Introduction to File Input/Output

Introduction

- All Input/Output up to now has been done from/to the keyboard/screen.
- There is a serious limitation with this: ***when the program terminates, the data is lost.*** So we can't write a database program, which stores information and retrieves this information next time the program is started. Or a word processor. Or an email system. Or a spreadsheet. Or …
- We can however use external storage devices like the computer's hard disk for storage. Instead of writing output to the screen, we can write it to a file. ***This data won't be lost when the program terminates.***

Binary and Text Files

C can handle two types of files: ***binary*** files and ***text*** (or ASCII) files.

- The difference between the two file types is in the way they store numeric data. If numeric data is stored in a binary file, it is stored as binary numbers. If numeric data is stored in a text file, each digit of the numeric data is converted to its ASCII format and stored like that.
- So the number 123 requires three bytes of storage in a text file because it has three digits: '1', '2', and '3'.

Introduction to File Input/Output (contd.)

- 123 only requires 2 bytes of storage in a binary file because the binary representations of integer values take up two bytes. 123 is stored as

00000000	01111011

- So binary files are more compact for storing data which is primarily numeric. Also, numeric data must be converted to its ASCII form before writing to a text file and converted from its ASCII form when reading from a text file. With binary files, no such conversions are necessary.
- However, text files are easily displayed on the screen (using e.g. the Windows notepad editor), so their contents can be easily examined or checked. They can also be read easily by other programs. Neither of these is true for binary files.

Important concepts:

- **File** -- a named set of data stored together on the disk. C views any file as a stream of bytes – it's up to the programmer to give it a structure.
- to use: (1) **open** the file; (2) read/write from/to the file; (3) **close** file.

Input from File – Example

```
#include <stdio.h>
void main(void){
int var1, var2; /* not initialised */
FILE *fp;
 /* fp is a variable of type "FILE *" and is */
 /* called a file pointer. The datatype      */
 /* "FILE" is defined in stdio.h, and is a   */
 /* structure in which C keeps information   */
 /* about a file */
fp=fopen("datafile.txt", "r"); /* open read-only */
fscanf(fp, "%d %d", &var1, &var2);
fclose(fp);
printf("first value read in was %d\n", var1);
printf("second value read in was %d\n", var2);
}
```

Contents of file **`datafile.txt`**:

```
3
-1
```

Produces the screen output:

```
first value read in was 3
second value read in was -1
```


Input from File – Example (contd.)

```
fp=fopen("datafile.txt", "r");
```

This statement "associates" the file **`datafile.txt`** with the file pointer **`fp`**, and tells the C compiler that the program is opening **`datafile.txt`** for reading only.

2nd argument to **`fopen()`** gives the file's "open mode" – possibilities include:

`r` open an existing file for ***reading only***

`w` open a new file (or overwrite an existing file) for ***writing only***

`a` open a file for ***appending*** (writing at the end of the file) – creates a new file if it doesn't currently exist

`r+` open an existing file for ***reading and writing***, starting at the beginning of the file

`w+` open a new file (or overwrite an existing file) for ***reading and writing***

`a+` open a file for ***reading and appending*** – creates a new file if it doesn't currently exist

Input from File – Example (contd.)

```
fscanf(fp, "%d %d", &var1, &var2);
```

Enables formatted input from the file pointed to by **fp**, otherwise has the same functionality as **scanf()**

```
fclose(fp);
```

When program is finished with an open file, it should "break" the association that was formed when the file was opened. This **fclose()** statement closes the file pointed to by **fp**.

Any files which have not been closed by **fclose()** are automatically closed (by the Operating System) when program execution is finished.

Output to File – Example

```
#include <stdio.h>
void main(void){
int var1, var2; /* not initialised */
FILE *fp;
FILE *fptr;
fp=fopen("datafile.txt", "r");
fscanf(fp, "%d %d", &var1, &var2);
fclose(fp);
fptr=fopen("datafile.txt", "w"); /* open write-only */
fprintf(fptr, "%d\n%d", var2, var1); /* swap values */
fclose(fptr);
}
```

Contents of file **datafile.txt before** program execution:

```
3
-1
```

Contents of file **datafile.txt after** program execution:

```
-1
3
```

(so in this case, the contents of **datafile.txt** have been overwritten)

Output to File – Example (contd.)

```
fptr=fopen("datafile.txt", "w");
```

Now we open **datafile.txt** again, this time for writing. We use a different file pointer, **fptr**, to point to the opened file.

```
fprintf(fptr, "%d\n%d", var2, var1);
```

Enables formatted output to the file pointed to by **fp**, otherwise has the same functionality as **printf()**

```
fclose(fptr);
```

Again, it is important to close the file when the program is finished with it.

Output to File – Example (alternative solution)

```
#include <stdio.h>
void main(void){
int var1, var2; /* not initialised */
FILE *fp;
fp=fopen("datafile.txt", "r");
fscanf(fp, "%d %d", &var1, &var2);
fp=fopen("datafile.txt", "w"); /* change mode */
fprintf(fp, "%d\n%d", var2, var1);
fclose(fp);
}
```

Contents of file **datafile.txt before** program execution:

```
3
-1
```

Contents of file **datafile.txt after** program execution:

```
-1
3
```

Output to File – Example of appending

```
#include <stdio.h>
void main(void){
int var1, var2; /* not initialised */
FILE *fp;
FILE *fptr;
fp=fopen("datafile.txt", "r");
fscanf(fp, "%d %d", &var1, &var2);
fclose(fp);
fptr=fopen("datafile.txt", "a");
fprintf(fptr, "%d\n%d", var2, var1);
fclose(fptr);
}
```

Contents of file **`datafile.txt` before** program execution:

```
3
-1
```

Contents of file **`datafile.txt` after** program execution:

```
3
-1
-1
3
```


Reading data from a file

First, you have to know the name of the file and where it is stored.

You also need to know how the data is laid out in the file. In particular:

- Order and datatype(s) of the data values.
- How much data is in the file. 3 common methods for this:
  - First line tells the number of data "records" that follow.
  - Sentinel signal: a value outside the range of actual data values which indicates that the end of the data has been reached.
  - Test to see if the special ***end-of-file indicator*** (inserted into every file by the Operating System) has been reached. This is usually the best choice.

Of course, if you write data to a file, you may have to decide which way of storing the data is most suitable for your application.

Reading data from a file: Examples

Contents of **`datafile1.txt`**:

```
3
0.2    80
0.7    60
0.1    50
```

(here, the first value tells us the number of lines of data – to read them, you should use a ***counter-controlled loop***…)

Contents of **`datafile2.txt`**:

```
0.2    80
0.7    60
0.1    50
-99    -99
```

(here, the sentinel values tell us that the end of the data has been reached – you should use a ***loop and test for the sentinel values each time around***…)

Contents of **`datafile3.txt`**:

```
0.2    80
0.7    60
0.1    50
```

(here, the file is just data – you should check whether each read got the end-of-file indicator. An example program to do this is given next)

Reading data from a file: Example program

```
#include <stdio.h>
void main(void){
float prob, avg=0.0;
int quantity, num_values=0;
FILE *fptr;
fptr=fopen("datafile3.txt", "r");
while(fscanf(fptr, " %f %d", &prob, &quantity)==2){
 /* if return value from this fscanf() is not 2, */
 /* the end-of-file indicator has been reached   */
 avg = avg + (prob*quantity);
}
fclose(fptr);
printf("average value is %.2f\n", avg);
}
```

Produces the correct screen output: **`average value is 63.00`**

Note: return value from **`fscanf()`** – and **`scanf()`** – is the number of successful assignments. So in this case, this return value should be 2 every time a valid data "record" has been read. Otherwise – error or end-of-file.