UNIT V

FILE PROCESSING

Files – Types of file processing: Sequential access, Random access - Sequential access file - Example Program: Finding average of numbers stored in sequential access file - Random access file - Example Program: Transaction processing using random access files – Command line arguments

FILES

C File management

A File can be used to store a large volume of persistent data. Like many other languages 'C' provides following file management functions,

1. Creation of a file
2. Opening a file
3. Reading a file
4. Writing to a file
5. Closing a file

Following are the most important file management functions available in 'C,'

function	purpose
fopen ()	Creating a file or opening an existing file
fclose ()	Closing a file
fprintf ()	Writing a block of data to a file
fscanf ()	Reading a block data from a file
getc ()	Reads a single character from a file

putc ()	Writes a single character to a file
getw ()	Reads an integer from a file
putw ()	Writing an integer to a file
fseek ()	Sets the position of a file pointer to a specified location
ftell ()	Returns the current position of a file pointer
rewind ()	Sets the file pointer at the beginning of a file

Why files are needed?

- 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 all.
  However, if you have a file containing all the data, you can easily access the contents of the file using a few commands in C.
- You can easily move your data from one computer to another without any changes.

Types of Files

When dealing with files, there are two types of files you should know about:

1. Text files
2. Binary files

1. Text files

Text files are the normal **.txt** files. You can easily create text files using any simple text editors such as Notepad.
When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.

They take minimum effort to maintain, are easily readable, and provide the least security and takes bigger storage space.

2. Binary files

Binary files are mostly the **.bin** files in your computer.
Instead of storing data in plain text, they store it in the binary form (0's and 1's).

They can hold a higher amount of data, are not readable easily, and provides better security than text files.

File Operations

In C, you can perform four major operations on files, either text or binary:

1. Creating a new file
2. Opening an existing file
3. Closing a file
4. Reading from and writing information to a file

Working with files

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and the program.

```
FILE *fptr;
```

Opening a file - for creation and edit

Opening a file is performed using the `fopen()` function defined in the `stdio.h` header file.

The syntax for opening a file in standard I/O is:

```
ptr = fopen("fileopen","mode");
```

For example,

```
fopen("E:\\cprogram\\newprogram.txt","w");

fopen("E:\\cprogram\\oldprogram.bin","rb");
```

- Let's suppose the file `newprogram.txt` doesn't exist in the location `E:\cprogram`. The first function creates a new file named `newprogram.txt` and opens it for writing as per the mode **'w'**.
  The writing mode allows you to create and edit (overwrite) the contents of the file.
- Now let's suppose the second binary file `oldprogram.bin` exists in the location `E:\cprogram`. The second function opens the existing file for reading in binary mode **'rb'**.
  The reading mode only allows you to read the file, you cannot write into the file.

Opening Modes in Standard I/O

Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, `fopen()` returns NULL.
rb	Open for reading in binary	If the file does not exist, `fopen()`

Opening Modes in Standard I/O

Mode	Meaning of Mode	During Inexistence of file
	mode.	returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
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	Open for append. Data is added to the end of the file.	If the file does not exist, it will be created.
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+	Open for both reading and	If the file does not exist, fopen()

Opening Modes in Standard I/O

Mode	Meaning of Mode	During Inexistence of file
	writing.	returns NULL.
rb+	Open for both reading and writing in binary mode.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
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+	Open for both reading and appending.	If the file does not exist, it will be created.
ab+	Open for both reading and appending in binary mode.	If the file does not exist, it will be created.

Closing a File

The file (both text and binary) should be closed after reading/writing.

Closing a file is performed using the fclose() function.

```
fclose(fptr);
```

Here, `fptr` is a file pointer associated with the file to be closed.

Reading and writing to a text file

For reading and writing to a text file, we use the functions `fprintf()` and `fscanf()`. They are just the file versions of `printf()` and `scanf()`. The only difference is that `fprint()` and `fscanf()` expects a pointer to the structure FILE.

Example 1: Write to a text file

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

int main()
{
  int num;
  FILE *fptr;

  // use appropriate location if you are using MacOS or Linux
  fptr = fopen("C:\\program.txt","w");

  if(fptr == NULL)
  {
    printf("Error!");
    exit(1);
  }

  printf("Enter num: ");
  scanf("%d",&num);

  fprintf(fptr,"%d",num);
  fclose(fptr);

  return 0;
}
```

This program takes a number from the user and stores in the file `program.txt`.

After you compile and run this program, you can see a text file program.txt created in C drive of your computer. When you open the file, you can see the integer you entered.

Example 2: Read from a text file

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

int main()
{
  int num;
  FILE *fptr;

  if ((fptr = fopen("C:\\program.txt","r")) == NULL){
      printf("Error! opening file");

      // Program exits if the file pointer returns NULL.
      exit(1);
  }

  fscanf(fptr,"%d", &num);

  printf("Value of n=%d", num);
  fclose(fptr);

  return 0;
}
```

This program reads the integer present in the program.txt file and prints it onto the screen.If you successfully created the file from **Example 1**, running this program will get you the integer you entered.
Other functions like fgetchar(), fputc() etc. can be used in a similar way.

Reading and writing to a binary file

Functions fread() and fwrite() are used for reading from and writing to a file on the disk respectively in case of binary files.

Writing to a binary file

To write into a binary file, you need to use the `fwrite()` function. The functions take four arguments:

1. address of data to be written in the disk
2. size of data to be written in the disk
3. number of such type of data
4. pointer to the file where you want to write.

```
fwrite(addressData, sizeData, numbersData, pointerToFile);
```

Example 3: Write to a binary file using fwrite()

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

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

int main()
{
  int n;
  struct threeNum num;
  FILE *fptr;

  if ((fptr = fopen("C:\\program.bin","wb")) == NULL){
      printf("Error! opening file");

      // Program exits if the file pointer returns NULL.
      exit(1);
  }

  for(n = 1; n < 5; ++n)
  {
    num.n1 = n;
    num.n2 = 5*n;
    num.n3 = 5*n + 1;
    fwrite(&num, sizeof(struct threeNum), 1, fptr);
```

```
    }
    fclose(fptr);

    return 0;
}
```

In this program, we create a new file `program.bin` in the C drive.
We declare a structure `threeNum` with three numbers - `n1, n2 and n3`, and define it in the main function as num.
Now, inside the for loop, we store the value into the file using `fwrite()`.
The first parameter takes the address of `num` and the second parameter takes the size of the structure `threeNum`.
Since we're only inserting one instance of `num`, the third parameter is `1`. And, the last parameter `*fptr` points to the file we're storing the data.
Finally, we close the file.

Reading from a binary file

Function `fread()` also take 4 arguments similar to the `fwrite()` function as above.

```
fread(addressData, sizeData, numbersData, pointerToFile);
```

Example 4: Read from a binary file using fread()

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

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

int main()
{
    int n;
    struct threeNum num;
    FILE *fptr;

    if ((fptr = fopen("C:\\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("n1: %d\tn2: %d\tn3: %d", num.n1, num.n2, num.n3);
    }
    fclose(fptr);

    return 0;
}
```

In this program, you read the same file `program.bin` and loop through the records one by one.

In simple terms, you read one `threeNum` record of `threeNum` size from the file pointed by `*fptr` into the structure `num`.

You'll get the same records you inserted in **Example 3**.

Getting data using fseek()

If you have many records inside a file and need to access a record at a specific position, you need to loop through all the records before it to get the record.

This will waste a lot of memory and operation time. An easier way to get to the required data can be achieved using `fseek()`.

As the name suggests, `fseek()` seeks the cursor to the given record in the file.

Syntax of fseek()

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

The first parameter stream is the pointer to the file. The second parameter is the position of the record to be found, and the third parameter specifies the location where the offset starts.

Different whence in fseek()

Whence	Meaning
SEEK_SET	Starts the offset from the beginning of the file.
SEEK_END	Starts the offset from the end of the file.
SEEK_CUR	Starts the offset from the current location of the cursor in the file.

Example 5: fseek()

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

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

int main()
{
  int n;
  struct threeNum num;
```

```
   FILE *fptr;

   if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
       printf("Error! opening file");

       // Program exits if the file pointer returns NULL.
       exit(1);
   }

   // Moves the cursor to the end of the file
   fseek(fptr, -sizeof(struct threeNum), SEEK_END);

   for(n = 1; n < 5; ++n)
   {
     fread(&num, sizeof(struct threeNum), 1, fptr);
     printf("n1: %d\tn2: %d\tn3: %d\n", num.n1, num.n2, num.n3);
     fseek(fptr, -2*sizeof(struct threeNum), SEEK_CUR);
   }
   fclose(fptr);

   return 0;
}
```

This program will start reading the records from the file program.bin in the reverse order (last to first) and prints it.

TYPES OF FILE PROCESSING

- Sequential access
- Random access

Random Access

If we want to read access record in the middle of the file and if the file size is too large sequential access is not preferable. In this case, random access to file can be used, which allows to access any record directly at any position in the file.

C supports three functions for random access file processing:
fseek()

ftell()
rewind()

- **fseek()**
  This function is used for setting the file pointer at the specified byte.

  Syntax

  int fseek(FILE *fp, long displacement, int origin);

  Here,
  fp – file pointer
  displacement – It denotes the number of bytes which are skipped backward or forward from the position specified in the third argument. It is a long integer which can be positive or negative
  origin – It is the position relative to which the displacement takes place.

- **ftell()**
  This function returns the current position of the file position pointer. The value is counted from the beginning of the file.

  Syntax

  long ftell(FILE *fp);

- **rewind()**
  This function is used to move the file pointer to the beginning of the file. This function is useful when we open file for update.
  Syntax:
  rewind(FILE *fp);
  Here, fp is file pointer of type FILE

Example program to understand rewind() function

```
#include<stdio.h>
void main()
{
        FILE *fp;
        fp=fopen("student.bin","rb+");
        if(fp==NULL)
        {
```

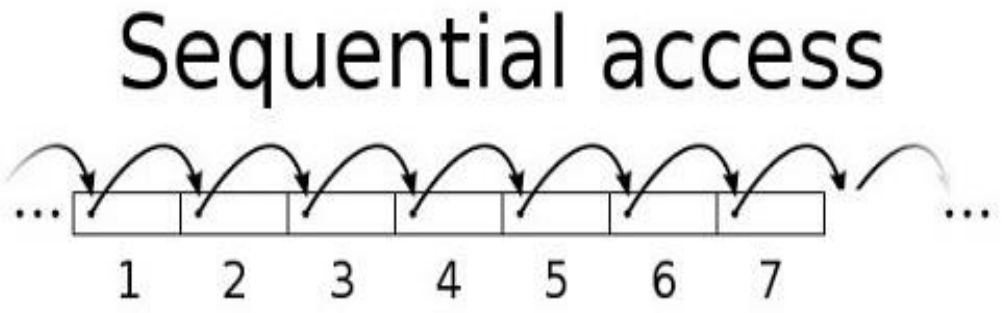
```
                printf("Error in opening file\n");
        }
        printf("Position pointer %ld\n",ftell(fp));
        fseek(fp,0,2);
        printf("Position pointer %ld\n",ftell(fp));
        rewind(fp);
        printf("Position pointer %ld\n",ftell(fp));
fclose(fp);
}
```

Sequential Access

Sequential Access to a data file means that the computer system reads or writes information to the file sequentially, starting from the beginning of the file and proceeding step by step.

Example

Sequential access is reading any file by searching from the storage device from the beginning. Let's suppose that there are four files stored in the storage device. Now if you want to access 3rd file then you start searching the 3rd file from beginning of the storage device i.e. 1, 2, 3, 4. You have to go through 1 and 2 to access 3. Similarly if you open any file then data is also saved in file in sequential i.e. old data of the file is stored in old index and new data is stored in higher index. Now to access the data in the file computer has to access data from lower index to higher.

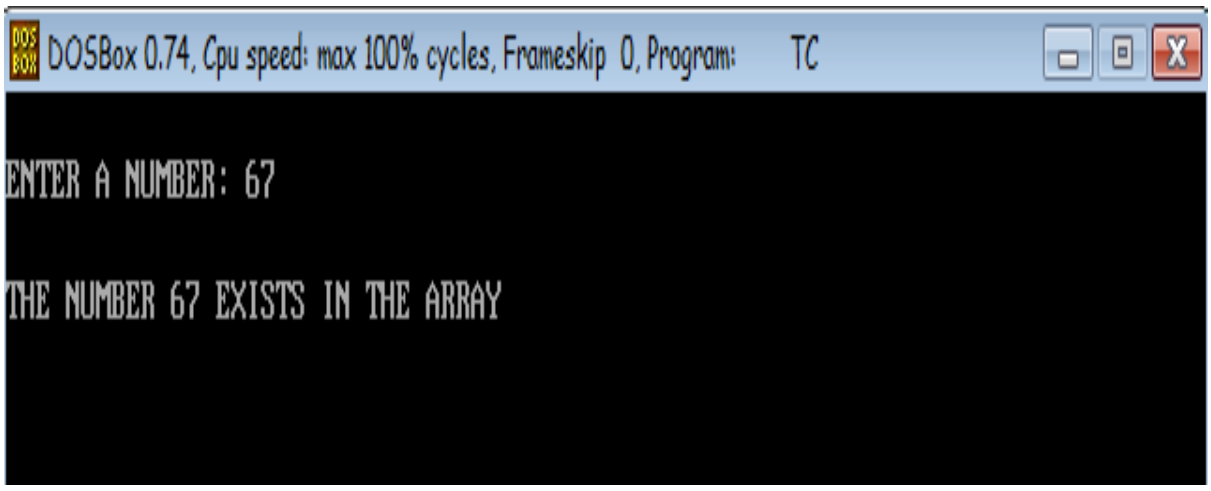


C program to search a key number in an array using Sequential Search Method.

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

```
        int arr[]={12,23,78,98,67,56,45,19,65,9},key,i,flag=0;
        clrscr();
        printf("\nENTER A NUMBER: ");
        scanf("%d",&key);
        for(i=0;i<10;i++)
        {
                if(key==arr[i])
                        flag=1;
        }
        if(flag==1)
                printf("\nTHE NUMBER %d EXISTS IN THE ARRAY",key);
        else
printf("\nTHE NUMBER %d DOES NOT EXIST IN THE ARRAY",key);
        getch();
}
```

OUTPUT



COMMAND LINE ARGUMENTS

Command line argument is a parameter supplied to the program when it is invoked. Command line argument is an important concept in C programming. It is mostly used when you need to control your program from outside. Command line arguments are passed to the main() method.

Syntax:

int main(int argc, char *argv[])

Here argc counts the number of arguments on the command line and argv[ ] is a pointer array which holds pointers of type char which points to the arguments passed to the program.

Example for Command Line Argument

```
#include <stdio.h>
#include <conio.h>
int main(int argc, char *argv[])
{
   int i;
   if( argc >= 2 )
   {
      printf("The arguments supplied are:\n");
      for(i = 1; i < argc; i++)
      {
         printf("%s\t", argv[i]);
      }
   }
   else
   {
      printf("argument list is empty.\n");
   }
   return 0;
}
```

Remember that argv[0] holds the name of the program and argv[1] points to the first command line argument and argv[n] gives the last argument. If no argument is supplied, argc will be 1.

Properties of Command Line Arguments:

1. They are passed to main() function.
2. They are parameters/arguments supplied to the program when it is invoked.
3. They are used to control program from outside instead of hard coding those values inside the code.
4. argv[argc] is a NULL pointer.
5. argv[0] holds the name of the program.
6. argv[1] points to the first command line argument and argv[n] points last argument.

C program to illustrate

```
// command line arguments
#include<stdio.h>

int main(int argc,char* argv[])
{
   int counter;
   printf("Program Name Is: %s",argv[0]);
   if(argc==1)
```

```
      printf("\nNo Extra Command Line Argument Passed Other Than Program
Name");
   if(argc>=2)
   {
      printf("\nNumber Of Arguments Passed: %d",argc);
      printf("\n----Following Are The Command Line Arguments Passed----");
      for(counter=0;counter<argc;counter++)
         printf("\nargv[%d]: %s",counter,argv[counter]);
   }
   return0;
}
```

Output in different scenarios:

1. **Without argument:** When the above code is compiled and executed without passing any argument, it produces following output.

```
$ ./a.out

Program Name Is: ./a.out

No Extra Command Line Argument Passed Other Than Program Name
```

2. **Three arguments:** When the above code is compiled and executed with a three arguments, it produces the following output.

```
$ ./a.out First Second Third

Program Name Is: ./a.out

Number Of Arguments Passed: 4

----Following Are The Command Line Arguments Passed----

argv[0]: ./a.out

argv[1]: First

argv[2]: Second

argv[3]: Third
```

3. **Single Argument :** When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following output.

```
$ ./a.out "First Second Third"

Program Name Is: ./a.out

Number Of Arguments Passed: 2

----Following Are The Command Line Arguments Passed----
```

```
argv[0]: ./a.out

argv[1]: First Second Third
```

4. **Single argument in quotes separated by space :** When the above code is compiled and executed with a single argument separated by space but inside single quotes, it produces the following output.

```
$ ./a.out 'First Second Third'

Program Name Is: ./a.out

Number Of Arguments Passed: 2

----Following Are The Command Line Arguments Passed----

argv[0]: ./a.out

argv[1]: First Second Third
```