Introduction

Variables used in programs will die at the end of program execution. There may be instances where there is a need to display large data on the console. As the CPU memory is volatile, it is impossible to recover the programmatically generated data again and again. We use files to persist data even after the program execution is completed. **A file represents a sequence of bytes.** File handling process enables us to create, update, read, and delete the files stored on the local file system.
Example: User Login Credentials on Web page. The user ID and password entry is compared with information stored at the time Registration.

The data in a file can be **structured** – stored in the form of rows and columns, file name and values. The data can also be **unstructured** like social media posts. A program in C is itself data for the 'C' compiler. The keyboard and output screen are also considered as files. A file is maintained by the operating system. The operating system decides the naming convention of a file. This is referred to as the physical filename. In a program in a programming language like 'C', we use an identifier to refer to a file. This is called the logical name or the file handle. In a programming language like 'C', we use an **identifier to refer to a file.** This is called the **logical name or the file handle.** There is an opaque type (structure declared in stdio.h) called **FILE – which is a typedef**. This type varies from one implementation to another. We use **FILE* in our program so that our program will not depend on the layout of the type FILE.**

Need of Files

- When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
- If you have to enter a large number of data, it will take a lot of time to enter them every time you run the code.
- However, if you have a file containing all the data, you can easily access the contents of the file using few functions in C.
- To check the output of the program several times is accomplished by running the same program multiple times.
- Usage of file addresses the problem of storing data in bulk

- There are certain programs that require a lot of input from the user and can easily access any part of the code with the help of certain commands.

File Classification

Files are referred as data files in C .There are basically two distinct types of data files available in C

- Text Files
- Binary Files

Text Files

- The simplest form of file created with a **.txt** extension using any simple text editor.
- A text file stores information in the form of ASCII characters internally, but content of the text readable to humans.
- Text files are not secured as the information is in readable form.
- Text files consume large storage space.

Binary Files

- A binary file stores information in the form of the binary number system (0's and 1's) and is created with **.bin** extension.
- Binary files store information in the same way as the information is held in computer memory, hence it occupies less storage.
- Binary Files are not in readable form, Hence, it is safe to to store information in a data file.

Operations on Files

To perform any operation on a file, we connect the physical filename, the logical filename and the mode by using a function **fopen()**

- **Physical Name:** A file is maintained by the OS. The OS decides the naming convention of a file

- **Logical Name:** In a C Program, identifier is used to refer to a file. Also called as File Handle

- **Mode:** Can be read only, write only, append or a combination of these.

Major File handling Functions used is as below:

- Creation of a new file.
- Opening an existing file.
- Reading data from a file.
- Writing data in a file.
- Closing a file.

Let us discuss each one in a detailed section.

Opening a file

- fopen is a C library function used to open an existing file or create a new file.
- The basic format of fopen is:FILE *fopen(const char * filePath, const char * mode);
- filePath: The first argument is a pointer to a string containing the name of the file to be opened.
- mode: The second argument is an access mode.
- fopen function returns NULL in case of a failure and returns a FILE stream pointer on success.
- The mode can be read ,write,append.

Closing a file

- fclose() function is a C library function and it's used to release the memory stream, opened by fopen() function.
- The basic format of fclose(0 function is: int fclose(FILE * stream);
- fclose returns EOF in case of failure and returns 0 on success.

Read /Write operations on Files

File I/O operations are categorized into following types:

- Character I/O
- String I/O
- Formatted I/O
- Block read / write

Character I/O operations on File

fputc()

- The function is used to write at the current file position and then increments the file pointer.
- On success it returns an integer representing the character written, and on error it returns EOF.
- **Syntax:**int fputc(int c,FILE *fp);

fgetc()

- This function reads a character from the file and increments the file pointer position. On success it returns an integer representing the character written, and on error it returns EOF.
- **Syntax:**int fgetc(FILE *fp);

getc() and putc()

- The operation performed by getc() and put() functions is the same as that performed by fgetc() and fputc() functions.
- The difference is fgetc() and fputc() are the macros while getc() and putc() are functions.

.

String I/O operation on File

fputs()

- This function writes the null terminated string pointed to by str to a file.
- This null character that marks the end of string is not written to the file.
- On success it returns the last character written and EOF on error.
- **Syntax:**int fputs(const char *str, FILE *fp);

fgets()

- This function is used to read characters from the file and these characters are stored in the string pointed to by s.
- It reads n-1 characters from the file where n is the second argument.
- fp is a file pointer which points to the file from which character is read.
- This function returns of string pointed to by s on success, and NULL on EOF.
- **Syntax:**char *fgets(char *s,int n, FILE *fp);

Formatted I/O operation on File

fprintf()

- This function is the same as printf () function and writes formatted data into the file instead of the standard output.
- This function has the same parameter as printf () but has one additional parameter which is a pointer of FILE type.
- It returns the number of characters output to the file on success and EOF on error.
- **Syntax:** fprintf (FILE *fp, const char *format [, argument….]);

fscanf()

- This function is the same as scanf () function but it reads data from the file instead of the standard output.
- This function has a parameter which is a pointer of FILE type.
- It returns a number of arguments that were assigned some values on success and EOF on error.
- **Syntax:**fscanf(FILE *fp,const char *format[,address,…..]);

Block read / write operation on File

fwrite()

- The fwrite() function writes the data specified by the void pointer ptr to the file.
- On success, it returns the count of the number of items successfully written to the file,on error, it returns a number less than n.
- **Syntax:**size_t fwrite(const void *ptr, size_t size, size_t n, FILE *fp);

fread()

- fread() function is commonly used to read binary data.
- It accepts the same arguments as the fwrite() function does.
- The syntax of fread() function is as follows:
- **Syntax:** size_t fread(void *ptr, size_t size, size_t n, FILE *fp);
- The function reads n items from the file where each item occupies the number of bytes specified in the second argument.

- On success, it reads n items from the file and returns n. On error or end of the file, it returns a number less than n.

Random access to file

Random access to a file means permitting non-sequential or random access to a file's contents. To access a file randomly, you open the file, seek a particular location, and read from or write to that file.

In order to access a particular file or record , C supports below functions .

1. **fseek()**
2. **ftell()**
3. **rewind()**

fseek()

- This function is used for seeking the pointer position in the file at the specified byte.
- **Syntax:** fseek(file pointer, displacement, pointer position);

  Where ,

  file pointer ---- It is the pointer which points to the file

  displacement ---- It is positive or negative based on the number of bytes, skipped forward or backward position from the current position

  Pointer position -----This sets the pointer position in the file rom where displacement must happen

	Value	pointer position
SEEK_SET	**0**	Beginning of file.
SEEK_CUR	**1**	Current position
SEEK_END	**2**	End of file

ftell()

- This function returns the value of the current pointer position in the file.
- The value is counted from the beginning of the file.
- **Syntax:** ftell(fptr);Where fptr is a file pointer.

rewind()

- The function is used to move the file pointer to the beginning of the given file.
- The function sets the file position to the beginning of the file for the stream pointed to by stream.
- The rewind function does not return anything.
- **Syntax:** rewind(fptr);where fptr is a file pointer.(pending)

Let us write a few example codes to understand the above functions.

Cdocs.txt contains the below data in it.

Good morning all

Welcome to files and file handling section in C Programming

Have a nice day

Stay well and safe

Example 1:**Illustrate fopen()**

```
//Program to open an existing file.
#include<stdio.h>
int main()
{
   FILE *fp;
   fp = fopen("Cdocs.txt","w");
   return 0;
}
```

Example2:Illustrate fclose()

```
#include<stdio.h>
int main()
{
   FILE *fp;
   fp = fopen("Cdocs.txt","w");
   fprintf(fp, "%s", "Sample C File");
   fclose(fp);
```

```
    return 0;
}
```

Example 3: Illustrate getc() and putc()

```
#include <stdio.h>
int main()
{
  char ch;
  FILE *fp;
  if (fp = fopen("test.c", "r"))
  {
        ch = getc(fp);
        while (ch != EOF)
        {
                putc(ch, stdout);// ch on the terminal
                ch = getc(fp);
        }
        fclose(fp);
    }
  return 0;
}
```

Example 4: Illustrate fgetc() and fputc()

```
#include<stdio.h>
int main()
{
 FILE *fp;
 char ch;
 fp = fopen("Cdocs.txt","w");          //Statement   1
 if(fp == NULL)
 {
      printf("\nCan't open file or file doesn't exist.");
 }
```

```
        else
        {
                while((ch=getchar())!=EOF)          //Statement   2
                        fputc(ch,fp);
                printf("\nData written successfully...");
                fclose(fp);
          }

        return 0;

      }
```

Example 5: Illustrate fputs()

```
      #include <stdio.h>
      int main () {
        FILE *fp;
        fp = fopen("Cdocs.txt", "w+");
        fputs("This is c programming.", fp);
        fputs("This is a system programming language.", fp);
        fclose(fp);
          return(0);
      }
```

Example 6: Illustrate fgets()

```
       #include <stdio.h>
      #define MAX 15
      int main()
      {
         char buf[MAX];
         fgets(buf, MAX, stdin);
         printf("string is: %s\n", buf);
         return 0;
      }
```

Example 7: Illustrate fwrite()

Version1: Writing a variable

```
float *f = 100.13;
fwrite(&p, sizeof(f), 1, p);
```

Version2: Writing an array

```
int arr[3] = {101, 203, 303};
fwrite(arr, sizeof(arr), 1, fp);
```

Version 3: Writing some elements of array

```
int arr[3] = {101, 203, 303};
fwrite(arr, sizeof(int), 2, fp);
```

Version 4: Writing structure

```
struct student
{
        char name[10];
        int roll;
         float marks;
};
struct student student_1 = {"Tina", 12, 88.123};

fwrite(&student_1, sizeof(student_1), 1, p);
```

Version 5: Writing array of structure

```
struct student {
        char name[10];
        int roll;
        float marks;
};
struct student students[3] = {
      {"Tina", 12, 88.123},
      {"Jack", 34, 71.182},
                    {"May", 12, 93.713}
                  };
                fwrite(students, sizeof(students), 1, fp);
```

Think about using fread with different types of data as above.

Example 8: Illustrate ftell()

```
     #include<stdio.h>

      int main(){

       /* Opening file in read mode */

       FILE *fp = fopen("Cdocs.txt","r");

       printf("%ld", ftell(fp));   // Printing position of file pointer

         /* Reading first string */

       char string[20];

       fscanf(fp,"%s",string);

         /* Printing position of file pointer again */

       printf("%ld", ftell(fp));

       return 0;

     }
```

Example 9: Illustrate fseek()

```
#include<stdio.h>

int main()
{
     FILE *fp = fopen("data_out.txt","r");
     if(fp == NULL)
          printf("cannot open the file");
```

```
        else
        {
                printf("%d\n",ftell(fp));
                fseek(fp,5,SEEK_SET);   // start from the beginning
                printf("%d\n",ftell(fp)); //5
                //fseek(fp,2,SEEK_SET);
                //printf("%d",ftell(fp)); // 2

                //fseek(fp,2,SEEK_CUR);  //start from the current position of the pointer
                //printf("%d",ftell(fp));  //7

                fseek(fp,-2,SEEK_END);    //start from the end of the file
                printf("%d",ftell(fp));
                putchar(fgetc(fp));
                fclose(fp);
        }
        return 0;
}
```

Example 10: Illustrate rewind()

```
#include <stdio.h>
int main () {
  char str[] = "Hello All";
  FILE *fp;
  char ch;
  /* First let's write some content in the file */
  fp = fopen( "Cdocs.txt" , "w" );
  fputs(str,fp);
  fclose(fp);

  fp = fopen( "Cdocs.txt" , "r" );
  printf("%d", ftell(fp));
  ch = fgetc(fp);
  printf("%c", ch);
  rewind(fp);
  printf("%d", ftell(fp));
  fclose(fp);
  return 0;
}
```

Example 11: Illustrate fprintf() and fscanf()

Version1:

```
#include<stdio.h>
void read(int *a, int n,FILE *fp);
int find_sum(int *a, int n);
int main()
{
        FILE *fp1 = fopen("formatted_data.txt","r");
        FILE *fp2 = fopen("formatted_out.txt","w");
        if(fp1==NULL || fp2 ==NULL)
                printf("issue in opening the file\n");
        else
        {
                int a[500];
                int n = 5;
                read(a,n,fp1);
                int sum = find_sum(a,n);
                //fprintf(stdout,"sum is %d",sum);
                fprintf(fp2,"sum is %d",sum);
        }
        return 0;
}

void read(int *a, int n,FILE *fp)
{
        int i;
        for(i = 0;i<n;i++)
```

```
          {
                    fscanf(fp,"%d",a+i);
          }
}
int find_sum(int *a, int n)
{
          int i;
          int sum = 0;
          for(i = 0;i<n;i++)
          {
                    sum = sum+*(a+i);
          }
          return sum;
}
```

Version 2:

```
//        if we do not specify n, above code fails.
//        So use the return value of fscanf to find n
#include<stdio.h>
int read(int *a,FILE *fp);
int find_sum(int *a, int n);
int main()
{
          FILE *fp1 = fopen("formatted_data.txt","r");
          FILE *fp2 = fopen("formatted_out.txt","w");
          if(fp1==NULL || fp2 ==NULL)
                    printf("issue in opening the file\n");
          else
          {
                    int a[500];
                    int n = read(a,fp1);
                    int sum = find_sum(a,n);
                    //fprintf(stdout,"sum is %d",sum);
                    fprintf(stdout,"sum is %d",sum);
```

```
        }
        return 0;
}

int read(int *a, FILE *fp)
{
        int i = 0;
        while (fscanf(fp,"%d",a+i) != EOF)
                i++;
        return i;
}
int find_sum(int *a, int n)
{
        int i;
        int sum = 0;
        for(i = 0;i<n;i++)
        {
                sum = sum+*(a+i);
        }
        return sum;
}
```

Introduction to Error Handling

Errors are the problems or the faults that occur in the program, which makes the behaviour of the program abnormal, and experienced programmers and developers can also make these faults. **Programming errors** are also known as the **bugs or faults.**

While dealing with files, it is possible that an error may occur. The most common errors that may occur are:

1. **Reading beyond the end-of- file*:*** EOF is a condition in a computer operating system where no more data can be read from a data source. The data source is usually a file or stream.
2. **Performing operations on the file that has not still been opened:** Before performing any operation on a file, you must first open it. An attempt to perform operations on a file that has not been opened yet will lead to error
3. **Writing to a file that is opened in the read mode:** A mode is used to specify whether you want to open a file to perform operations like read, write append etc.
4. **Opening a file with invalid filename:** If you attempt to upload a file that has an invalid file name or type you will receive error.
5. **Write to a write-protected file:** Write-protected file is the ability to prevent new information from being written or old information being changed. In other words, information can be read, but nothing can be added or modified.

Error handling helps during the processing of input-output statements by catching severe errors that might not otherwise be noticed.

Types of Error Handling in C

1. **Global Variable errno:**

   C programming does not provide direct support for error handling but being a system programming language, it provides you access at lower level in the form of return values from the function.

Most of the C function calls return -1 or NULL in case of any error and set an error code global variable errno. When a function is called in C, a variable named as errno is automatically assigned a code (value) which can be used to identify the type of error that has been encountered. Its a global variable **indicating the error occurred during any function call** and defined in the header file **errno.h.**

Different codes (values) for errno mean different types of errors.

errno value	Type of Error
1	Operation not permitted
2	No such file or directory
5	I/O error
7	Argument list too long
9	Bad file number
11	Try again
12	Out of memory
13	Permission denied

2. **perror() and strerror():**

The errno value indicate the types of error encountered. If you want to show the error description, then there are two functions that can be used to display a text message that is associated with errorno.

The functions are:

i. **perror():** The function perror() stands for print error. In case of an error, the programmer can determine the type of error that has occurred using the perror() function. It displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current errno value.

Syntax: void perror (const char *str) // (str: is a string containing a custom message to be printed before the error message itself.)

Let us consider the below code to understand perror function

Example Code 1:

This Program illustrate the use of perror function.

```
#include <stdio.h>
#include <errno.h>
#include<string.h>
int main ()
{
   FILE *fp;
   fp = fopen ("File.txt", "r");  // File with this name doesn't exist
   printf("Value of errno: %d %s\n ", errno, strerror(errno));
   perror("Bad code");
   return 0;
}
```

Note that the function perror**() displays a string passed to it, followed by a colon and the textual message of the current errno value.**

ii. **strerror()**: returns a pointer to the textual representation of the current errno value.
Syntax: char *strerror (int errnum) //errnum is the error number (errno).

Example Code2:

This program illustrates how perror() and strerror() functions are used to display error messages.

```
#include <stdio.h>
#include <errno.h>
#include<string.h>
int main ()
{
    FILE *fp;
    fp = fopen ("file1.txt", "r");  // File with this name exist
    fputc('A',fp);   // writing to a file which is opened for read
    printf("value of errno is %d with info:%s\n",errno,strerror(errno));
    perror("Bad code");
    fclose(fp);
    return 0;
}
```

3. **Two-status library functions are used to prevent performing any operation beyond EOF**.

1. **feof():** Used to test for an end of file condition
2. **ferror():** Used to check for the error in the stream

<u>**feof():**</u>

In C, getc() returns EOF when end of file is reached. getc() also returns EOF when it fails. So, only comparing the value returned by getc() with EOF is not sufficient to check for actual end of file. To solve this problem, C provides **feof() which returns non-zero value only if end of file has reached, otherwise it returns 0.**

Syntax: feof(FILE *file_pointer);

<u>**Example Code 3:**</u>

Consider the following C program to print contents of file test.c on screen. In the program, returned value of getc() is compared with EOF first, then there is another check using feof(). By putting this check, we make sure that the program prints “End of file reached” only if end of file is reached. And if getc() returns EOF due to any other reason, then the program prints “BAD Code”

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

```
    fp = fopen ( "test.c", "r" ) ; // opening an existing file
    if ( fp == NULL )
    {
        printf("Could not open file test.c" ) ;
    }
    else
    {
        printf( "Reading the file\n" ) ;
        while ( !feof(fp) )  // returns nonzero if EOF encountered
        {
            c = fgetc(fp) ; // reading the file
            printf ("%c",c) ;
        }
        //c = getc(fp);  // doesnt show any error. so better to use feof function
        //perror("BAD Code\n");
        fclose ( fp ) ; // Closing the file
     }
    return 0;
}
```

ferror():

The ferror() function checks for any error in the stream. **It returns a value zero if no error has occurred and a non-zero value if there is an error**. The error indication will last until the file is closed unless it is cleared by the clearerr() function.

Syntax: int ferror (FILE *file_pointer);

The following two example programs(4 and 5) illustrate the errors generated while trying to open a file with incorrect permissions.

Example Code 4:

```
#include <stdio.h>
int main()
{
        FILE* fp;
        char feedback[100];
        fp = fopen("feedback.txt", "r");  // opened in r mode. File must be existing
        if (fp == NULL)
                printf("\n The file could not be opened");
        else
        {
                printf("\n Provide feedback on this article: ");
                fgets(feedback, 100, stdin);
                for (i = 0; i < feedback[i]; i++)
                        fputc(feedback[i], fp);  // writing to a file character by character
                //printf("%d",errno);
                //perror("Bad code\n");
                // writing to a file is not allowed as the mode used is r while opening
                if (ferror(fp))
                {
                        printf("\n Error writing in file");
                        printf("%d",errno);
                        perror("Bad code\n");
                }
                fclose(fp);
        }
        return 0;
}
```

Example Code 5:

```
#include <stdio.h>
int main()
{
        FILE* fp;
        char feedback[100];
        fp = fopen("feedback.txt", "r");  // opened in r mode. File must be existing
        if (fp == NULL)
                printf("\n The file could not be opened");
        else
        {
                printf("\n Provide feedback on this article: ");
                fgets(feedback, 100, stdin);
                int i;
                for (i = 0; i < feedback[i]; i++)
                  fputc(feedback[i], fp);  // writing to a file character by character
                //printf("%d",errno);
                //perror("Bad code\n");
                // writing to a file is not allowed as the mode used is r while opening
                if (ferror(fp))
                        printf("\n Error writing in file");
                clearerr(fp);   // error indication is stopped . next ferror not executed
                printf("after error thrown\n");
                if (ferror(fp)) {
                        printf("Error in reading from file : file.txt\n");
        }
        return 0;
}
```

File handling Best Practices

1. Given a file pointer check whether it is NULL before proceeding with further operations
2. Use errno.h and global variable errno to know the type of error that occurred. Usage of strerror() and perror() helps in providing textual representation of the current errno value
3. Good to check whether EOF is reached or not before performing any operation on the file

Problem Solving: File Handling

Let us consider few problem statements to solve.

1. Count the number of matches played in the year 2008 – Solution: 58
2. Count the number of times the Toss winner is same as the Winner of the match
3. Display the count of matches played between KKR and RCB
4. Display the Winner of each match played in 2016
5. Display the list of Player of the match when there was a match between RCB and CSK in the year 2010

Optional : The output of all the above can be written to a file to store the record of outputs in one file

Data file Description

File to be used is matches.csv: Contains the information of cricket matches held between 2008 and 2016 **.** First row represents the column headers such as id, season, city, date, team1, team2, toss_winner, toss_decission, result, dl_applied, winner, win_by_runs, win_by_wickets, player of match, venue, umpire1, umpire2 and umpire3**.**

Contains around 577 rows. Every data is stored with a comma in between.

Let us consider the first problem to solve - **Count the number of matches played in the year 2008**

```
// solution1.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int main()
{
    FILE *fp=fopen("matches.csv","r");
    char line[500];
    if(fp==NULL)
    {
            printf("error in opening the file\n");
    }
    else
```

```
        {
                int count=0;
                while(fgets(line,500,fp)!=NULL)
                {
                        char *val=strtok(line,",");
                        val=strtok(NULL,",");
                        if(strcmp(val,"2008")==0)
                        {
                                count++;
                        }
                }
                fclose(fp);
                printf("Number of matches in 2008 are %d\n",count);
        }
        return 0;
    }
```

Strtok: Available in string.h

This function returns a pointer to the first token found in the string. A null pointer is returned if there are no tokens left to retrieve.

Function takes two arguments – a source string or NULL and the delimiter string. The first time strtok is called, the string to be splitted is passed as the first argument. In subsequent calls, NULL is passed to indicate that strtok should keep splitting the same string for the next token.

Can you pass the FILE pointer to a user defined function? Modify the code by separating the interface and implementation

//Header file: p2.h

```
#include<stdio.h>   // To use FILE in p2.h, stdio.h must be used
int get_count(FILE* fp);
```

//Client code: p2_client.c

```
#include"p2.h"
#include<stdio.h> // To use NULL and FILE , need to ass stdio.h
int main()
{
    FILE *fp=fopen("matches.csv","r");
    char line[500];
    if(fp==NULL)
    {
            printf("error in opening the file\n");
    }
    else
    {
            int count = get_count(fp);
            printf("Number of matches in 2008 are %d\n",count);
            fclose(fp);
    }
    return 0;
}
```

//Source File: p2.c

```
#include<stdio.h>    // To use FILE in p2.c, stdio.h must be used
#include<string.h>
int get_count(FILE* fp)
{
    char line[500];
    int count=0;
    while(fgets(line,500,fp)!=NULL)
    {       char *val=strtok(line,",");
            val=strtok(NULL,",");
            if(strcmp(val,"2008")==0)
            {
                    count++;
```

```
                }
        }
        return count;
    }
```

Other questions listed in the beginning can be considered as practice questions for you to solve.

Introduction

Even a single day does not pass without we searching for something in our day to day life. Might be car keys, books, pen, mobile charger and what not. Same is the life of a computer. There is so much data stored in it and whenever user asks for some data, computer has to search it's memory to look for the data and make it available to the user. And the computer has it's own techniques to search through it's memory in a faster way.

Why Searching?

We often need to find one particular item of data amongst many hundreds, thousands, millions or more. For example, if one wants to find someone's phone number in your phone, or a particular business's address from address book.

A searching algorithm can be used to help to find the data without spending much time.

What if you want to write a program to search a given integer in an array of integers? Two popular algorithms available:

• **Linear Search**

• **Binary Search**

Linear Search

A linear search, also known as a sequential search, is a method of finding an element within a collection. It checks each element of the list sequentially until a match is found or the whole list has been searched. It returns the position of the element in the array, else it return -1. Linear Search is applied on unsorted or unordered collection of elements.

Example: Implementation of Linear Search

```
int key = 100;
int a[100] = {22,55,77,99,25,90,100};

int n = 7;

int linear_search(int *a, int n, int key)

{

        int i; int found = 0; int pos = -1;

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

        {

                if(a[i]==key && found == 0)

                {

                        found = 1;

                        pos = i;

                }

        }

        return pos;

}
```

Binary Search

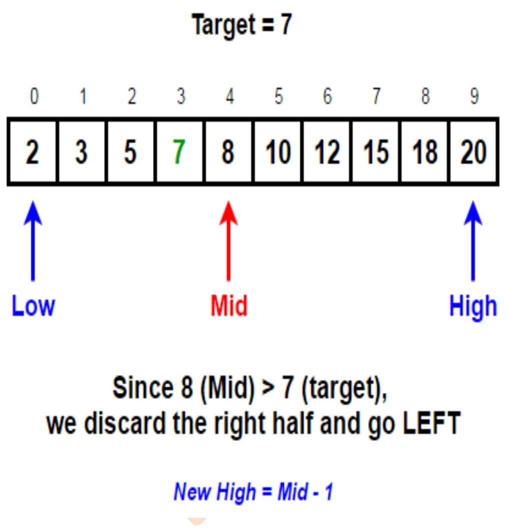
Binary Search is used to search an element in **a sorted array**. The following steps are applied to search an element.

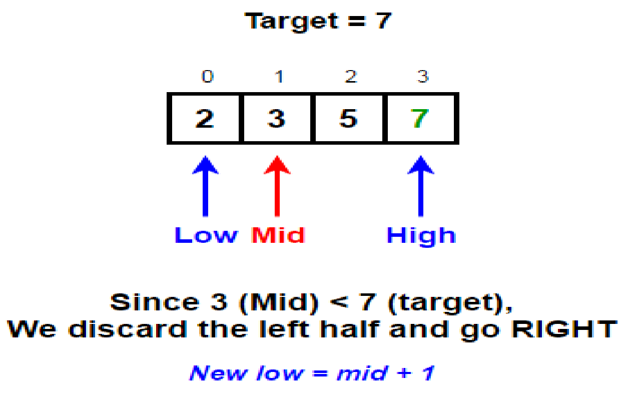
1. Start by comparing the element to be searched with the element in the middle of the array.
2. If we get a match, we return the index of the middle element.

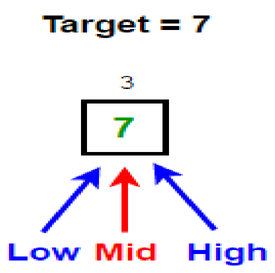
3. If we do not get a match, we check whether the element to be searched is less or reater than in value than the middle element.
4. If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element(as the array is sorted, hence on the right, we will have all the numbers greater than the middle number), and start again from the step 1.
5. If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.

Binary Search Representation

Example: Take an input array by name arr = [2, 3, 5, 7, 8, 10, 12, 15, 18, 20] and **target or key to be searched is 7.**




Target = 7
0 1 2 3
2 3 5 7
Low
Mid
High
Since 3 (Mid) < 7 (target),
We discard the left half and go RIGHT
New low = mid + 1



Now our search space consists of only one element 7. Since 7 (Mid) = 7 (target), we return index of 7 i.e. 3 and terminate our search

Note: Between 2,3,5,7 and 7, Diagram is missing for two iterations. Please write those two itereations also to understand

Binary Search Algorithm(Iterative solution)

```
Algorithm Binary_Search(list, item)
1. Set L to 0 and R to n: 1
2. if L > R, then Binary_Search terminates as unsuccessful
3. else,Set m (the position in the mid element) to the floor of (L + R) / 2
4. if $A_m$ < T, set L to m + 1 and go to step 2
5. if $A_m$ > T, set R to m: 1 and go to step 2
6. Now, $A_m$ == T, the search is done  and return (m).
```

Explanation

The iterative binary search algorithm works on the principle of "Divide and Conquer".First it divides the large array into smaller sub-arrays and then finding the solution for one sub-array.It finds the position of a key or target value within an array and compares the target value to the middle element of the array, if they are unequal, the half in which the target cannot lie is eliminated.The search continues on the remaining half until it is successful.

Binary Search Algorithm(Recursive solution)

```
BinarySearch.(x:target;{a1,a2,…,an}:sorted list of items;
1. begin:pos;end:pos
```

2. mid:=(begin+end)/2
3. If begin>end:return -1
4. else if x==amid:return mid
5. else if x<amid:return BinarySearch(x;{a1,a2,…,an};begin;mid-1)
6. else return BinarySearch(x;{a1,a2,…,an};mid+1,end);

Explanation

In recursive approach, the function BinarySerach is recursively called to search the element in the array. It accepts the input from the user and store it in **m.** The array elements are declared and initialized.In the recursive call to function **the array, the lower index, higher index and the number are passed as arguments** to be searched again and again until element is found. If not found, then it returns -1.

Example: Implementation of Binary Search (iterative & recursive)

```
// Binary search:
/*      works on sorted collection of elements.
        Time required to access each element in the collection must be same - constant time
        */
#include<stdio.h>
int mysearch(int a[],int low,int high,int key)
{
        /* binary search - iterative solution
        int pos = -1;
        int found = 0;
        // if found variable is not created, what is the problem. Think about it?
        while(low<=high && found ==0)
        {
```

```
        int mid = (low+high)/2;
        if(a[mid]==key)
                {
                pos = mid;found = 1;
                }
        else if(key<a[mid])
                high = mid-1;
        else
                low = mid+1;
}
return pos;
*/
```

// Recursive solution

```
if(low > high) // base condition
        return -1;
else
{
        int mid = (low+high)/2;
        if(a[mid]==key)
        {
                return mid;
        }
        else if(key<a[mid])
                return mysearch(a,low,mid-1,key);
```

```
          else
                    return mysearch(a,mid+1,high,key);
     }
}
// client code is as below
int main()
{
     int a[100];
     int key; int n; int i;
     FILE *fp = fopen("numbers.txt","r"); // file exists with 5 integers in it
     printf("How many numbers you want to read from the file?");
     scanf("%d",&n);
     for(i = 0;i<n;i++)
          fscanf(fp,"%d",&a[i]);
     fclose(fp);
     printf("enter the element to be searched\n");
     scanf("%d",&key);
     int res = mysearch(a,0,5,key);
     if(res == -1)
          printf("not found");
     else
          printf("found at %d\n",res);
}
```

Introduction

There are times when an array of pointers to structs is useful. For example, suppose the structure had 100 elements, that one of the elements was a person's surname and one wanted to sort the structure in alphabetical order of surnames. The sorting technique involves a lot of swapping and moving structure around. This is slower than moving pointers around because pointers occupy less space. So to speed up the sorting process and efficiency factor, array of pointers is used for the structure rather than the whole structure or record being accessed.

Let's understand the below sections in detail:

1. Array of Pointers
2. Pointers to Structure
3. Array of Pointers to Structures

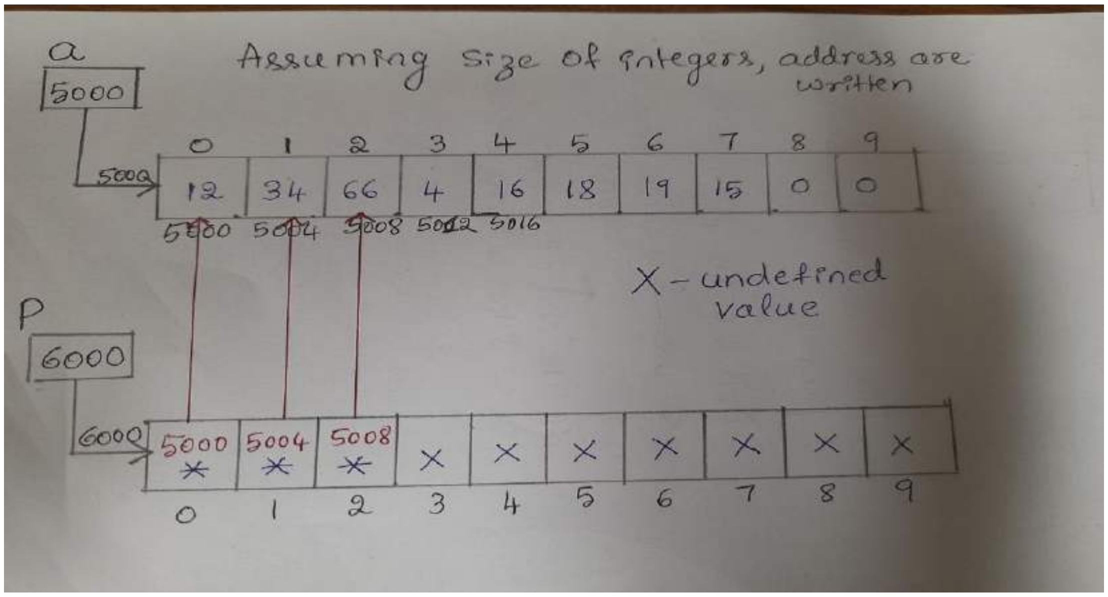
Array of Pointers

An array of pointers is an indexed set of variables in which the variables are pointers. Array and pointers are closely related to each other. The name of an array is considered as a pointer, i.e., the name of an array contains the address of an element.

Consider the below code:

```
int i;
int a[10] = {12,34,66,4,16,18,19,15};
int *p[10]; // p is created with the intention of storing an array of addresses
p[0] = &a[0];
p[1] = &a[1];
p[2] = &a[2];

for(i = 0 ; i<4;i++)
{
		printf("%p %p\n",p[i],&a[i]);
	// same address for all 3 pairs except the last iteration as we have not assigned.
}
// printing arry elements using p
for(i = 0 ; i<4;i++)
{
		printf("%d\t",*p[i]); // dereference every element of the array of pointer
}
```

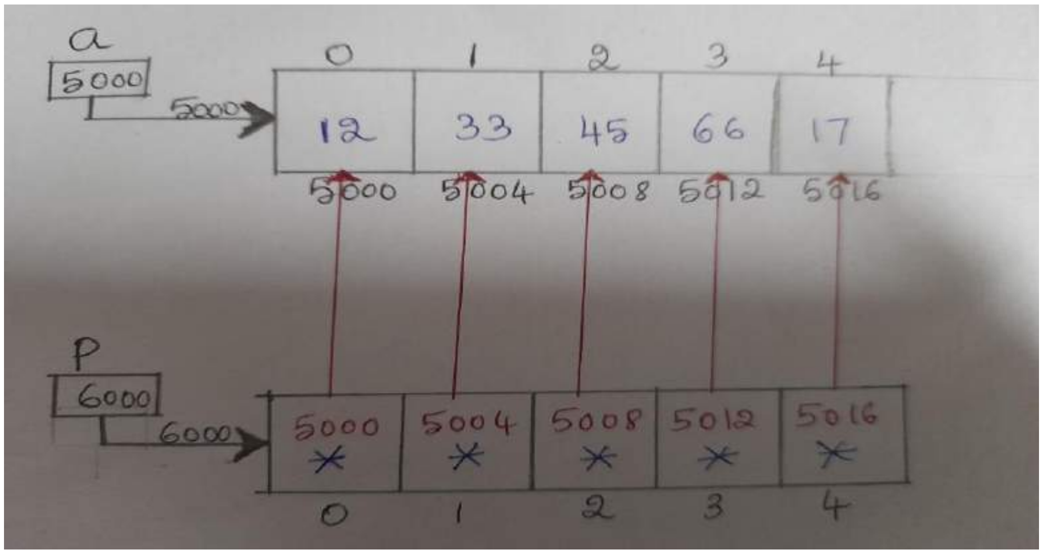


Program 1:

Program to print the elements of the array using array of pointers

```
#include<stdio.h>
int main()
{
```

```
int a[5] = {12,33,45,66,17}; // a is an array
printf("Using original array\n");
for(int i = 0;i<5;i++)
{
        printf("%d ",a[i]);
}
printf("\n");
int *p[5]; // p is an array of pointers.
for(int i = 0;i<5;i++)
{
        p[i] = &a[i]; // address of a[i] stored in p[i]
}
```



```
printf("Using array of pointers\n");
for(int i = 0;i<5;i++)
{
        printf("%d ",*p[i]);
        // content at p[i] is displayed. All elements are displayed
```

```
                }
                printf("\n");
                return 0;
}
```

Pointers to Structures

A pointer is a variable which points to the address of another variable of any data type like int, char, float etc. Similarly, a pointer to structure is a variable that can point to the address of a structure variable .Declaring a pointer to structure is same as pointer to variable:

Syntax: struct tagname *ptr;

There are two ways of accessing members of structure using pointer:

1. Using indirection (*) operator and dot (.) operator.
2. Using arrow (->) operator or membership operator.

Array of Pointers to Structures

An array of pointers to structure is also a variable that contains the address of structure variables. In order to create array of pointers for structures three declarations are defined. The first step in array of pointers for Structure is creation of an array of structure variable

Syntax: struct tag name array-variable[size];

The final step is to create an array of pointers with size specified to hold the addresses of structures in the array of structure variable.

Syntax: struct tagname *pointer_variable[size];

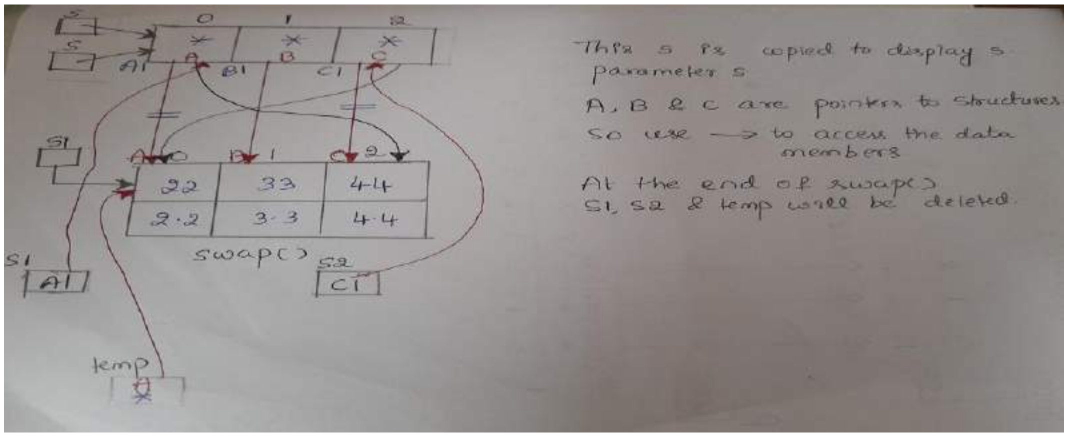
Program 2:

Program to swap first and last elements of the array of structures using array of pointers and display the array of structures using array of pointers.

```
struct Sample
{
            int a;
            float b;
};
void display(struct Sample* s[], int n);
void swap(struct Sample** s1, struct Sample** s2);
int main()
{
                        struct Sample s1[] = {{22, 2.2}, {33,3.3},{44,4.4}};
                        struct Sample* s[3];
                        for(int i = 0;i<3;i++)
                                    s[i] = &s1[i];
                        printf("Before swap--->\n");
                        display(s,3);
                        swap(&s[0],&s[2]);
                        printf("\nAfter swap--->\n");
                        display(s,3);
                        return 0;
}

void swap(struct Sample** s1, struct Sample** s2)
{
                        struct Sample *temp;
                        temp = *s1;
                        *s1 = *s2;
                        *s2 = temp;
}
```

```
void display(struct Sample* s[], int n)
{
                    for(int i = 0;i<n;i++)
                    {
                              printf("%d %f\n",s[i]->a, s[i]->b);
                    }
}
```



This s is copied to display s
parameter s

A, B & C are pointers to structures
So use -> to access the data
members

At the end of swap()
S1, S2 & temp will be deleted.

Introduction

There are so many things in our real life that we need to search for, like a particular record in the database, roll numbers in the merit list, a particular telephone number in a telephone directory, a particular page in a book etc. The concept of sorting came into existence, making it easier for everyone to arrange data in order to make the searching process easy and efficient. **Arranging the data in ascending or descending order is known as sorting.**

Why Sorting?

Think about searching for something in a sorted drawer and unsorted drawer.If the large data set is sorted based on any of the fields, then it becomes easy to search for a particular data in that set.Sometimes in real world applications, it is necessary to arrange the data in a sorted order to make the searching process easy and efficient.

Example: Candidates selection process for an Interview- Arranging candidates for the interview involves sorting process(sorting based on qualification ,Experience, skills or age etc.

Sorting Algorithms

Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order. Depending on the data set, Sorting algorithms exhibit different time and space complexity.

Following are samples of sorting algorithms.

1. Bubble Sort
2. Insertion Sort
3. Quick Sort
4. Merge Sort
5. Radix Sort

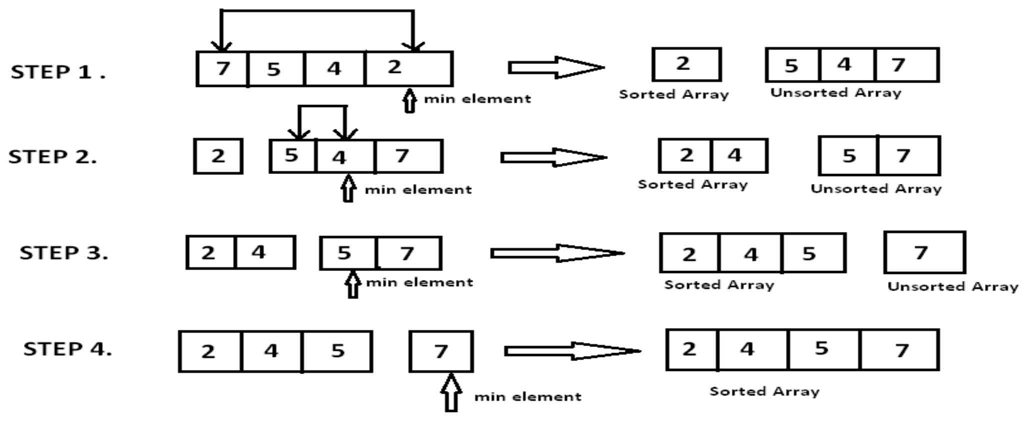
6. Selection Sort -- Dealt in detail

7. Heap Sort

Selection sort

- The method of sorting is faster as the movement of data is minimized.
- Used to arrange data in ascending order, the smallest number is selected and placed at the beginning of the collection
- When the next pass takes place the second minimum number is selected and placed in the second postion
- The process is repeated until all the elements are placed in the correct order.
- With each pass the elements are reduced by 1 as one element is sorted and placed in a position.

Pictorial Representation



}

Selection Sort Algorithm

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.
2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

C Program to Illustrate the Selection Sort

//client.c

```
#include<stdio.h>
#include"sort.h"
int main()
{               int a[100];
                int n;
                printf("enter the number of elements u want to sort\n");
                scanf("%d",&n);
                printf("enter %d elements\n",n);
                read(a,n);
                printf("before sorting\n");
                disp(a,n);
                sort(a,n);
                printf("after sorting\n");
                disp(a,n);
                return 0;
}
```

//sort.h

```
    void sort(int a[],int n);
    void swap(int *x,int *y);
    void disp(int a[],int n);
    void read(int a[],int n);
```

//sort.c

```
    include<stdio.h>
    void disp(int a[],int n)
    {
        for(int i = 0;i<n;i++)
        {
            printf("%d ",a[i]);
        }
        printf("\n");
    }
    void read(int a[],int n)
    {
        for(int i = 0;i<n;i++)
        {
            scanf("%d",&a[i]);
        }
    }
```

```
void swap(int *x,int *y)
{
        int temp = *x;
        *x = *y;
        *y = temp;
}
void sort(int a[],int n)
{
        int i,pos,j;
        for(i = 0;i<n-1;i++)
        {
                pos = i;
                for(j = i+1;j<n;j++)
                {
                        if(a[pos]>a[j])
                                pos = j;
                }
                if(pos != i)    // think about this
                        swap(&a[pos],&a[i]);
        }
}
```

Problem Solving: Sorting using Array of Pointers

Given a data file student.csv, create a menu driven program to perform selection sort based on roll_number and name. Write the sorted record to a new file.

Data file Description:

- Contains the information of students in two columns only
- First row represents the column headers such as roll_no and name
- Contains around 55 rows. Every data is stored with a comma in between

Let us separate the Interface and Implementation

//sort.h is as below

```
struct student
{
    int roll;
    char name[20];
};
void init_ptr(struct student s[], struct student *p[], int n);
void swap( struct student** lhs,  struct student** rhs);
void disp(struct student* p[], int n) ;
void selection_sort_roll_no(struct student* s[],int n);
void selection_sort_names(struct student* s[],int n);
```

//client.c is as below

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include"sort.h"
int main()
{
    FILE *fr=fopen("student.csv","r");
    char a[200];
    fgets(a,200,fr);
    char *item;
    struct student s[100];
    int i=0;
    // Reading a line from the file and splitting it based on the delimiter comma(,),
copying the details to array of student structure
    while(fgets(a,200,fr))
    {
            item=strtok(a,",");
            s[i].roll=atoi(item);
            item=strtok(NULL,",");
            strcpy(s[i].name,item);
            i++;
    }
    int n = i;
    fclose(fr);
    //Creation of array of pointers to point to array of structures
    struct student *p[100];
    init_ptr(s, p, n);    // every element is array of pointer is made to point to every
element of array of struture
    disp(p,n);      // to display the array of structures using array of pointers

    // Menu driven code
    int ch;
    printf("enter the choice.\n 1.sort on roll\n");
    printf("2. sort on name");
    scanf("%d",&ch);
    switch(ch)
    {
            case 1:selection_sort_roll_no(p,n);  // array of pointer and n is the argument
                        disp(p,n);
                        break;
            case 2:selection_sort_names(p,n); disp(p,n);break; // same as above
```

```
                    default: printf("exiting from the program");
                                    exit(0);

            }

          // Writing the sorted record to a new file
           FILE *fw=fopen("student_sorted.csv","w");
           fprintf(fw,"Roll_number,Name\n");
           i=0;
           while(i<n)
           {
                    fprintf(fw,"%d,%s",p[i]->roll,p[i]->name);
                    i++;
           }
           fclose(fw);
           printf("sorted data is written to a file student_sorted.csv\n");

           return 0;
}
```

Let us create the source file to have the implementation of all the functions used above.

//sort.c

```
        #include"sort.h"
        #include<stdio.h>
        #include<string.h>
        // Initializing the pointer
        void init_ptr(struct student s[], struct student *p[], int n)
        {
                for(int i = 0; i < n; ++i)
                {
                        p[i] = &s[i];
                }

        }
```

```
//Swap the pointers in the array
void swap( struct student** lhs,  struct student** rhs)
{
    struct student* temp = *lhs;
    *lhs = *rhs;
    *rhs = temp;
}

void disp(struct student* p[], int n)
{
    for(int i = 0; i < n; ++i)
    {
            //printf("%s %d\n", (s+i)->name, (s+i)->roll);
            printf("%d %s\n", p[i]->roll, p[i]->name);
    }
}
void selection_sort_roll_no(struct student *s[],int n)
{
    int i,pos,j;
    for(i = 0;i<n-1;i++)
    {
            printf("in for\n");
            pos = i;
            for(j = i+1;j<n;j++)
            {
                    if((s[pos]->roll) > (s[j]->roll))
                    // compare the roll numbers using pointers to sort the record
                            pos = j;
            }
            if(pos != i)
                    swap(&s[pos],&s[i]);      // Swap the pointers not the structures
    }
}
```

```
        void selection_sort_names(struct student* s[],int n)
        {
            int i,pos,j;
            for(i = 0;i<n-1;i++)
            {
                    pos = i;
                    for(j = i+1;j<n;j++)
                    {
                            if(strcmp(s[pos]->name,s[j]->name)>0)
                            // compare the names using pointers to sort the record
                                    pos = j;
                    }
                    if(pos != i)
                            swap(&s[pos],&s[i]);
            }
}
```

Few points to think about the above code:

- Could you modify the code to sort in descending order?
- What happens when you sort on names when two students have the same name?
- Rather than having two functions to sort on roll_no and sort on names, can we use one function which does sorting based on some condition?
- Can we pass the function name as an argument to a function and call appropriate function inside the sort function? This is done using the concept of callback. We will be discussing this shortly.

Introduction

In computer programming, a **callback** is also known as a "**call-after function"**. The callback execution may be immediate or it might happen at a later point in time. Programming languages support callbacks in different ways, often implementing them with subroutines, lambda expressions,blocks, or function pointers. In C, a **callback function is a function that is called through a function pointer/pointer to a function.** A callback function has a specific action which is bound to a specific circumstance. It is an important element of GUI in C programming.

Let us understand the use of **pointer to a function** or a **function pointer**.

Function Pointer points to code, not data. The function pointer stores the start of executable code. The function pointers points to functions and store its address.

Syntax:int (*ptrFunc) ();

The **ptrFunc** is a pointer to a function that takes no arguments and returns an integer If parentheses are not given around a function pointer then the compiler will assume that ptrFunc is a normal function name, which takes nothing and returns a pointer to an integer. Function pointers are not used for allocating or deallocating memory, instead used in reducing code redundancy (sorting algorithms).

let us consider few examples below on pointer to function.

Case 1: int *a1(int, int, int); // a1 is a function which takes three int arguments and returns a pointer to int.

Case 2: int (*p)(int, int, int); p is a pointer to a function which takes three int as parameters and returns an int.

We can assign a function name to p as long as the function has the same signature which means the function takes three integer parameters and returns an integer. The function name acts like a pointer to a function.

```
int a2(int,int,int);
p = a2;
int y = p(2,3,4); // This statement is same as calling a2.
```

Why callback function required?

Let us answer few questions to understand the importance of callaback function.

- ✓ How to write a generic library for sorting algorithms such as bubble sort, shell short, shake sort, quick sort?
- ✓ How to create a notification event to set a timer in your application?
- ✓ How to have one common method to develop libraries and event handlers for many programming languages?
- ✓ How redirect page action is performed for the user based on one click action?
- ✓ How to extend the features of one function using another one?

A callback is any executable code that is passed as an argument to other code, which is used to call (execute) the argument at a given time. In simple language, if a function name is passed to another function as an argument to call it, then it will be called as a Callback function.

Program 1: Consider the below simple example to understand callback

```
int what(int x, int y, int z, int (*op)(int, int,int))
{
        return op(x, y, z);
}
```

Observe the third argument op. It is a pointer to a function which takes three int arguments and returns an int. The return statement in turn calls the function with three arguments. This function does not know which function is getting called. It depends on the value of op. Compiler knows the type and not the value. The value of a variable is a runtime mechanism.

```
int add(int x, int y, int z)
{
	return x + y+z;
}
int mul(int x, int y, int z)
{
	return x * y * z;
}
int main()
{
	int (*p)(int, int, int);
	p = add; // p = &add; Both are equivalent
	int res = p(2,3,4); // (*p)(2, 3, 4); Both are equivalent
	printf("result is %d\n", res);
	p = mul;
	res = p(2,3,4);
	printf("result is %d\n", res);
	// function name can be directly passed as argument to function with callback
	printf("sum is %d\n", what(1, 3, 4, add));
	printf("product is %d\n", what(5, 3, 4, mul));
}
```

Program 2: If you are aware of Python’s Functional programming constructs such as map, reduce and filter, we will be implementing those here in C using callback functions.

// Mimic map, filter and reduce function of python

//client_callback.c contains the below code

```
#include<stdio.h>
#include "fun.h"
int incr(int x)
{
        return x+1;
}
int is_even(int x)
{
        return x%2 == 0;
}
int add(int x,int y)
{
        return x+y;
}
int main()
{
        int a[] = {11,22,33,44,55};
        int n = sizeof(a)/sizeof(*a);
        int b[n];
        printf("a is --------\n");
        disp(a,n);
        mymap(a,b,n,incr);
        printf("\nb is ------- \n");
        disp(b,n);
```

```
	/*int c = myfilter(a,b,n,is_even); // write appropriate implementation in fun.c for
myfilter function returning integer
	printf("\nb is ------- \n");
	disp(b,c);*/

	int m;
	myfilter(a,b,n,&m,is_even); // this myfilter function doesn'treturn any value
	printf("\nb is ------- \n");
	disp(b,m);
	myfilter(a,b,n,&m,is_greater_than_22);
	printf("\nb is ------- \n");
	disp(b,m);
	int result = myreduce(a,n,add);
	printf("Result is %d\n",result);
	return 0;
}
```

// fun.c containsthe below code

```
#include<stdio.h>
void mymap(int a[],int b[],int n,int (*p)(int))
{
	//int n = (sizeof(a)/sizeof(*a));// array degenerates to a pointer at runtime. Think about size
of pointer
	//printf("n is %d\n",n);
	for(int i = 0;i<n;i++)
	{		b[i] = (*p)(a[i]);	}
}

void disp(const int a[],int n)
{
        for(int i = 0;i<n;i++)
        {
                printf("%d ",a[i]);
        }
}
void myfilter(const int a[],int b[],int n,int *m, int (*op)(int))
{
        int count = 0;
        for(int i = 0;i<n;i++)
        {
                if(op(a[i]))
                {
                        b[count] = a[i];
                        count++;
                }
        }
        *m = count;
}

int is_greater_than_22(int x)
{       return x > 22;          }

int myreduce(int a[],int n,int (*op)(int,int))
{
        int res = 0;
        for(int i = 0;i<n;i++)
        //for(int i = 0;i<n-1;i++)
        {
                //res = op(a[i],a[i+1])  // logical error
                //res = op(res,a[i+1])   // logical error
                res = op(res,a[i]);
        }
        return res;
}
```

//fun.h contains below function declarations

```
        void mymap(int a[],int b[],int n,int (*p)(int));
        void disp(const int a[],int n);
        void myfilter(const int a[],int b[],int n,int *m,int (*)(int));
        int is_greater_than_22(int x);
        int myreduce(int a[],int n,int (*)(int,int));
```

Problem Solving: Callback

Let us continue the sorting program using call back mechanism. Rather than calling two different functions based on user's choice, call the same function with two different arguments.

// Minor modifications to **client.c** is as below

```
int ch;
printf("enter the choice.\n 1.sort on roll\n");
printf("2. sort on name\n");
scanf("%d",&ch);
switch(ch)
{
    //observe the bold statements. Last argument is different
    case 1: selection_sort(p,n,compare_roll_number); disp(p,n); break;
    case 2: selection_sort(p,n,compare_names); disp(p,n); break;
    default: printf("exiting from the program"); exit(0);
}
```

//sort.h is as below.

```
int compare_names(student_t*,student_t*);
int compare_roll_number(student_t*,student_t*);
void selection_sort(student_t *s[],int n,int (*comp)(student_t*,student_t*))
```

//sort.c:

Contains additional function definitions of compare_roll_number and compare_names. Observe the modification to selection_sort definition using call back function mechanism.

```
int compare_roll_number(student_t* s1,student_t* s2)
{
        return s1->roll > s2->roll;
}
```

```
int compare_names(student_t *s1,student_t *s2)
{
        return strcmp(s1->name,s2->name)>0;
}

void selection_sort(student_t *s[],int n,int (*comp)(student_t*,student_t*))
    {
            int i,pos,j;
            for(i = 0;i<n-1;i++)
            {
                printf("in for\n");
                pos = i;
                for(j = i+1;j<n;j++)
                {
                        if(comp(s[pos],s[j])) // callback function is called
                        {
                                pos = j;
                        }
                }
                if(pos != i)
                        swap(&s[pos],&s[i]);
            }
}
```

Now, let us consider the implementation of Binary Search using call back when there is a constraint to check for before searching for the element.

A. Search for a number if the number is even

B. Search for a number if the number is less than 22.

Solution Link is here:

https://drive.google.com/open?id=1qkMI15vC7hFWKqKGVzjsdiusox-CeQsV