UNIT 4
SORTING AND FILE HANDLING

FILE HANDLING

A computer file is a computer resource for recording data discretely in a computer storage device.

C provides a range of functions in the header file stdio.h for writing to and reading from external devices. The external device you would use for storing and retrieving data is typically a disk drive, but not exclusively. Because, consistent with the philosophy of C, the library facilities you'll use for working with files are device independent, they apply to virtually any external storage device

A file is essentially a serial sequence of bytes stored on a medium.

Operations on Files:

Read the contents of the file or Write to a file or both. To perform these operations, we connect the physical filename, the logical filename and the mode by using a function and use other functions in C to perform operations on Files.

File Handling functions in C

Return values of below functions talk about the success story of any of these functions. Use return values for robust programming. **EOF character is used to mark the end of file**.

Fopen("file","mode") :

Opens a file in the specified mode and returns a FILE pointer(address of the structure which contains information about the attributes of the file) if it succeeds and else returns NULL. The file opening could fail for a number of reasons: File might not be available or File might not have permission to open in the mode specified. Initializing a FILE pointer does not mean that the whole file is made available in memory. We need to use other functions to access the contents.

Modes available:

File can be opened in different modes.

"r" Opens a file for reading. The file must exist.

"w" Creates an empty file for writing.

If a file with the same name already exists, its content is erased and the file is considered as a new empty file.

"a" Appends to a file. Append data always at the end of the file.

The file is created if it does not exist.

"r+" Opens a file for both reading and writing. The file must exist.

"w+" Creates an empty file for both reading and writing.

"a+" Opens a file for appending and reading. Append data always at the end of the file. The file is created if it does not exist.

fclose(file_pointer)

Opening of a file causes creation of data structures in the memory. We should remove all these when the file is no more required. We do that by closing the file. Closes the stream. All buffers are flushed. Returns zero if the stream is successfully closed. On failure, EOF is returned.

fgetc(file_pointer)

Reads a character from the file and returns the character read as an unsigned char cast to an int(ascii) and **advances the file pointer to the next byte to be read** or EOF on end of file or error.

. **fputc(int ch,File_pointer)**

. Writes a character (an unsigned char) specified by the argument ch to the specified stream and advances the position indicator for the stream. If there are no errors, the same character that has been written is returned. If an error occurs, EOF is returned and the error indicator is set.

. **fgets(char_array_variable, bytes, file_pointer**)

Reads a line from the specified stream and stores it into the string pointed to by char_array_variable. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first. On success, the function returns the same char_array parameter. On error or if EOF is encountered, it returns NULL.

. **fputs(char_array_variable,file_pointer)**

Writes a string to the specified stream up to but not including the null character. Returns a non-negative value, or else on error it returns EOF.

. **ftell(File_pointer)**

Returns the current file position of the given file_pointer. If an error occurs, -1 is returned.

. **fseek(File_pointer, offset_bytes, From_where**)

. Used to provide random access to the file. Sets the file position of the stream to the given offset. Returns zero if successful, else it returns a non-zero value. From_where can be one of the following values

SEEK_SET or 0 SEEK_CUR or 1 SEEK_END or 2

Beginning of file Current position of the file pointer End of file

- **rewind(File_pointer)** sets the file_pointer position to the beginning of the file. Doesn't return any value.

- **feof(File_pointer)** Tests the end-of-file indicator for the given stream. Returns a non-zero value when End-of- File indicator associated with the stream is set, else zero is returned. Formatted i/o functions available in C for file handling are as follows:

- **fscanf(file_pointer,"format_string",&arg1,&arg2,...)** It reads formatted input from a stream and returns the number of input items successfully matched and assigned, which can be fewer than provided for, or even zero in the event of an early matching failure.

- **fprintf(file_pointer,"format_string", arg1,arg2,...)** Sends formatted output to a stream. If successful, the total number of characters written is returned otherwise, a negative number is returned.

Program to copy one file to another

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

void copy_file (const char *destfile, const char *srcefile)
{
        FILE *fsrce, *fdest;
      char buf [120];
        int in_char;

        if ((fsrce = fopen (srcefile, "r")) == NULL)
        {
                printf ("Error in opening the source file\n");
                exit (2);
        }
        if ((fdest = fopen (destfile, "w")) == NULL)
        {
                printf ("Error in opening the destination file\n");
                exit (2);
        }

        while (1)
        {
                if (fgets (buf, 120, fsrce) == NULL) // Nothing more to read
                        break;
                if (fputs (buf, fdest) == EOF)
                {
                        printf ("Error in writing to the destination file\n");
                        exit (3);
                }
        }
        fclose (fsrce);
        fclose (fdest);
}

int main (void)
{
        char dest_file [50];
        char srce_file [50];
```

```
        printf ("Key in the name of the source file:");
        scanf ("%s", srce_file);
        printf ("Key in the name of the destination file:");
        scanf ("%s", dest_file);
        copy_file (dest_file, srce_file);
        return 0;
}
```

Program to find the size of a given file

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

int FileSize (char *filename)
{
        FILE *fp;
        int count = 0, in_char;

        if ((fp = fopen (filename, "r")) == NULL)
        {
                printf ("Error in opening the file\n");
                exit (2);
        }

        while (1)
        {
                in_char = fgetc (fp);
                if (in_char == EOF)
                        break;
                count ++;
        }
        fclose (fp);
        return count;
}

int main (void)
{
        char filename [50];

        printf ("Name of the file:");
        scanf ("%s", filename);

        printf ("size of the file is %d bytes\n", FileSize (filename));
        return 0;
}
```

Illustrate fseek and ftell

fseek is used to have random access to the file contents

```
FILE *fp1 = fopen("abc.txt","r");

printf("fp1 %d\n",ftell(fp1));

char ch = fgetc(fp1);

printf("fp1 %d\n",ftell(fp1));

FILE *fp1 = fopen("abc.txt","r");

char ch[200];

printf("fp1 %d\n",ftell(fp1));

fgets(ch,200,fp1);

 printf("%s",ch);

printf("fp1 %d\n",ftell(fp1));

FILE *fp1 = fopen("abc.txt","r");

char ch[200];

printf("fp1 %d\n",ftell(fp1));

fseek(fp1,14, SEEK_SET);

printf("fp1 %d\n",ftell(fp1));

fclose(fp1);

printf("fp1 %d\n",ftell(fp1));
```

SORTING

Depending the data set, Sorting algorithms exhibits different time and space complexity. Following are samples of sorting algorithms.

Selection Sort

Bubble Sort

Selection Sort on array of integers

. Start from the first element in the array and search for the smallest element in the array.

. Swap the first element with the smallest element of the array.

. Take a subarray (excluding the first element of the array as it is at its place) and search for the smallest number in the subarray (second smallest number of the entire array) and swap it with the first element of the subarray (second element of the entire array).

. Repeat the steps 2 and 3 with new subarrays until the array gets sorted.

```
#include <stdio.h>

void swap(int *xp, int *yp)
{
   int temp = *xp;
   *xp = *yp;
   *yp = temp;
}

void selectionSort(int arr[], int n)
{
   int i, j, min_idx;

   // One by one move boundary of unsorted subarray
   for (i = 0; i < n-1; i++)
   {
      // Find the minimum element in unsorted array
      min_idx = i;
      for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
          min_idx = j;
      if (min_idx != i)  // If it is, the elements are in the right order
```

```
            // Swap the found minimum element with the first element
            swap(&arr[min_idx], &arr[i]);
    }
}
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
 int main(void)
{
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

```
bubbleSort(array)

  for i <- 1 to indexOfLastUnsortedElement-1

    if leftElement > rightElement

      swap leftElement and rightElement

end bubbleSort
```

```
#include <stdio.h>

void bubbleSort(int array[], int size) {

for (int step = 0; step < size - 1; ++step) {
    for (int i = 0; i < size - step - 1; ++i) {


      if (array[i] > array[i + 1]) {
```

```
int temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;
      }
    }
  }
}

void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
  }
  printf("\n");
}

int main()
{
  int data[] = {-2, 45, 0, 11, -9};
  int size = sizeof(data) / sizeof(data[0]);
  bubbleSort(data, size);
  printf("Sorted Array in Ascending Order:\n");
  printArray(data, size);
}
```

Searching

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.

Two popular algorithms available:

. Linear Search

. Binary Search

. **Linear Search**

In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or the element is found. It compares the element to be searched with the elements present in the array and when the element is **matched** successfully, 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.

```
        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)

        {

}

return pos;

}
```

Binary Search

Binary Search works on sorted collection of elements. In binary search, we follow the following steps:

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 greater 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.

A necessary condition for Binary search to work is that the **array should be sorted.**

Let us write the code for binary search using iterative method and using recursive functions. We will read the collection of elements from the file and apply binary search to check whether the key element is available in the file

20 30 40 45 55 89 101 : Numbers available in the text file.

```
 int main()

{

int a[100];

  int key;

 int i;

int n = 0;

FILE *fp = fopen("file","r");

 while(fscanf(fp,"%d",&a[n]) != EOF))

n++;

fclose(fp);

printf("enter the element to be searched\n");

scanf("%d",&key);

int res = mysearch(a,0,n,key);

if(res == -1)
```

```
printf("not found");
 else
printf("found at %d\n",res);
}
```

Array of pointers

An array of pointers is an indexed set of variables in which the variables are pointers.

```
int i;
int a[10] = {12,34,66,4,16,18,19,15};
int *p[10];
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]);
}
```