Chapter 4

File Processing

4.1 Motivation

Several programs that we have written required data to be read via the standard input device and store the values to variables associated with contiguous bytes in the primary memory. An example of an input instruction is `scanf("%d", &n);` which requires the user to input the value of an integer variable named as `n`.

There are several practical problems that would require programs to read data NOT from the standard input device but from a file in the secondary memory.[1] One reason why data have to be read from a file is that it could be be impractical for the user to input (a lot of) values interactively and repetitively such as in the following code snippet

```
for (i = 0; i < 10000; i++)
     scanf("%d", &A[i]);
```

Another reason is that we need to save data from the primary memory into secondary memory because the primary memory is volatile. For example, let us assume that we are typing a research paper that would require several lines, paragraphs and pages. We would normally use a word processor to create and edit the document. At some point in time, we would choose a functionality such as `"File"` followed `"Save"` save the current document that we have edited as a file in the secondary memory. Later on, this file can be opened (i.e., `"File"`, `"Open"`), and the contents will be opened and copied onto the primary memory.

[1] Secondary storage devices include hard disk, mobile disk, DVD, etc.

In this chapter we will learn how to read data from a file, and how to write data to a file. We will consider both text and binary files.

4.2 What is a File?

A file is a group of elements. The elements are essentially a collection of data stored in memory, specifically, in the secondary memory, for example in the hard disk or mobile disk. There are basically two types of files, namely: *text* file and *binary* file.

C language does not actually have keywords or commands for handling input/output. Instead, there are several library functions which are part of what is now known as standard input/output library. The function prototypes are declared inside `stdio.h` file. In this chapter, we will learn several library functions that will allows us to do the following: (i) open a file, (ii) close a file, (iii) create a new file, (iv) read data from an existing file, (v) write data to a new file and (vi) append data to an existing file. Details are provided in the following sections.

4.3 Text File

A *text file* is a file containing elements all of type `char`. These characters are encoded in a standard format such as ASCII (American Standard Code for Information Interchange) or UNICODE (for international encoding).[2]

Text files can be opened by text editors such as `edit`, `Notepad` or `vi`.[3] The contents of a text file can also be displayed in the console by commands such as `type <filename>` in Windows command line or `cat <filename>` in UNIX and Linux.

Example: Assume a text file `"SAMPLE.TXT"` containing two lines of text as follows:

1 APPLE
2 BANANAS

The characters in the text file are listed in Table 4.1. Notice the presence of the newline character after letter 'E'.

[2]Different cultures use different characters in their written languages.

[3]`Edit` and `Notepad` are text editors in Windows OS and `vi` is an editor in UNIX and Linux.

Table 4.1: Example Characters and Their ASCII Codes

Character	ASCII Code
'1'	49
' '	32
'A'	65
'P'	80
'P'	80
'L'	76
'E'	69
newline	10
'2'	50
' '	32
'B'	66
'A'	65
'N'	78
'A'	65
'N'	78
'A'	65
'S'	83

Question: How do we know when/where the text file ends? That is, how do we know that we have already reached the last character in a text file?

Answer: There is a marker with a macro name of `EOF` which denotes the end of the file. `EOF` is an abbreviation of **E**nd **o**f **F**ile. In the example above, the `EOF` appears after character 'S'.

It is very important to note that the '1' stored in the text file is a character encoded as ASCII code 49. It is not numeric 1 of type integer.

► Self–Directed Learning Activity ◄

Determine what is the value of `EOF`. Hint: Write a C program that will print the value of `EOF`.

4.4 File Pointer Declaration

A *file pointer* is a pointer that will be mapped to an external file uniquely specified by its filename and extension. The syntax for file pointer declaration is:

```
FILE *<pointer name>;
```

Note that `FILE` is not a keyword in the C programming language. It is actually a type definition for a structure used in storing information about a file.

Examples of file pointer declarations are:

```
FILE *fp;   /* fp is taken to mean file pointer */

FILE *fp_input, *fp_output;
/* fp pointers to input file and output files */

FILE *fp1, *fp2, *fp3;
```

4.5 Opening and Closing a File

4.5.1 Opening a File

The first operation that should be done on files is to open it. The pre–defined library function for opening a file is `fopen()` with the following prototype:

```
FILE *fopen(<filename>, <mode>);
```

`fopen()` returns a pointer to the file (data type of `FILE *`)if the file exists; otherwise, it returns `NULL`. It has two parameters, namely:

- `<filename>` is a string indicating the filename and extension (in some cases, we may need to specify the relative or the absolute path) of the file
- `<mode>` is a string indicating the mode in which the file will be opened. There are several modes; we will limit our discussions to the following:
  - `"r"` open the file in read mode (i.e., input from the file)

- `"w"` open the file in write mode (i.e., output to a file); if the file exists, the original contents will be overwritten
- `"a"` open the file in append mode (i.e., output to a file); adds new entries at the end of the original file
- `"b"` this letter is added after `"r"` or `"w"` to indicate the fact that the file to be manipulated is a binary file.

Simple examples of using `fopen()` are shown in the following codes:

```
/* open a text file named "input.txt" in read mode */
fp_input = fopen("input.txt", "r");

/* open a text file named "output.txt" in write mode */
fp_output = fopen("output.txt", "w");

/* open a text file named "output.txt" in read mode */
/* file can be found in c:\data subdirectory */
fp1 = fopen("c:\data\sample.txt", "r");

/* open text file named "test.txt" in append mode */
fp2 = fopen("test.txt", "a");

/* open a binary file named "sample.dat" in read mode */
fp3 = fopen("sample.dat", "rb");
```

4.5.2 Closing a File

The last operation that should be performed on a file is to close it. The function prototype for this is:

```
fclose(<file pointer>);
```

WARNING: Make sure to close all files that were opened. It is possible that data maybe lost if the associated file is not closed. Some operating system will automatically close all files after a program's termination, but we should not depend on this behavior. Only one file at a time can be closed in one invokation of `fopen()`. Thus, if there are m number of files that need to be opened, then `fopen()` will need to be called m number of times. Examples of how to close (previously opened) files are given below:

```
/*example shows how to close three files */
fclose(fp);
fclose(fp_input);
fclose(fp_output);
```

ILLUSTRATIVE EXAMPLES

Listing 4.1 shows an example program that demonstrates what we learned so far, (i) how to declare a file pointer, (ii) how to open a file named as `"test.txt"` in read mode and later after performing some other tasks (iii) how to close the file. If the file does not exist in the same directory as the executable program, `fopen()` will return a `NULL` value. Otherwise, if the file exists, the address of the file will be stored in file pointer `fp`. All other file related operations will have the file pointer as a parameter (see sample programs in the latter sections).

Listing 4.1: `fopen()` with `"r"` mode sample program

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      FILE *fp;
7
8      fp = fopen("test.txt", "r");  /* open in read mode */
9
10     if (fp == NULL) {
11        printf("ERROR: file does not exist.\n");
12        exit(1);
13     }
14     else
15        printf("File opened successfully.\n");
16
17     /*-- other statements follow --*/
18
19     fclose(fp);  /* close the file */
20     return 0;
21 }
```

▶ Self–Directed Learning Activity ◀

1. Encode and compile the program in Listing 4.1. Test it for two possible cases:

(a) the file `"test.txt"` is not in the same directory as the executable file

(b) the file `"test.txt"` is present in the same directory.

2. It is possible to open a file that is not in the same directory as the executable file. As an experiment, do the following:

   (a) First, create a new directory named `MYDIR` in drive `D`. At this point in time, the subdirectory `MYDIR` is empty.

   (b) Next, modify `fopen()` command into `fopen("D:/MYDIR/test.txt", "r");`.

   (c) Compile and test the program. What is the result?

   (d) Next, make sure that there is a `test.txt` file in the `MYDIR` subdirectory. You may create a new file using a text editor. Alternatively, you may just move or copy an existing file into the said subdirectory.

   (e) Run the program again. What is the output?

Instead of hardcoding the filename, i.e., the first parameter of `fopen()`, we can ask the user to input it via `scanf()`. Listing 4.2 illustrates this technique.

Listing 4.2: More generalized filename

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      char filename[40];    /* filename and extension */
7      FILE *fp;
8
9      printf("Input filename: ");
10     scanf("%s", filename);
11
12     if ( (fp = fopen(filename, "r")) == NULL) {
13        printf("ERROR: %s does not exist.\n", filename);
14        exit(1);
15     }
16     else
17        printf("%s was opened successfully.\n", filename);
18
19     /*-- other statements follow --*/
20
21     fclose(fp);
22     return 0;
23 }
```

▶ Self–Directed Learning Activity ◀

Encode and compile the program in Listing 4.2. Test it for the following cases:

1. a file that does not exist
2. a file that is in the same directory as the executable file
3. a file that is in a different drive/directory

Listing 4.3 shows how to open a file named as `"sample.txt"` in write mode. Take note that the `"w"` mode is a destructive mode. This means that if a file already exists, the contents of the said file will be erased and will be overwritten.[4]. If it does not exist, a new file with the filename of `"sample.txt"` will be created.

Listing 4.3: `fopen()` with `"w"` mode sample program

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      FILE *fp;
7
8      fp = fopen("sample.txt", "w");  /* open in write mode */
9
10     /*-- other statements follow --*/
11
12     fclose(fp);
13     return 0;
14 }
```

▶ Self–Directed Learning Activity ◀

1. Encode and compile the program in Listing 4.3. Test it for the following cases:
   (a) the file does not exist
   (b) the file exists and contains some texts
2. What is the size of the file in the first case? What happened with the original file contents in the second case?

[4]Be careful! You may accidentally erase the contents of an important file if you incorrectly specified `"w"` instead of `"r"`.

A file can be opened and closed more than once within the same program or even within the same function. The following code snippet shows an example:

```
    FILE *fp;

    fp = fopen("test.txt", "r");   /* open 1st time */

    /*-- other statements --*/

    fclose(fp); /* close 1st time */

    /*--- then later on --*/

    fp = fopen("test.txt", "r");   /* open 2nd time */

    /*-- other statements --*/

    fclose(fp);                    /* close 2nd time */
}
```

Several files maybe opened, possibly in different modes, within the same program or even within the same function. For example:

```
    FILE *fp1, *fp2;

    fp1 = fopen("abc.txt", "r");  /* open first file */
    fp2 = fopen("def.txt", "w");  /* open second file */

    /*-- other statements --*/

    fclose(fp1);  /* close the files */
    fclose(fp2);
```

4.6 Formatted File Output

We are already familiar with the formatted output library function `printf()` which displays the output on the standard output device (display screen). `printf()` can be thought of as a specific case of the more generalized formatted output library function named `fprintf()`. Its function prototype is:

```
int fprintf(<file pointer>, <format string>, <expression>);
```

We first prove that `printf()` is just a specific case of `fprintf()` using Listing 4.4. To do this, we will need to use a predefined pointer `stdout` as the first parameter to `fprintf()`. `stdout` means standard output which in our case refers to the display screen. In C, it is automatically opened (we do not need to call `fopen()` with `stdout` and can be readily used in any part of the program. Everything that we did before using `printf()` can be done using `fprintf(stdout, ...)`.

Listing 4.4: `fprintf()` with `stdout`

```
1  #include <stdio.h>
2
3  int main()
4  {
5      /* no need to call fopen() with stdout */
6
7      fprintf(stdout, "Hello world!\n");
8
9      /* no need to call fclose() with stdout */
10     return 0;
11 }
```

► Self–Directed Learning Activity ◄

1. Encode, compile and test the program in Listing 4.4.
2. Add the following statements after `fprintf()` in Listing 4.4. Run and test the program. What are the results?

```
fprintf(stdout, "%d\n", 123);
fprintf(stdout, "%f\n", 6.3745674123456);
fprintf(stdout, "%.2f\n", 6.3745674);
```

The true purpose of `fprintf()` is to output/write data to be stored in secondary memory. Listing 4.5 shows how we can output the values of variables declared using the simple data types `char`, `int`, `float` and `double` into a named text file. It illustrates the four basic steps involved in text file output, namely:

1. Declare a file pointer.
2. Open the file for output using `fopen()` in write mode, that is, the second parameter is `"w"`.
3. Output data to the file using `fprintf()`. Note that `fprintf()` maybe called any number of times as necessary after `fopen()` and before `fclose()`.
4. Close the file using `fclose()`.

Listing 4.5: Output Basic Data Type Values

```
1  #include <stdio.h>
2  int main()
3  {
4      char ch = 'A';
5      int i = 123;
6      float f = 4.0;
7      double d = 3.1415159;
8
9      FILE *fp;
10
11     fp = fopen("sample.txt", "w");
12     fprintf(fp, "%c %d %.2f %lf", ch, i, f, d);
13     fclose(fp);
14     return 0;
15 }
```

► Self–Directed Learning Activity ◄

1. Encode, compile and test the program in Listing 4.5. How many characters do you think were written onto the text file?
2. Go to the command line, and change your subdirectory to the directory containing the `exe` file. What are the results for the following commands?
   (a) `dir sample.txt`
   (b) `type sample.txt`

Listing 4.6 is a program that illustrates how to output a sequence of numbers from 0 to 4 generated using a `for` loop to a file. We will compare the result of this program later with another program discussed in the section related to binary files.

Listing 4.6: Output Basic Data Type Values

```
1  #include <stdio.h>
2
3  int main()
4  {
5      char filename[40];
6      int i;
7      FILE *fp;
8
9      fprintf(stdout, "Input filename: ");
10     fscanf(stdin, "%s", filename);
11
12     fp = fopen(filename, "w");
13
14     for (i = 0; i < 5; i++)
15         fprintf(fp, "%d\n", i);
16
17     fclose(fp);
18     return 0;
19 }
```

4.7 Formatted File Input

We are already familiar with the formatted input library function `scanf()`. The counterpart function for formatted file input is the library function `fscanf()`. Its function prototype is as follows.

```
int fscanf(<file pointer>, <format string>, <address>);
```

Similar to the previous section, we first prove that `scanf()` is just a specific case of `fscanf()` using Listing 4.7. To do this, we will need to use a predefined name `stdin` as the first parameter to `fscanf()`. `stdout` means standard input device which in our case refers to the keyboard. In C, it is automatically opened (we do not need to call `fopen()` with `stdin` and can be readily used in any part of the program. Everything that we did before using `scanf()` can be done using `fscanf(stdin, ...)`.

Listing 4.7: `fscanf()` with `stdin`

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int n;
6
7      /* no need to call fopen() and fclose() with stdin and stdout */
8      fprintf(stdout, "Input an integer value: ");
9      fscanf(stdin, "%d", &n);
10     fprintf(stdout, "n = %d\n", n);
11     return 0;
12 }
```

► Self–Directed Learning Activity ◄

1. Encode, compile and test the program in Listing 4.4.
2. Modify the program by:
   (a) asking the user to input the value of a float variable and a double data type variable; provide prompts using `fprintf()`.
   (b) afterwards, output the values entered by the user using `fprintf()`.

The input in the previous program come from the standard input device `stdin`. How we can get the input if data are stored in a secondary memory? Listing 4.8 shows an example of how to read data from a named text file using `fscanf()`.[5]

The program illustrates the four basic steps involved in text file input, namely:

1. Declare a file pointer.
2. Open the file for input using `fopen()` in read mode, that is, the second parameter is `"r"`.
3. Input data from the file using `fscanf()`. Note that `fscanff()` maybe called any number. of times as necessary after `fopen()` and before `fclose()`.
4. Close the file using `fclose()`.

[5]In order for the program to work, the file `"sample.txt"` with the correct contents must exist. There will be a logical/run–time error if `"sample.txt"` is not found, or if the contents are incorrect. To ensure that this will not happen, run first the program in Listing 4.5.

Listing 4.8: Read data from a text file

```
#include <stdio.h>

int main(void)
{
      char ch;
      int i;
      float f;
      double d;
      FILE *fp;

      /* open input file */
      fp = fopen("sample.txt", "r");

      /* read data from the file */
      fscanf(fp, "%c %d %f %lf", &ch, &i, &f, &d);

      /* write data to stdout */
      fprintf(stdout, "ch = %c, i = %d, f = %f, d = %lf\n",
                      ch, i, f, d);

      /* close input file */
      fclose(fp);
      return 0;
}
```

▶ Self–Directed Learning Activity ◀

1. Encode, compile and test the program in Listing 4.8. What are the results when you run the program?

2. Modify the contents of `"sample.txt"` using an editor as Notepad. Re–run the program; what are the corresponding results?

4.8 How to Read ALL Data From the Input File

The previous program reads only four data values from the input file. It is possible that there are more data.

Many problems involving file processing would have to read ALL data from the input file. We will give in the following subsections several example problems and their corresponding techniques on how to solve them.

4.8.1 A program similar to the `type` command

The `type` command that we issue in Windows command line reads all the characters stored in the text file and outputs them on the screen. The algorithm for such a program is as follows:

1. Input the name of the file.
2. Open the file in `"r"` mode. Let us assume that it exists.
3. Read one character from the file using `fscanf()`.
4. If the return value of `fscanf()` is 1 it means that `fscanf()` successfully read one character, i.e., it is not yet the end of file.
   (a) Display the character on the screen.
   (b) Repeat step 3.
5. If the return value of `fscanf()` is not 1 it means that we have reached the end of file. Close the file.

The corresponding program is shown Listing 4.9. The technique uses a `fscanf()` inside a `while` loop to read all characters stored in the text file. The loop condition involves two operations actually, (i) read one character from the file using `fscanf()`, (ii) compare the return value of `fscanf()` with 1. A return value of 1 means that `fscanf()` successfully read one character from the file; otherwise, it means that there are no more characters and that we have reached the end of file.[6]

[6] Actually, there is a pair of pre–defined functions dedicated to single character file I/O. These are `fgetc()` and `fputc()`. We chose not to discuss them in these course notes. It is suggested that you consult the references if you are interested in learning more about these functions.

Another new concept introduced in the sample program is the use of the pre–defined name `stderr` which denotes the standard error device. This is the device that should be used when we want to display an error. The standard error device corresponds to the display screen in the platform that we are using.

Listing 4.9: A program similar to the `type` command

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

int main()
{
    char filename[40];
    char ch;
    FILE *fp;

    fprintf(stdout, "Input filename: ");
    fscanf(stdin, "%s", filename);

    if ((fp = fopen(filename, "r")) == NULL) {
       fprintf(stderr, "ERROR: %s does not exist.\n", filename);
       exit(1);
    }

    while ((fscanf(fp, "%c", &ch)) == 1)
        fprintf(stdout, "%c", ch);

    fclose(fp);
    return 0;
}
```

► Self–Directed Learning Activity ◄

1. Encode, compile and test the program in Listing 4.9. Test for files that exist and that do not exist.

2. Run the program again. When asked for the filename, input the filename you used when you save Listing 4.9. What is the result?

4.8.2 A program similar to the `copy` command

The `copy` command copies the contents of one file to another file. Its syntax is `copy <source file> <destination file>`. The `copy` command works for both text files and binary files. We show how to implement a similar functionality but it will be limited to copying text files only.

The algorithm for `copy` shown below is very much similar to that for `type`. The difference here is that instead of an `fprintf()` to the `stdout`, we do an `fprintf()` to the destination file.

1. Input the name of the source file.
2. Open the source file. Let us assume it exists.
3. Input the name of the destination file.
4. Open the destination file. (Note: If the file exists, the original contents will be overwritten.)
5. Read one character from the source file using `fscanf()`.
6. If `fscanf()` return value is 1:
   (a) Write the character onto the destination file.
   (b) Repeat Step 5.
7. If `fscanf()` return value is not 1, close the files.

The implementation is shown in Listing 4.10.

Listing 4.10: A program similar to the `copy` command

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      char source_filename[40];
7      char dest_filename[40];
8      char ch;
9      FILE *fp_source;
10     FILE *fp_dest;
11
```

```
12      fprintf(stdout, "Input source filename: ");
13      fscanf(stdin, "%s", source_filename);
14
15      /* open source file */
16      if ((fp_source = fopen(source_filename, "r")) == NULL) {
17         fprintf(stderr, "ERROR: %s does not exist.\n",
18                 source_filename);
19         exit(1);
20      }
21
22      fprintf(stdout, "Input destination filename: ");
23      fscanf(stdin, "%s", dest_filename);
24
25      /* open destination file in "w" mode */
26      fp_dest = fopen(dest_filename, "w");
27
28      /* copy one character at a time from source to dest. */
29      while ((fscanf(fp_source, "%c", &ch)) == 1)
30          fprintf(fp_dest, "%c", ch);
31
32      fclose(fp_source);
33      fclose(fp_dest);
34      return 0;
35  }
```

4.8.3 A program that reads a sequence of numbers from a text file

Let us assume that a text file contains a sequence of integers (actually, the digits corresponding to these numbers are stored as chacters). Furthermore, we assume that each line contains only one integer. What we do not know is how many integers are stored in the file. Our problem is to create a program that will read all the integers from the file and display them on the screen one integer per line of output.

The solution to this problem is very much similar to the `type` algorithm. The difference is that instead of reading one character at a time, we read integers. The reader is advised to create first an algorithm before reading the next page.

The program corresponding to the requirement is shown in Listing 4.11.

Listing 4.11: A program that reads a sequence of numbers from a file

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      char filename[40];
7      int value;
8      FILE *fp;
9
10     fprintf(stdout, "Input filename: ");
11     fscanf(stdin, "%s", filename);
12
13     if ((fp = fopen(filename, "r")) == NULL) {
14        fprintf(stderr, "ERROR: %s does not exist.\n", filename);
15        exit(1);
16     }
17
18     while ((fscanf(fp, "%d", &value)) == 1)
19         fprintf(stdout, "%d\n", value);
20
21     fclose(fp);
22     return 0;
23 }
```

4.8.4 A program that reads a sequence of tuples from a text file

A "tuple" is one group of values which maybe of different types. For this example, let us assume that each tuple is composed of 4 values: (i) a string of at most 20 characters, (ii) one integer, (iii) one float and (iv) one double in the given sequence.

Assume that a text file contains data corresponding to a sequence of tuples. Tuples are stored per line, and values within the same tuple are separated by white spaces (blanks, tabs). For example:

```
PEDRO 1  2.5  3.1
MARIA 6  0.5  8.9
JUAN -4 1.8  3.7
```

Our task is read all the tuples from a named text file and output each tuple one

line at a time on the display screen. Take note that we do not know in advance how many tuples are present.

The solution can be patterned from the algorithms in the previous subsections. We read the values for each tuple using `fscanf()`. A successful `fscanf()` should return a value equivalent to 4 which means that all 4 in a tuple were successfully read. In such a case, we output the tuple on the screen. If the `fscanf()` does not return a value equal to 4, then it means there are no more tuples, and therefore it is already the end of the file.

The solution is shown in Listing 4.12.

Listing 4.12: A program that reads a sequence of tuples from a file

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      char filename[40];
7      char name[20];
8      int i;
9      float f;
10     double d;
11
12     int count;
13
14     FILE *fp;
15
16     fprintf(stdout, "Input filename: ");
17     fscanf(stdin, "%s", filename);
18
19     if ((fp = fopen(filename, "r")) == NULL) {
20         fprintf(stderr, "ERROR: %s does not exist.\n", filename);
21        exit(1);
22     }
23
24     while ((fscanf(fp, "%s %d %f %lf", name, &i, &f, &d)) == 4)
25         fprintf(stdout, "%s %d %f %lf\n", name, i, f, d);
26
27     fclose(fp);
28     return 0;
29 }
```

The general rule that we should have learned from this example is that `fscanf()` returns the number of values that were successfully read from the file. If we are trying to read m number of values, and `fscanf()` returned m as a result, then we can conclude that it was a successful read; otherwise it was unsuccessful. An unsuccessful `fscanf()` will return a values less than m. This could mean two possibilities, (i) some values are missing (example, two values instead of the expected four values) or (ii) there are no more values, i.e., we have reached the `EOF`.

4.9 Opening a File in Append Mode

The last topic that we will cover in text file processing is how to open a file in append `"a"` mode. A text file opened in `"a"` mode has the following characteristics:

1. If the file does not exist yet, a new file will be created.
2. If the file exists, its contents are not overwritten (unlike in `"w"` mode). New data written to the file will be added (i.e., appended) towards the end of the file.

Listing 4.13 shows a simple program to demonstrate the concept. Every time the program is executed, it will ask the user to input 3 integers. Each integer is output to a file. If the named file already exists, these integers will be appended to the original contents.

Listing 4.13: `fopen()` with `"a"` mode sample program

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      char filename[40];
7      int i, n;
8      FILE *fp;
9
10     fprintf(stdout, "Input filename: ");
11     fscanf(stdin, "%s", filename);
12
13     /* open file in "a" mode (append) */
14     fp = fopen(filename, "a");
15
```

```
16      /* ask the user to input 3 integers, output
17         each integer to the text file */
18      for (i = 0; i < 3; i++) {
19          fprintf(stdout, "Input an integer value: ");
20          fscanf(stdin, "%d", &n);
21
22          /* output data to file */
23          fprintf(fp, "%d\n", n);
24      }
25
26      fclose(fp);
27      return 0;
28  }
```

▶ Self–Directed Learning Activity ◀

1. Encode and compile the program in Listing 4.13. Run the program three times or more times. For each execution, always input the same file name to see the effect of the append mode. The input values for the integers need not be the same (varying values are recommended though to see the effect).

2. What are the contents of the file during the first time you executed it? second time? third time?

4.10 Binary File Processing

A *binary file* is a file containing data encoded in binary, i.e., combinations of 0 and 1. Its format should be known in advance to be able to read and decode its contents properly. Examples of binary files are object and executable files.

There are four steps in manipulating binary files:

1. Declare a file pointer
2. Open the binary file
3. Access the binary file
4. Close the binary file

The file pointer declaration, and the commands for opening and closing the files are the same with binary files. The `"rb"` mode should be used as the second parameter of `fopen()` in order to open the file for input; while mode `"wb"` should be used to open it for output. In the following sections, we discuss how to read data from and write data to a binary file using the pre–defined functions `fread()` and `fwrite()`.

4.11 Writing Data into a Binary File

Data can be written to a binary file using the `fwrite()` function. Its corresponding syntax is as follows:

```
size_t fwrite(<buffer pointer>, <size>, <count>, <file pointer>)
```

Parameter `buffer pointer` is the base address of the element (for example, variable of a simple data type) or group of elements (for example, arrays and structures) to be written. The size, in number of bytes, of each element is indicated by parameter `size`, while `count` indicates the number of elements to be written. Thus, the total number of bytes to be written in the files is equivalent to `size` multiplied with `count`. The last parameter `file pointer` is the associated file pointer to the the named binary file.

The function returns an integral value corresponding to the number of elements written to a file. A successful `fwrite()` should return a value equivalent to the

third parameter denoted as `count`. If this is not the case, then a write error has occurred. There is a pre–defined function called `ferror()` that can be used to determine if an error has occurred. We will not discuss `ferror()` at this point in time.

We illustrate in the following subsections how to use `fwrite()` to output data based on the different data types that we learned in the past.

4.11.1 How to Write Values of Simple Data Types

We consider first how to write values of simple data types `char`, `int`, `float` and `double`. Listing 4.14 shows a simple program that writes just one character into a binary file named `"sample_01.dat"`.[7]

Take note of the parameters supplied to the `fwrite()` function call. The first parameter `&ch` is the base address of the memory space containing the value that we want to write to the file, the second parameter is the size of the `char` data type, the third parameter is 1 indicating that there is only one character to be written to the file, and `fp` is the file pointer.

Listing 4.15 extends the previous program by writing one character, one integer, one float and double data type values into a binary file named `"sample_02.dat"`. The first parameter for each of the `fwrite()` calls is the address of the variable, and the second parameter is the size corresponding to the variable's data type.

Listing 4.14: Writing one character into a binary file

```
1  #include <stdio.h>
2
3  int main()
4  {
5      FILE *fp;
6      char ch;
7
8      ch = 'A';
9      fp = fopen("sample_01.dat", "wb");
10     fwrite(&ch, sizeof(char), 1, fp);
11     fclose(fp);
12     return 0;
13 }
```

[7] We will use `"DAT"` as file extension throughout this document to denote data for a binary file.

Listing 4.15: Writing values of variables of simple data type

```
#include <stdio.h>

int main()
{
    char ch = 'A';
    int i = 123;
    float f = 4.56789;
    double d = 3.1415159;

    FILE *fp;

    fp = fopen("sample_02.dat", "wb");

    fwrite(&ch, sizeof(char), 1, fp);
    fwrite(&i, sizeof(int), 1, fp);
    fwrite(&f, sizeof(float), 1, fp);
    fwrite(&d, sizeof(double), 1, fp);

    fclose(fp);
    return 0;
}
```

► Self–Directed Learning Activity ◄

1. Encode, compile and test the programs in Listings 4.14 and 4.15. What do you think is the size of the binary file?

2. Go to the command line, and change your subdirectory to the directory containing the `exe` file. What are the results for the following commands?

   (a) `dir sample_01.dat`

   (b) `type sample_01.dat`

   (c) `dir sample_02.dat`

   (d) `type sample_02.dat`

4.11.2 How to Write Values of Group of Elements

We consider next how to write values for group of elements such arrays and structures. Listing 4.16 shows how to write a list of integer values stored in memory space which was allocated dynamically. The function call `fwrite(ptr, sizeof(int), n, fp)` indicate that the first parameter `ptr` is the base address of the memory space, each element has a size equal to `sizeof(int)`, and the number of elements is `n` – which is a user input in the sample program, and `fp` is the corresponding file pointer.

Listing 4.16: Writing values stored in dynamically allocated memory space

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int i;
7      int n;
8      int *ptr;
9      FILE *fp;
10
11     /* input number of integer elements to dynamically allocate */
12     printf("Input n: ");
13     scanf("%d", &n);
14
15     if ((ptr = malloc(sizeof(int) * n)) == NULL) {
16        fprintf(stderr, "ERROR: not enough memory.");
17        exit(1);
18     }
19
20     /* initialize elements */
21     for (i = 0; i < n; i++)
22         *(ptr + i ) = i * 5;
23
24     /* write data to binary file */
25     fp = fopen("sample_03.dat", "wb");
26     fwrite(ptr, sizeof(int), n, fp);
27     fclose(fp);
28     free(ptr);
29     return 0;
30 }
```

Listing 4.17 is essentially the same as the previous example except that the

memory space for a list of integers is stored in an array. Recall that memory space for the array is associated with static memory allocation.

Listing 4.17: Writing values stored in an array

```
#include <stdio.h>

int main()
{
    int i;
    int A[5];
    FILE *fp;

    /* initialize array elements */
    for (i = 0; i < 5; i++)
        A[i] = i * 100;

    /* write data to binary file */
    fp = fopen("sample_04.dat", "wb");
    fwrite(A, sizeof(int), 5, fp);
    fclose(fp);
    return 0;
}
```

► Self–Directed Learning Activity ◄

1. Encode, compile and test the programs in Listings 4.16 and 4.17. What do you think is the size of the binary file?

2. Change the value of the third parameter of `fwrite()` from 5 to 1. What do you think will be the effect? Try changing it from 5 to 3 also. Verify your answer by running the modified programs and checking the size of the resulting data files.

3. Create new programs similar to the examples above to write list of values with the following types: `char`, `float` and `double`.

Listing 4.18 shows an example of how to write the value of a single structure variable. The example structure contains four members corresponding to the four simple data types.

Listing 4.18: Writing the value of a structure

```
1  #include <stdio.h>
2
3  struct sampleTag {
4      char ch;
5      int i;
6      float f;
7      double d;
8  };
9
10 int main()
11 {
12     struct sampleTag s;
13     FILE *fp;
14
15     /* initialize structure members */
16     s.ch = 'A';
17     s.i = 123;
18     s.f = 4.56789;
19     s.d = 3.1415159;
20
21     /* write structure into file */
22     fp = fopen("sample_05.dat", "wb");
23     fwrite(&s, sizeof(struct sampleTag), 1, fp);
24     fclose(fp);
25     return 0;
26 }
```

► Self–Directed Learning Activity ◄

1. Encode, compile and test the program in Listing 4.18.

2. Modify the program to declare an array of structure with 5 elements of type `struct sampleTag`. Initialize the values. Thereafter, write it into a file name `"sample_06.dat"`.

4.12 Reading Data From a Binary File

Data can be read from a binary file using the `fread()` function. Its corresponding syntax is as follows:

```
size_t fread(<buffer pointer>, <size>, <count>, <file pointer>)
```

The meaning of each parameter is the same with those used in `fwrite()`.

The function returns an integer value corresponding to the number of elements read from a file. A successful `fread()` should return a value equivalent to the third parameter. There are two possible scenarios when the return value is not the same as `count` (i) a read error has occurred or (ii) the `EOF` was reached. The program has to be able to handle these cases appropriately to avoid run–time error. There is a pre–defined function called `ferror()` that can be used to determine if a read error has occurred. We will not discuss `ferror()` at this point in time.

In the following subsections, we show how to read data from the sample data files which were created as specified in Listings 4.14 to 4.18.

4.12.1 How to Read Values of Simple Data Types

We consider first how to read from an existing binary file the values of simple data types `char`, `int`, `float` and `double`. Listing 4.19 shows a simple program that reads just one character into a binary file named `"sample_01.dat"` using `fread()`. The value is stored in the local variable `ch`. This value is displayed on the screen using `fprintf()`; note that this particular step is not really required in all programs manipulating binary files.

Listing 4.20 extends the previous program by reading one `char`, one `int`, one `float` and `double` data type values from a binary file named `"sample_02.dat"`.

Listing 4.19: Reading a single character from a binary file

```
#include <stdio.h>

int main()
{
    FILE *fp;
    char ch;

    fp = fopen("sample_01.dat", "rb");
    fread(&ch, sizeof(char), 1, fp);
    fprintf(stdout, "ch = %c\n", ch);
    fclose(fp);

    return 0;
}
```

Listing 4.20: Reading values of simple data types from a binary file

```
#include <stdio.h>

int main()
{
    char ch;
    int i;
    float f;
    double d;
    FILE *fp;

    fp = fopen("sample_02.dat", "rb");

    fread(&ch, sizeof(char), 1, fp);
    fread(&i, sizeof(int), 1, fp);
    fread(&f, sizeof(float), 1, fp);
    fread(&d, sizeof(double), 1, fp);

    fprintf(stdout, "ch = %c, i = %d, f = %f, d = %lf\n",
            ch, i, f, d);

    fclose(fp);
    return 0;
}
```

► Self–Directed Learning Activity ◄

1. Encode, compile and test the program in Listings 4.19 and 4.20. Make sure that the binary files `"sample_01.dat"` and `"sample_02.dat"` exist in the same directory as the executable files.

2. Find out what will happen if the data files are not present.

3. Find out what will happen if we interchange the sequence of the first two `fread()` statements in Listing 4.20. That is, the program will read first an integer value, then it will read a character value.

4.12.2 How to Read Values of Group of Elements

We consider next how to read values for group of elements such as arrays and structures from an existing binary file.

Listing 4.21 shows how to read a list of integers from the file `"sample_03.dat"`. The sample program illustrates a technique for reading data from a file without prior knowledge regarding how many values are stored in the file. This is achieved via a `while` loop which checks the return value of `fread()`. If `fread()` returns a value of 1, then there is still at least one integer value that can be read from the file. If the `while` condition becomes false, then it means that we have reached the `EOF`.

Listing 4.21: Reading an unknown number of integer values from a binary file

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int count;
6      int value;
7      FILE *fp;
8
9      fp = fopen("sample_03.dat", "rb");
10
11     while ((count = fread(&value, sizeof(int), 1, fp)) == 1)
12        fprintf(stdout, "count = %d, value = %d\n", count, value);
13
14     fclose(fp);
15     return 0;
16 }
```

We know in advance that there are 5 integer values stored in `"sample_04.dat"`. This apriori information lets us write the codes such that the elements can be stored in an array with a known size. It will only require us to invoke `fread()` only once as shown in Listing 4.22.

Listing 4.22: Reading a known number of integer values from a binary file

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int i;
6      int A[5];
7      FILE *fp;
8
9      /* read data to binary file */
10     fp = fopen("sample_04.dat", "rb");
11     fread(A, sizeof(int), 5, fp);
12     fclose(fp);
13
14     for (i = 0; i < 5; i++)
15        fprintf(stdout, "A[%d] = %d\n", i, A[i]);
16
17     return 0;
18 }
```

Listing 4.23 shows an example of how to read the value of a single structure variable.

Listing 4.23: Reading a structure from a binary file

```
#include <stdio.h>

struct sampleTag {
    char ch;
    int i;
    float f;
    double d;
};

int main()
{
    struct sampleTag s;
    FILE *fp;

    fp = fopen("sample_05.dat", "rb");
    fread(&s, sizeof(struct sampleTag), 1, fp);

    fprintf(stdout, "ch = %d, i = %d, f = %f, d = %lf\n",
            s.ch, s.i, s.f, s.d);

    fclose(fp);
    return 0;
}
```

▶ Self–Directed Learning Activity ◀

1. Encode, compile and test the program in Listings 4.21 and 4.23. Make sure that the binary files exist in the same directory as the executable files.

2. Find out what will happen if the data files are not present.

3. In Listing 4.22, change the value of the third parameter of `fread()` from 5 to 1. What do you think will be the effect? Try changing it from 5 to 3 also. Verify your answer by running the modified programs.

4.13 Checking for the EOF

The pre–defined function `feof()` allows us to check if we have reached the `EOF`. Its syntax is given as:

```
int feof(<file pointer>)
```

It returns a non–zero value if the `EOF` was reached; otherwise, it returns a zero. This function can be used for both text and binary files.

We illustrate one way to use `feof()` in Listing 4.24. It is an alternative implementation for Listing 4.21. The program reads all the integer values from `"sample_03.dat"`. The technique requires that we read the first integer value from the file. Thereafter, we test if we have NOT yet reached the `EOF`. This is specified by the expression `!feof(fp)` in the `while` loop.. Take note of the use of the NOT logical operator before `feof()`. If it is not yet the `EOF`, we execute the body of the function which contains another call to `fread()`. The `while` loop condition will become false when `feof()` returns a non–zero value.

Which technique/implementation do you think is more efficient: the technique without `feof()` or the one with `feof()`?

Listing 4.24: Example usage of `feof()`

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int count;
6      int value;
7      FILE *fp;
8
9      fp = fopen("sample_03.dat", "rb");
10
11     count = fread(&value, sizeof(int), 1, fp);
12     while (!feof(fp)) {
13        fprintf(stdout, "count = %d, value = %d\n", count, value);
14        count = fread(&value, sizeof(int), 1, fp);
15     }
16
17     fclose(fp);
18     return 0;
19 }
```

4.14 Random File Access

4.14.1 File Position and `ftell()`

A binary file is actually organized as a group of elements with each element being one byte in size. Each byte is associated with a unique *file position*. The first byte is stored at file position 0, the second byte is at position 1, the next byte is at position 2, ..., and so on. Thus, if there are N bytes, i.e. the file size is N, the last byte is stored at file position N-1. This organization is actually the same with how the RAM is organized. It should be noted that for files, the file position N correspond to the position of the `EOF`.

In the following discussions, we will use the name `fpos` to denote file position. We can think of it as a variable whose value is from 0 to N.

Immediately after opening a file (either for read or write mode), the value of `fpos` is zero, i.e., it denotes the beginning or start of the file. A call to `fread()` or `fwrite()` will advance the file position by a certain number of bytes which is equivalent to parameter `size` multiplied with parameter `count`.

The current file position can be obtained via the return value of the pre–defined function `ftell()`.

```
long int ftell(<file pointer>)
```

We illustrate how to use `ftell()` in Listing 4.25. It is actually a modified version of the program in Listing 4.15. We just added several pairs of function calls to `ftell()` and `fprintf()`. When executed, this modified program will clearly show how the file position advances as new data values are written into the file.

Listing 4.25: Example usage of `ftell()`

```
1  #include <stdio.h>
2
3  int main()
4  {
5      char ch = 'A';
6      int i = 123;
7      float f = 4.56789;
8      double d = 3.1415159;
9
10     FILE *fp;
```

```
11      long int fpos;
12
13      fp = fopen("sample_06.dat", "wb");
14      fpos = ftell(fp);
15      fprintf(stdout, "After fopen(), fpos = %d\n", fpos);
16
17      fwrite(&ch, sizeof(char), 1, fp);
18      fpos = ftell(fp);
19      fprintf(stdout, "After writing ch, fpos = %d\n", fpos);
20
21
22      fwrite(&i, sizeof(int), 1, fp);
23      fpos = ftell(fp);
24      fprintf(stdout, "After writing i, fpos = %d\n", fpos);
25
26      fwrite(&f, sizeof(float), 1, fp);
27      fpos = ftell(fp);
28      fprintf(stdout, "After writing f, fpos = %d\n", fpos);
29
30      fwrite(&d, sizeof(double), 1, fp);
31      fpos = ftell(fp);
32      fprintf(stdout, "After writing d, fpos = %d\n", fpos);
33
34      fclose(fp);
35      return 0;
36  }
```

► Self–Directed Learning Activity ◄

1. Encode, compile and test the program in Listing 4.25. Take note of the printed values of `fpos`.

2. Modify the program in Listing 4.20 to detemine the file position after `fopen()` and `fread()`. This will allow us to verify and prove that `fread()` advances the file position by a certain number of bytes equivalent to `size` multiplied with `count` parameters.

4.14.2 File Position and `fseek()`

It is possible to reposition the file position anywhere between 0 and the `EOF` using `fseek()`. Its usage is as follows

```
int fseek(<file pointer>, <offset>, <origin>)
```

where the `offset` parameter is an integral value (measured in bytes) by which the file position will be moved relative to a specified `origin`. The new file position is computed as `origin + offset`. There are three possible position for `origin`, namely, the position of the first byte, the current file position, and the position immediately after the last byte, i.e., `EOF`. The `stdio.h` file conveniently defines these as macros which are listed in Table 4.2.

Table 4.2: Macros for `origin`

Macro	**Meaning**
`SEEK_SET`	beginning of the file, i.e., first byte
`SEEK_CUR`	current file position
`SEEK_END`	end of the file, i.e., after the last byte

Case 1: Reposition relative to start of the file. Example: the call `fseek(fp, 4, SEEK_SET)` will result into a new file position equal to 4. The new position is computed as `origin + offset` where origin is 0 (since it is the start of the file) and `offset` is 4.

Case 2: Reposition relative to current file position. Example: let us assume that `fpos` is currently 5. The call `fseek(fp, 4, SEEK_CUR)` will result into a new file position equal to 9. The new position is computed as `origin + offset` where origin is 5 (the current file position) and `offset` is 4.

Case 3: Reposition relative to `EOF`. Let us assume that the file is 17 bytes in size. The call `fseek(fp, -4, SEEK_END)` will result into a new file position equal to 13. The new position is computed as `origin + offset` where `origin` is 17 (since it is the `EOF`) and `offset` is -4. Take note that the `offset` must be negative when using `SEEK_END` as the `origin`.

Take note that `fseek()` will fail if it attempts to use a negative value for the new position. The `fseek()` function returns zero if repositioning was successful; otherwise, it returns a non–zero value indicating failure.

We illustrate how to use `fseek()` in Listing 4.26.

Listing 4.26: Example usage of `fseek()`

```
#include <stdio.h>

int main()
{
    FILE *fp;
    long int fpos;
    int offset;
    int start_fpos;    /* fpos at start */
    int current_fpos;  /* fpos at current */
    int eof_fpos;      /* fpos at EOF */

    fp = fopen("sample_06.dat", "rb");
    start_fpos = ftell(fp);

    /* Case 1: reposition relative to start of file */
    offset = 5;
    fseek(fp, offset, SEEK_SET);
    fpos = ftell(fp);
    fprintf(stdout, "offset = %d, SEEK_START at %d, fpos = %d\n",
            offset, start_fpos, fpos);

    /* Case 2: reposition relative to the current fpos */
    offset = 4;
    current_fpos = fpos;
    fseek(fp, offset, SEEK_CUR);
    fpos = ftell(fp);
    fprintf(stdout, "offset = %d, SEEK_CUR at %d, fpos = %d\n",
            offset, current_fpos, fpos);

    /* Case 3: reposition relative to EOF position */
    fseek(fp, 0, SEEK_END);
    eof_fpos = ftell(fp);

    offset = -2;  /* note: negative 2 */
    fseek(fp, offset, SEEK_END);
    fpos = ftell(fp);
    fprintf(stdout, "offset = %d, SEEK_END at %d, fpos = %d\n",
            offset, eof_fpos, fpos);
    fclose(fp);
    return 0;
}
```

► Self–Directed Learning Activity ◄

1. Encode, compile and test the program in Listing 4.26.
2. Assume a binary file containing 100 bytes. Which of the following `fseek()` call will be successful? Which will fail? If there is a failure, explain why it failed. For the items with `SEEK_CUR` assume that the current `fpos` is at 50. The best way to check if your answer is correct is to write a program, run it and check the return value of `fseek()`.
   (a) `fseek(fp, sizeof(char) * 2, SEEK_SET)`
   (b) `fseek(fp, -2 * sizeof(char), SEEK_END)`
   (c) `fseek(fp, sizeof(char)* 2, SEEK_END)`
   (d) `fseek(fp, -2 * sizeof(char), SEEK_SET)`
   (e) `fseek(fp, 50, SEEK_CUR)`
   (f) `fseek(fp, -50, SEEK_CUR)`
   (g) `fseek(fp, 51, SEEK_CUR)`
   (h) `fseek(fp, -51, SEEK_CUR)`

4.14.3 Random Access With `fseek()`

The programs that we developed so far read data from the binary files in the same sequence that they were written. It is actually possible to read the values in any sequence that we want. It is possible to access directly the first value, the last value or any value anywhere in the file without having to read any other elements. This type of access is called random access (as opposed to sequential access).

The function `fseek()` allows us to reposition `fpos` anywhere in the file starting from the first byte to the `EOF`. Once positioned, we can use `fread()` to read a set of bytes starting at the current `fpos`.

The binary file `"sample_06.dat"` which we created using a previous program (see Listing 4.25) contains one character value, one integer value, one float value and one double data type value in that sequence. We demonstrate in Listing 4.27 the concept of random access using `fseek()`. The program will print the data values in the following order: first the integer value, followed by double, then the character and finally the float value. This order is totally different from the order by which these values were written. Note that we deliberately used different `origin` values. It is actually possible to use just one common `origin`, for example, `SEEK_SET` or `SEEK_END`, and then specify the correct `offset` value.

Listing 4.27: Random access with `fseek()`

```
#include <stdio.h>

int main()
{
    char ch;
    int i;
    float f;
    double d;
    FILE *fp;

    fp = fopen("sample_02.dat", "rb");

    /* position fpos to the integer data */
    fseek(fp, 1, SEEK_SET);
    fread(&i, sizeof(int), 1, fp);
    fprintf(stdout, "i = %d\n", i);

    /* position fpos to read double data */
    fseek(fp, -8, SEEK_END);
    fread(&d, sizeof(double), 1, fp);
    fprintf(stdout, "d = %lf\n", d);

    /* position fpos to read char data */
    fseek(fp, 0, SEEK_SET);
    fread(&ch, sizeof(char), 1, fp);
    fprintf(stdout, "ch = %c\n", ch);

    /* position fpos to read float data */
    fseek(fp, 4, SEEK_CUR);
    fread(&f, sizeof(float), 1, fp);
    fprintf(stdout, "f = %f\n", f);

    fclose(fp);
    return 0;
}
```

► Self–Directed Learning Activity ◄

1. Encode, compile and test the program in Listing 4.27. What values were displayed?

2. The `offset` values that we used to position `fpos` are hardcoded numbers. This is not really a good programming practice. A better approach would

be to specify the `offset` based on the `sizeof()` operator. This will help us avoid problems when we port the source codes from one platform to another platform which may have differing sizes for their data types.

Change the original codes to their improved version as shown in the following table, and then run the program.

Original	**Improved Version**
`fseek(fp, 1, SEEK_SET)`	`fseek(fp, sizeof(char), SEEK_SET)`
`fseek(fp, -8, SEEK_END)`	`fseek(fp, -sizeof(double), SEEK_END)`
`fseek(fp, 4, SEEK_CUR)`	`fseek(fp, sizeof(int), SEEK_CUR)`

3. Modify the codes from the previous item by using only one uniform `origin`. That is, all `fseek()` should have the same `origin` unlike the original program which uses all the three possibilities. Modify the program such it uses only `SEEK_SET`. Thereafter, modify the program such that it uses only `SEEK_CUR`. The last modification should use only `SEEK_END`.

4. Create your own programs for accessing and displaying the data values stored in `"sample_06.dat"` in the sequence given below (one program per sequence). Try to create several variations of the programs, i.e. a variation when the `origin` is uniform for all `fseek()` and another variation where different `origin` values are used.

   (a) `char` value, `float` value, `double` value, `int` value

   (b) `float` value, `int` value, `char` value, `double` value

   (c) `double` value, `char` value, `float` value, `int` value

   (d) `double` value, `int` value, `float` value, `char` value

   (e) `double` value, `float` value, `int` value, `char` value

We present another example of random file access in Listing 4.28. The program first creates a binary file which will contain 10 integer values. Thereafter, the user is prompted to input an `"index"` corresponding to the integer that he/she would like to read from the file. The term `"index"` in this case takes on the same meaning as in the discussion of memory addressing way back in Chapter 1.

There are two key concepts that are worth learning from this particular example program, specifically:

1. Files can be opened, accessed, and closed in functions other than `main()`
2. If the file contains values of homogeneous data type (that is, all values have the same data type, for example an array), the `offset` parameter in the `fseek()` function can be specified as the `sizeof` the data type multiplied by the index of the element with an `origin` set to `SEEK_SET`.

Listing 4.28: Random access (second example)

```
1  #include <stdio.h>
2  #define FILENAME "sample_07.dat"
3
4  void CreateFile(char filename[])
5  {
6      FILE *fp;
7      int value;
8      int i;
9
10     fp = fopen(FILENAME, "wb");
11     for (i = 0; i < 10; i++) {
12         value = i * 100;
13         fwrite(&value, sizeof(int), 1, fp);
14     }
15
16     fclose(fp);
17 }
18
19 int main()
20 {
21     FILE *fp;
22     int value;
23     int index;
24     int count;
25     int retval;
26
```

```
27      /* first, we create the file */
28      CreateFile(FILENAME);
29
30      /* next, we re-open the file */
31      fp = fopen(FILENAME, "rb");
32
33      while (1) {
34         printf("Input index of the element you want to read.");
35         printf("  Enter -1 to exit: ");
36         scanf("%d", &index);
37         if (index == -1)
38            break;
39         else {
40            /* reposition fpos, then read data */
41            /* retval is zero for successful fseek,
42               otherwise it's negative */
43            retval = fseek(fp, sizeof(int) * index, SEEK_SET);
44            fprintf(stdout, "retval = %d, ", retval);
45
46            /* count should be 1 for successful fread */
47            count = fread(&value, sizeof(int), 1, fp);
48            fprintf(stdout, "count = %d, Data at index %d = %d\n\n",
49                    count, index, value);
50         }
51      }
52      fclose(fp);
53      return 0;
54  }
```

▶ Self–Directed Learning Activity ◀

1. Encode, compile and test the program in Listing 4.28. What is the size of the file? What are the contents of the binary file? Write down the value and the corresponding position for all the values stored in the file.

2. Run the program by supplying `index` values starting from 0, then 1, then 2, and so on until 9. Are the output consistent with the values you wrote in the previous item? Take note of the values of `retval` and `tcount` as well.

3. The index values from 0 to 9 are valid. Other values are actually incorrect. Try running the program with an `index` of -5. What is the return value of `fseek()` as stored in variable `retval`?

4. Next try an index value of 12. It will reposition `fpos` way past the `EOF`. What is the value of `retval`? What is the return value of `fread()` as stored in variable `count`?

4.15 Reading and Writing on the Same File

In applications such as Notepad, we can open an existing file, view the contents, edit the contents and save it. Essentially, this means that the file was opened for reading, but later on it performed a write operation as well.

Reading and writing on the same file (though not at exactly the same time) is actually possible using what we call "mixed" modes. These are:

- `"rb+"`or `"r+b"` – open the file for both read and write, note that file must exist.
- `"wb+"` or `"w+b"` – open the file for both read and write. If the file does not exist, a new one will be automatically created. If the file exists, the original contents will be erased and overwritten with a new one.

The presence of the `"+"` allows mixed mode access, i.e., both reading and writing. We will limit our discussion to `"rb+"` mode since this is the mode that will be most prevalent in actual practice.

Listing 4.29 shows an example of mixed mode access. The program opens the file `"sample_07.dat"` with a mode of `"rb+"`. Note that `"sample_07.dat"` was created by the program in the previous section. It then sets the file position via `fseek()` and then reads the data stored in the said position using `fread()`. The value is assigned to local variable `n`. The value of `n` is displayed on the screen using `fprintf()`. Thereafter, value of `n` is decremented by 1. The programs sets the file position again via `fseek()` and then writes the new value of `n` on the same position occupied by the original value of `n` (prior to decrement). The program is a simple illustration of a technique for editing/changing the contents of an existing file.

Listing 4.29: Mixed mode access

```
1  #include <stdio.h>
2  int main()
3  {
4      FILE *fp;
5      int n;
6
7      fp = fopen("sample_07.dat", "rb+");
8
9      fseek(fp, sizeof(int) * 5, SEEK_SET);
10     fread(&n, sizeof(int), 1, fp);
11
12     n = n - 1;
13     fseek(fp, sizeof(int) * 5, SEEK_SET);
14     fwrite(&n, sizeof(int), 1, fp);
15
16     fprintf(stdout, "n = %d\n", n);
17
18     fclose(fp);
19     return 0;
20 }
```

► Self–Directed Learning Activity ◄

1. Encode, compile and test the program in Listing 4.29. What are the contents of the binary file prior to running the program? after running the program?

2. Run the program several times. What do you notice? Try to give an explanation of what the program does everytime you run it.

3. Modify the program such that:

   (a) it will always increment the first data value by 1.

   (b) it will always multiply the last data value by 2.

4.15.1 The `fflush()` function

The `fflush()` function is used to force data to be written onto the output file. The function prototype is as follows:

```
int fflush(<file pointer>)
```

It returns 0 for if the operation is successful, otherwise it returns `EOF`. Use `fflush()` after `fwrite()` in case there are multiple `fread()` and `fwrite()` on the same file inside some loop.

Listing 4.30 shows another example of mixed mode access with `fflush()`. The program gives the user two options. The first option creates a file that will contain the characters `"Hello World!"`. The mode used here is `"wb"`. The second option updates the contents of the file by changing all lower case letters to their upper case equivalents using the predefined function `toupper()`.[8] The mode used here is mixed mode as denoted by `"rb+"`.

Listing 4.30: Mixed mode access (second example)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <ctype.h>
5
6  #define FILENAME "sample_08.dat"
7
8  void CreateFile(char filename[])
9  {
10      FILE *fp;
11      char message[20] = "Hello World!";
12
13      fp = fopen(filename, "wb");
14      fwrite(message, sizeof(char), strlen(message), fp);
15      fclose(fp);
16 }
17
18 void UpdateFile(char filename[])
19 {
20      FILE *fp;
```

[8] Do some information search to learn more about the `toupper()` and `tolower()` functions. Their corresponding header file is `ctype.h`.

```
21      char ch;
22
23      fp = fopen(filename, "rb+");
24
25      while ((fread(&ch, sizeof(char), 1, fp)) == 1) {
26          if (ch >= 'a' && ch <= 'z') {
27              ch = toupper(ch); /* change to upper case */
28              fseek(fp, -1, SEEK_CUR);
29              fwrite(&ch, sizeof(char), 1, fp);
30              fflush(fp);
31          }
32      }
33      fclose(fp);
34  }
35
36  int main()
37  {   int choice;
38
39      printf("Input [1] Create file, [2] Update file: ");
40      scanf("%d", &choice);
41      if (choice == 1)
42         CreateFile(FILENAME);
43      else if (choice == 2)
44              UpdateFile(FILENAME);
45      else fprintf(stderr, "Invalid choice.");
46
47      return 0;
48  }
```

▶ Self–Directed Learning Activity ◀

1. Encode, compile and test the program in Listing 4.30. Run the program twice. Choose option 1 during the first time. Make sure that you examine the contents of the file produced. Thereafter, run the program the second time and choose option 2. Examine again the contents of the file. What do you notice? Can you explain what actually happened?

2. Write your own program for reading the contents of a file containing characters. If the file is an upper case character, replace it with a corresponding lower case. Note: there is a pre–defined function `tolower()` that allows conversion from upper to lower case.

4.16 Chapter Summary

The key ideas presented in this chapter are summarized below:

- A file is a group of elements. We classify the files into two: text files and binary files.
- We learned how to create our own files, and how to read data from existing files using the following pre–defined functions:
  - `fopen()` to open the file
  - `fclose()` to close the file
  - `fprintf()` for formatted output; this function is used to write data to a text file
  - `fscanf()` for formatted input; this function is used to read data from a tex file
  - `fwrite()` to write data to a binary file
  - `fread()` to read data from a binary file
  - `feof()` to check the `EOF` was reached
  - `ftell()` to determine the current file position `fpos`
  - `fseek()` to reposition `fpos`
  - `fflush()` to force data to be written onto the output file.
- Mode `"w"` is for writing to a text file, `"a"` is for appending to a text file, and `"r"` is for reading from a text file.
- Mode `"wb"` is for writing to a binary file, `"rb"` for reading from a binary file.
- Mode `"w+b"` or `"wb+"` and `"rb+"` or `"r+b"` is for mixed mode, i.e., reading and writing to the same file.
- VERY IMPORTANT NOTE: We should have information regarding the data format of a binary file in order for us to be able to read and interpret its contents correctly.

Problems for Chapter 4

For problems 4.1 and 4.2 assume a text file named `"essay.txt"`.

Problem 4.1. Write a program that will read the contents of `"essay.txt"`. The output of the program are the values of the following:

- How many letters are in the file (lower case and upper case)
- How many lower case letters are in the file
- How many upper case letters are in the file
- How many lines are present in the file (note: a line ends with a newline character)
- How many are lower case vowels
- How many are upper case vowels
- How many are lower case consonants
- How many are upper case consonants

Problem 4.2. Write a program that will read the contents of `"essay.txt"` and copy all the lower case letters into a file named `"lower.txt"` and all upper case letters into a file named `"upper.txt"`.

Problem 4.3. Assume that there are two text files. Write a program that will determine whether these two text files are identical or not (i.e., the contents are the same).

For problems 4.4 and 4.5 assume a list of integers stored in a text file named `"integers.txt"`. *We do not know in advance how many integers are present.*

Problem 4.4. Write a program that will read the contents of `"integers.txt"`. The output of the program are the values of the following:

- The smallest integer value (minimum value)
- The largest integer value (maximum value)
- The number of integers read from the file
- The sum of all the integers

- The average of all the integers

Problem 4.5. Write a program that will read the contents of `"integers.txt"`. The output of the program is the frequency count of the following:

- How many integers are greater than or equal to 0
- How many integers are negative
- How many integers are odd numbers
- How many integers are even numbers

Problem 4.6. Assume two lists of integers both in ascending order. The first list was stored in a file named `list1.txt`, the second in `list2.txt`. We do not know how many integers are stored in each file. Write a program that will read data values from these two files and store them also in ascending order in a new file named `list3.txt`.

For the next two problems, assume a list of integers stored in a binary file named `"integers.dat"`. *We do not know in advance how many integers are present.*

Problem 4.7. Write a program that will read the contents of `"integers.dat"`. The output of the program are the values of the following:

- The smallest integer value (minimum value)
- The largest integer value (maximum value)
- The number of integers read from the file
- The sum of all the integers
- The average of all the integers

Problem 4.8. Write a program that will read the contents of `"integers.dat"`. The output of the program is the frequency count of the following:

- How many integers are greater than or equal to 0
- How many integers are negative
- How many integers are odd numbers

- How many integers are even numbers

Problem 4.9. Assume two lists of integers both in ascending order. The first list was stored in binary files named `list1.dat`, the second in `list2.dat`. We do not know how many integers are stored in each file. Write a program that will read data values from these two files and store them also in ascending order in a new binary file named `list3.dat`.

For the following problems, assume the following declarations:

```
struct EmployeeTag
{
     char name[40];
     int ID_number;
     float salary;
};

typedef struct EmployeeTag EmployeeType;
```

Problem 4.10 Write a program that will declare an array of 10 structures. Each structure is of type `EmployeeType`. Initialize the contents of the array. Finally, write the contents of the array onto a binary file named `"employee.dat"`.

Problem 4.11 Write a program that will read employee structure one at a time from the file `"employee.dat"` created in the previous problem. The value of each structure should be displayed on the standard output device.

Problem 4.12 Write a program that will read employee structure in REVERSED ORDER one at a time from the file `"employee.dat"` created in the previous problem. The value of each structure should be displayed on the standard output device.

Problem 4.13 Write a program that will read ALL employee structures from the file `"employee.dat"` created in the previous problem. Data read from the file should be stored into an array of employee structure. Thereafter, the value of each array element should be displayed on the standard output device.

Problem 4.14 Write a program that will ask the user to input an integer value representing an ID number. Determine if there is an employee structure stored in the file `"employee.dat"` with the same ID number. If there is such a structure, display the contents of the employee structure; otherwise, display a message indicating "There is no employee with such ID number". Note that this is essentially a search problem.

Problem 4.15 Write a program that will open the file using `"rb+"` mode. Update the employee structures stored in the file `"employee.dat"` by giving a salary increase of 10 percent to all employees. For example, if an employee's salary is 10,000.00 then the updated value of the `salary` member should be 10,100.00. Make sure to call `fflush()` right after `fwrite()`.

Problem 4.16 Write a program that will open the file using `"rb+"` mode. Sort the employee structures in ascending order based on ID_number member. DO NOT use an array for this; work directly with the binary file. Refer back to the straight selection sorting algorithm discussed in the chapter on arrays. Apply the algorithm on the binary file. Make sure to call `fflush()` right after `fwrite()`.