Paper 102: Programming & Problem solving through C

Lecture-20:Unit-III Files

Introduction-understanding how C performs input/output using streams

- A *stream* is a logical channel on which input/output (I/O) is performed
- Three *standard streams* are opened automatically when a program starts and closed when the program ends; they are:
  - standard input (**stdin**): associated with the keyboard
  - standard output (**stdout**): associated with the monitor
  - standard error (**stderr**): also associated with the monitor
- I/O to the standard streams is *formatted text*; values are converted from their internal byte (machine) representation into a sequence of (human-readable) characters by standard library (**stdio.h**) functions:
  - Output to **stdout** is done with **printf** (also **putchar** and **puts**)
  - Input from **stdin** is done with **scanf** (also **getchar** and **gets**)

Introduction

- I/O can also be performed on streams that the programmer defines
  - associated with a file on a secondary storage device, to use these streams, they must be:
    - declared
    - opened (and tested to make sure no error occurred in the process)
    - closed when no longer needed
- I/O on non-standard streams can be done with **fscanf** (input) and **fprintf** (output), both in the **stdio.h** standard library

Declaring a Stream for a File

- Before a stream can be used, it must be declared, using this syntax:
- **FILE ****stream_name***;**
- **FILE** is a data type (a structure) defined in **stdio.h**; it will contain data about the type of file (input or output) and where the *file position pointer* is currently located
- ***stream_name*** is any valid C identifier
  - it will be used to reference the stream in all I/O functions
  - the * in the declaration indicates it is not actually a **FILE** type, but will contain the address of a **FILE** structure
- example of a stream declaration:
- **FILE *infile;   // stream name is infile**
- streams are declared at the beginning of the block in which they are referenced, just like other variables in C

Opening a Stream

- The **standard streams** are *preconnected* input and output channels between a computer program and its environment when it begins execution.
- The three I/O connections are called **standard input** (**stdin**), **standard output** (**stdout**) and **standard error** (**stderr**).
- Once a stream has been declared, it must be *opened* before any I/O can be performed on it
- Opening a stream requests the operating system (Windows, Linux, MAC, etc.) to:
  - locate the file on disk
  - prepare the buffer space and initialize the variables in the **FILE** structure needed for I/O
- If the operating system is successful, it returns the memory address at which the **FILE** has been allocated; if not successful, it returns the value **NULL**
- Here is the syntax for opening a stream associated with a disk file:

stream_name **= fopen(*file_name, mode*);**

Opening a Stream

- ***file_name*** is a string specifying the name of the file to open; ***mode*** is one of the following for text I/O:

Mode	**Meaning**
r	Open for reading only (input); returns **NULL** if file does not exist
w	Create for writing (output); overwrites existing file
a	Append: open for writing at **eof,** or create new file for writing if it does not already exist
r+	read existing contents, writes new ones and modify
w+	overwrite existing contents, write new contents, read and modify
a+	read existing contents, append new contents .cannot modify

- Here is an example of opening the file

  c:\temp\data.txt for input (in Windows),

  assuming the stream **infile** has already been declared:

- **infile = fopen("c:\\temp\\data.txt", "r");**

- Note that the ***file_name*** string requires two backslash characters (\\) to represent a single backslash, if you are typing a string literal; otherwise, the backslash is interpreted as the start of a control character, like **'\n'** (newline)

Opening a Stream

- Another way to do this is to ask the user what file to open; to do this, you will need char array to store the input, which should be declared to be the longest string that can be given to system calls, such as open; this is the symbolic constant **CHAR_MAX** in the header file **limits.h**

-
  ```
  #include <limits.h>
  …
  char inFileName[CHAR_MAX];    /* filename */
  FILE *inFile;                 /* stream name */
  …
  puts("What file to open?");
  gets(inFileName);
  inFile = fopen(inFileName, "r");
  …
  ```

Verifying a File Has Been Successfully Opened

- Before attempting any I/O on a stream that has been opened, test the stream to make sure the operating system was successful (by not returning a **NULL**):
-

```
inFile = fopen("c:\\temp\\data.txt", "r");

if (inFile == NULL) {
  printf("File could not be opened.\n");
  exit(1);
}

/* if we get here, stream is ready for I/O ... */
```

Writing to a File

- Once a file has been successfully opened in one of the output modes (**w** or **a**) we can write to it with **fprintf**, whose syntax is:
- **fprintf(*stream_name*, *format_specifier*, val …);**
- This is exactly the same as **printf**, with the addition of a new first argument, the ***stream_name***; all conversion specifiers and formatting work exactly the same as they do in **printf**
- Example (assuming stream has been declared and successfully opened):
- **fprintf(outFile, "%d %.2f\n", intVar, floatVar);**

Reading from a File

- Once a file has been successfully opened for input (**r**) we can read from it with **fscanf**, whose syntax is:

  fscanf(*stream_name*, *format_specifier*, val …);

- This is exactly the same as **scanf**, with the addition of a new first argument, the ***stream_name***; all conversion specifiers and formatting work exactly the same as they do in **scanf**
- Example (assuming stream has been declared and successfully opened):

  ```
  fscanf(inFile, "%d%lf", &intVar, &floatVar);
  ```

- File input is usually done inside a loop – an **fscanf** is executed repeatedly until there is no more data in the file
- We can determine if we are at the end of a file (**EOF**) with the **feof** function, which returns a 1 (true) if we have reached **EOF** or 0 (false) if we have not; it is usually used like this:

Reading from a File

- /* read data item from file (priming input) */

  ```
  while (!feof(stream_name)) {
    /* do something with data just read … */
    /* read next data item (working input) */
  }
  ```

  e.g.,

  ```
  while(!feof(infile)
  { /* read or write instructions */
  }
  ```

- Another option for reading data (in this case an integer) from a file:

  ```
  while (fscanf(stream_name, "%d", &intVar) == 1)
  {
        stmts; …
      }
  ```

Rereading a File

- Each time you read from a file, it's file position pointer is updated, so that the next time you read the file, you will obtain the next value from it; this continues until the end of the file is reached – which you can test with the **feof** function, as described above
- There may be times when you need to reread the file; to do this, you can move the file position pointer back to the beginning of the file with:
- **rewind(*stream_name*);**
- Example (assuming the stream is open):
- **rewind(inFile);**

Closing a File

- When all I/O has been performed on a file, the associated stream should be closed:
- **fclose(*stream_name*);**

File Handling Functions

- The standard library in C has many functions related with files
- fgetc()

  Declaration: int fgetc(FILE *stream);

  e.g. c=fgetc(f1);

  fgtec returns the next character in the named input stream

  Return value: on success it returns the character read, on end of file it return EOF
- fputc()
- Declaration: int fputc(int c, FILE *stream);
- E.g. fputc(c,f1);
- The function fputc outputs the value of the character c to the specified stream
- Return value: on success it return the character c, on error it returns EOF
- EOF is a signed integer having value -1

File Handling Functions

- The loop which writes to the file is:

```
while( (c=getchar())!=EOF)
        fputc(c,fp);
```

File Handling Functions

- fputs() and fgets()
- fgets gets a string from a stream
- fputs outputs a string to a stream

- Declaration:

```
char  *fgets(char *s, int n, FILE *stream);
int  fputs(const char *s, FILE *stream);
```

- fgets reads characters from stream into the string s. It stops when it reads either n - 1 characters or a newline character, whichever comes first.
- fgets retains the newline character at the end of s and appends a null byte to s to mark the end of the string.

File Handling Functions

- fputs copies the null-terminated string s to the given output stream.
- It does not append a newline character, and the terminating null character is not copied.
- Return Value:
- On success,
  - fgets returns the string pointed to by s.
  - fputs returns the last character written.
- on end-of-file or error,
  - fgets returns null.
- On error,
  - fputs returns EOF.

Example of creating a text file

```
#include<stdio.h>
void main()
{
	FILE *fp;
	char c;
	fp=fopen("file1.txt","w");
	if(fp==NULL)
		exit();
	printf("\n enter a few lines\n");
	fflush(stdin);
	while(1)
	{
		c=fgetc(stdin);
		if(c==EOF)
				break;
		else
				fputc(c,fp);
	}
	fclose(fp);
}
```

Example of reading an existing file

```
#include<stdio.h>
#include<conio.h>
void main()
{
        FILE *fp;
        char c;
        fp=fopen("z:\cprogram\file1.txt","r");
        if(fp==NULL)
        {
         printf("Opening file error\n");
                                exit();
        }
        clrscr();
        printf("\n contents of the file\n");
        while(1)
        {
                c=fgetc(fp);
                if(c==EOF)
                                break;
                else
                                printf("%c",c);
        }       fclose(fp);
}
```

Class assignment

- WAP to create a text file and store a small paragraph in it. In the same program read the file and count how many vowels are there, display the whole file and in the last line display the total number of vowels.
- WAP to create a formatted file to store a list of books, and allow the user to view any book by specifying a book ID. Book information such as Book ID, Book name, author, quantity, edition, publisher, etc…