Files in C++

We read data into memory, and process it to produce the desired output. We generally type our input using the keyboard, and produce the output on the terminal screen. When input data is voluminous, we have learned to use either an input file, using the input redirection '<.' or an output file, created using output redirection '>'. Disk files store data persistently. These files can be accessed and processed at any time, with an appropriate program.

A file on a disk is regarded as a sequence of bytes. Since any such sequence is logically equivalent to an array of bytes, we should be able to read or write data at any byte position, just as we do with elements of an array. We will see that this can indeed be done. However, reading and writing arbitrary bytes will serve no purpose, unless we know at which position, and how many bytes we wish to read or write. We thus need to organize data in disk files in some meaningful way. This is usually accomplished by designing a 'record' structure for information about entities which we wish to manage. Each record can contain certain meaningful 'fields' which contain values of different attributes describing an entity. A text file containing information about students, is one such example. Each line of the file may contain roll number, name, batch number, and marks of one student. Multiple such lines amount to 'records' for so many students. However, such lines may contain strings of varying lengths, forcing records to be of different sizes. Such text data can be read or written sequentially, one line at a time; but we cannot directly access a particular line as we cannot know the byte position at which the line starts.

C++ permits us to define 'structures' of fixed length, resembling a record layout. The structure is essentailly a way to define a 'new' data type (like basic data types int, char, float, etc.). This *type* stands for the entire collection of all component fields, and is given a suitable name chosen by us. Variables of such a *structure type* can now be defined. We can access and modify individualbers of a structure variable. An entire structure as a whole, can be written to, or read from a file. Since a structure has a fixed number of bytes allocated to it. Given a record number in a file, we can identify exactly the byte position at which a particular record is presnt, or shuld be written. Here are two examples of stcutures used to describe our sample data for a student:

Example 1.

```
struct textstudentinfo {
  char  roll[6];
  char  batch[4];
  char marks[6]
  char name[30];
};
----
```

Example 2.

```
struct studentinfo
{
  int roll;
  char name[30];
  int batch;
  float marks;
};
------
```

Persistent file storage is vital in real world applications. In most cases, the volume of data is so large that it cannot be affordably read and processed entirely in main memory. A case in point is that of processing information in bank accounts. All bank these days, maintain data for accounts of all customers often numbering over 2 to 3 Crores (200 to 300 Million). All these records are stored in disk files (called Databases). Whenever you withdraw any amount, e.g. from an ATM, the amount paid to you has to be debited from your balance, and in place of the old balance, the new balance has to be recorded on the disk. Each transaction must also be recorded for future reference. Accessing data from a traditional storage disk is roughly 1000 times slower than accessing the data in main memory. Since we cannot quickly read and process all records sequentially, a facility is needed to directly access the desired record. This is possible with 'binary' file type in C++. Important file handling functions are listed below. Programs in lecture sessions illustrate their use.

Summary of some important library (cstdio) functions for handling files.

A file in C++, residing on the disk, has a *name* (e.g. “mydata.txt”, “db.bin”, etc.), associated permissions, a *size* (no. of bytes stored in the file), and a *location* or *path* on the disk known to the OS. It is logically equivalent to an array of *size* bytes on the disk. It can be either a *text* file (default), or a *binary* file. In a C++ program, each file is known by a FILE pointer. When we open a file, the pointer gets associated with a physical file. It then becomes the effective name of the file in our program, till it is closed.
Array elements in memory are accessed by an *index,* e.g. A[i] – meaning ith byte. Instead, a *position* index can is used to define the offset within the file. It must be of the type long int, e.g. long POS. For each open file, an internal position indicator is maintained, which can be accessed or set. Using this, any byte position can be accessed, and required number of bytes can be read or written. Following are the major functions used for reading from, or writing to the files.

fopen()	fp = fopen (“Filename”, “mode”); mode is: r (read), w (write), a (append), r+ (read and write), w+(create and read) : Text file is the default; modes with character 'b' added mean Binary file. Returns a pointer of type FILE. Returns NULL pointer if file cannot be opened.
fclose()	fclose(fp); returns 0 on success; EOF on failure.
fseek()	fseek(fp, POS, origin); POS is relative offset (long int), origin is either SEEK_SET (begining of the file), SEEK_CUR (current position, SEEK_END (end of file).
ftell()	POS = ftell(fp); Returns current position of the file.
Rewind()	rewind(fp); Restores file position to the begining of the file
fread()	fread(&s, size, count, fp); reads *count* number of records of *size* bytes into memory pointed to by &s. (size * count number of bytes are deposited into target memory, pointed to by &s. *s* can be an array, a structure, or other contiguous block of bytes. Returns: Number of records actually read.
fwrite()	fwrite(&s, size, count, fp); same as above, except that *count* number of records (count*size number of bytes) are written from *s* to to the file. Returns: Number of records actually written.
fgetc()	char c; c = fgetc(fp); Returns: character read from the file at current POS; EOF if file ends.
fgets()	char str[100]; fgets(str, num, fp); Read characters from fp, up to either a new-line ('\n') or EOF, into *str*. String is terminated by '\0', upto . '\n' is also inculded in *str.* A maximum of *num* characters are read, even if there are more in the input string.
fscanf(), and fprintf()	Exactly like scanf(), and printf(); but the text is either read from, or written to fp, not stdout or stdin. The first parameter is fp, e.g. fprintf(fp, “%d %d \n”, i, j);