));


continue in comparator_function_qsort file
*/
```

```
  // 3) A function's name can also be used to get functions' address. For example,
in the below program, we have removed address operator '&' in assignment. We have
also changed function call by removing *, the program still works.

  void (*fun_ptr_2)(int) = fun; // & removed

  fun_ptr_2(10); // * removed
          // Value of a is 10

  // 4) Like normal pointers, we can have an array of function pointers. Below
example in point 5 shows syntax for array of pointers.

  // 5) Function pointer can be used in place of switch case. For example, in below
program, user is asked for a choice between 0 and 2 to do different tasks.

  // fun_ptr_arr is an array of function pointers
  void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
  unsigned int ch, a = 15, b = 10;// the keyword unsigned modifies the int and char
data types to represent only positive numbers and zero, rather than both positive
and negative numbers://here it represent unsigned integer

  printf("Enter Choice: 0 for add, 1 for subtract and 2 "
      "for multiply\n");
  scanf("%d", &ch);

  if (ch > 2)
    return 0;//something not given in choice

  (*fun_ptr_arr[ch])(a, b);
/*
Enter Choice: 0 for add, 1 for subtract and 2 for multiply
1
Subtraction is 5
*/




// 6) Like normal data pointers, a function pointer can be passed as an argument
and can also be returned from a function.

// For example, consider the following C program where wrapper() receives a void
fun() as parameter and calls the passed function.

wrapper(fun1);
  wrapper(fun2);
/*
Fun1
Fun2
*/
```

```
// This point in particular is very useful in C. In C, we can use function pointers
to avoid code redundancy. For example a simple qsort() function can be used to sort
arrays in ascending order or descending or by any other order in case of array of
structures. Not only this, with function pointers and void pointers, it is possible
to use qsort for any data type.



// // An example for qsort and comparator

  int arr[] = { 10, 5, 15, 12, 90, 80 };
  int n = sizeof(arr) / sizeof(arr[0]);

  qsort(arr, n, sizeof(int), compare);//The function does not return any value, but
modifies the content of the array pointed to by base reordering its elements as
defined by compare.

  for (int i = 0; i < n; i++)
    printf("%d ", arr[i]);

// 5 10 12 15 80 90



// Similar to qsort(), we can write our own functions that can be used for any data
type and can do different tasks without code redundancy. Below is an example search
function that can be used for any data type. In fact we can use this search
function to find close elements (below a threshold) by writing a customized
compareT function.


int arrC[] = { 2, 5, 7, 90, 70 };
  int nSize = sizeof(arrC) / sizeof(arrC[0]);
  int x = 7;
  printf("\nReturned index is %d ",
      search(arrC, nSize, sizeof(int), &x, compareT));

// Returned index is 0




/*
Parameters
base
Pointer to the first object of the array to be sorted, converted to a void*.
num
Number of elements in the array pointed to by base.
size_t is an unsigned integral type.
size
Size in bytes of each element in the array.
```

```
size_t is an unsigned integral type.
compar
Pointer to a function that compares two elements.
This function is called repeatedly by qsort to compare two elements. It shall
follow the following prototype:
1
int compar (const void* p1, const void* p2);
Taking two pointers as arguments (both converted to const void*). The function
defines the order of the elements by returning (in a stable and transitive manner):
return value    meaning
<0  The element pointed to by p1 goes before the element pointed to by p2
0   The element pointed to by p1 is equivalent to the element pointed to by p2
>0  The element pointed to by p1 goes after the element pointed to by p2

For types that can be compared using regular relational operators, a general compar
function may look like:

int compareMyType (const void * a, const void * b)
{
 if ( *(MyType*)a <  *(MyType*)b ) return -1;
 if ( *(MyType*)a == *(MyType*)b ) return 0;
 if ( *(MyType*)a >  *(MyType*)b ) return 1;
}

*/


  return 0;
}
```

C learning\genBill.txt

Thanks Sudha for purchasing body massager from our outlet phenoxi Shanaya Bazaar. Please visit our outlet phenoxi Shanaya Bazaar for any kind of problems. We plan to serve you again soon.

C learning\getch.c

```
/*
getch() is a nonstandard function and is present in conio.h header file which is
mostly used by MS-DOS compilers like Turbo C. It is not part of the C standard
library or ISO C, nor is it defined by POSIX.
Like these functions, getch() also reads a single character from the keyboard. But
it does not use any buffer, so the entered character is immediately returned
without waiting for the enter key.

int getch(void);

Parameters: This method does not accept any parameters.
Return value: This method returns the ASCII value of the key pressed.

getch() method pauses the Output Console until a key is pressed.
It does not use any buffer to store the input character.
The entered character is immediately returned without waiting for the enter key.
The entered character does not show up on the console.
The getch() method can be used to accept hidden inputs like password, ATM pin
numbers, etc.

*/

// Example: To accept hidden passwords using getch()
// Note: Below code won't run on Online compilers, but on MS-DOS compilers like
Turbo IDE.
// C code to illustrate working of
// getch() to accept hidden inputs

#include <conio.h>
#include <dos.h> // delay()
#include <stdio.h>
#include <string.h>

void main()
{

  // Taking the password of 8 characters
  char pwd[9];
  int i;

  // To clear the screen
  clrscr();

  printf("Enter Password: ");
  for (i = 0; i < 8; i++) {

    // Get the hidden input
    // using getch() method
```

```
    pwd[i] = getch();

    // Print * to show that
    // a character is entered
    printf("*");
  }
  pwd[i] = '\0';
  printf("\n");

  // Now the hidden input is stored in pwd[]
  // So any operation can be done on it

  // Here we are just printing
  printf("Entered password: ");
  for (i = 0; pwd[i] != '\0'; i++)
    printf("%c", pwd[i]);

  // Now the console will wait
  // for a key to be pressed
  getch();
}
```

C learning\GlobalfooLink.c

```c
#include <stdio.h>

// GlobalfooLink.c
// In the GlobalfooLink.c file, we declare and define the global variable a as well as the foo function:


int aGloAnother = 100; // Global variable declared and defined here but it is accessed in  GlobalfooLink.c file

// As with function prototypes, we can of course declare extern int a in a header.h file. Incidentally, it is better than defining a global variable directly in the header.

// Of course, having a variable that is accessible by any function in any file of a program might quickly prove to be a security concern. Which is why we can make our global variables static

void foo(void)
{
 aGloAnother = 42;
 printf("Foo: a = %d\n", aGloAnother); // a == 42
}

/*
int main(){ // multiple definition of `main' error if both file are compiled together //hence comment this main

  foo();
}
*/


// gcc globalMain.c GlobalfooLink.c
```

C learning\globalMain.c

```c
#include <stdio.h>

// run code will compile only this file not the file from where extern will take value or defination //hence it always give undefined reference to `foo' , `aGloAnother'

// when using extern then it is must to compile that file along with it and below code must be consecutive

//  gcc globalMain.c GlobalfooLink.c
// ./a.exe

// In the globalMain.c file, we will declare the global variable with the extern keyword, to say that we’re defining this variable elsewhere. It’s a similar declaration to the foo function prototype that we will also define in a separate file. The compiler understands implicitly that it should consider the foo prototype as extern as well.

extern int aGloAnother; // Global variable, defined elsewhere //if this variable is declared extern then using foo(void) function which is declared in another same file need not to write extern if this function is called and it contains the same variable as output which is already declared as extern above



// extern int aGloAnother = 0; ////Error as if a variable is declared extern then it cannot be initialized

void foo(void); // Foo prototype, defined elsewhere
// is identical to
 // extern void foo(void);

extern void foo(void); // Foo prototype, defined elsewhere //Foo: a = 42//from this function defined somewhere //extern before function is necessary if extern is not used above this , like with extern int aGloAnother; if used above then using extern here is not needed hence


int main(void)
{

  int a=4;
  printf("%d\n",a); //4

  //  return (0);
```

```
   //we done it to automatically let it compile and create globalMain.exe file
because if there will be any error then globalMain.exe file will never be made and
it only shows undefined reference to `aGloAnother'

 printf("Main: aGloAnother = %d\n", aGloAnother); // aGloAnother == 100 //due to
extern , it is declared somewhere
 foo(); //undefined reference to `foo
 printf("Main: a = %d\n", aGloAnother); // aGloAnother == 42
 aGloAnother = 200;
 printf("Main: a = %d\n", aGloAnother); // aGloAnother == 200

 return (0);


//  gcc globalMain.c GlobalfooLink.c
// ./a.exe


 /* Final output
Main: aGloAnother = 100
Foo: aGloAnother = 42
Main: aGloAnother = 42
Main: aGloAnother = 200
*/

}
```

C learning\goto.c

```
#include <stdio.h>
int main()
{
  // int a = 3;
  int a = 9;
label:
  printf("Hello, world! less than 5\n");

  if (a < 5)
  {
    a++;
    goto label;
  }

  printf("This is a test.\n");
end:
  printf("This is the end.\n");

  if (a > 5)
```

```
  {
    a--;
    goto end;
    // goto label;
  }

  return 0;
}


/*
output:
Hello, world! less than 5
This is a test.
This is the end.
Hello, world! less than 5
This is a test.
This is the end.
Hello, world! less than 5
This is a test.
This is the end.
Hello, world! less than 5
This is a test.
This is the end.
Hello, world! less than 5
This is a test.
This is the end.
PS C:\Sudhadocuments\C files\C learning> cd "c:\Sudhadocuments\C files\C learning"
PS C:\Sudhadocuments\C files\C learning> cd "c:\Sudhadocuments\C files\C learning\"
; if ($?) { gcc goto.c -o goto } ; if ($?) { .\goto }
Hello, world! less than 5
This is a test.
This is the end.
This is the end.
This is the end.
This is the end.
This is the end.

*/
```