Appendix C

In this section of the book, we will look at some additional recipes that were outside the scope of `Chapter 6`, *File Handling*:

- Creating a sequential file and entering some text into it
- Reading content from a sequential file and displaying it onscreen
- Creating a random file and entering some data into it
- Reading content from a random file and displaying it onscreen
- Decrypting the contents of an encrypted file

Creating a sequential file and entering some data into it

In this recipe, we will be creating a sequential file and the user can enter any desired number of lines into it. The name of the file to be created is passed through command line arguments. You can enter as many lines in the file as you want and, when finished, you have to type `stop`, followed by pressing the *Enter* key.

How to do it…

1. Open a sequential file in write-only mode and point to it with a file pointer:

```
fp = fopen (argv[1], "w");
```

2. Enter the content for the file when prompted:

```
printf("Enter content for the file\n");
gets(str);
```

3. Enter `stop` when you are done entering the file content:

```
while(strcmp(str, "stop") !=0)
```

4. If the string you entered is not `stop`, the string is written into the file:

```
fputs(str,fp);
```

5. Close the file pointer to release all the resources allocated to the file:

```
fclose(fp);
```

The `createtextfile.c` program for creating a sequential file is as follows:

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

void main (int argc, char* argv[])
{
   char str[255];
   FILE *fp;

  fp = fopen (argv[1], "w");
  if (fp == NULL) {
     perror ("An error occurred in creating the file\n");
    exit(1);
  }
  printf("Enter content for the file\n");
  gets(str);
  while(strcmp(str, "stop") !=0){
      fputs(str,fp);
      gets(str);
  }
  fclose(fp);
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

We define a file pointer by the name `fp`. We will open the sequential file, whose name is supplied through the command-line argument, in write-only mode and set the `fp` file pointer to point at it. If the file cannot be opened in write-only mode, it might be because there are not enough permissions or disk space constraints. An error message will be displayed and the program will terminate.

If the file opens successfully in write-only mode, you will be prompted to enter the contents for the file. All the text you enter will be assigned to the `str` string variable, which is then written into the file. You should enter `stop` when you have completed entering the content into the file. Finally, we will close the file pointer.

Let's use GCC to compile the `createtextfile.c` program, as shown in the following statement:

```
D:\CBook>gcc createtextfile.c –o createtextfile
```

If you get no errors or warnings, it means that the `createtextfile.c` program has been compiled into an executable file, `createtextfile.exe`. Let's run this executable file:

```
D:\CBook>createtextfile textfile.txt
Enter content for the file
I am trying to create a sequential file. it is through C programming.
It is very hot today
I have a cat.  do you like animals?    It might rain
Thank you. bye
stop
```

Voila! We've successfully created a sequential file and entered data in it.

Now let's move on to the next recipe!

Reading content from a sequential file and displaying it onscreen

In this recipe, we assume that a sequential file already exists, so we will be reading the contents of that file and displaying it onscreen. The name of the file whose contents we want to read will be supplied through command-line arguments.

How to do it…

1. Open the sequential file in read-only mode and set the `fp` file pointer to point at it:

```
fp = fopen (argv [1],"r");
```

2. The program terminates if the file cannot be opened in read-only mode:

```
if (fp == NULL) {
    printf("%s file does not exist\n", argv[1]);
    exit(1);
}
```

3. Set a `while` loop to execute until the end of the file is reached:

```
while (!feof(fp))
```

4. Within the `while` loop, one line at a time is read from the file and is displayed onscreen:

```
fgets(buffer, BUFFSIZE, fp);
puts(buffer);
```

5. Close the file pointer to release all the resources allocated to the file:

```
fclose(fp);
```

The `readtextfile.c` program for reading a sequential file is as follows:

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

#define BUFFSIZE 255

void main (int argc, char* argv[])
{
   FILE *fp;
   char buffer[BUFFSIZE];

  fp = fopen (argv [1],"r");
  if (fp == NULL) {
    printf("%s file does not exist\n", argv[1]);
    exit(1);
  }
  while (!feof(fp))
  {
    fgets(buffer, BUFFSIZE, fp);
    puts(buffer);
  }
  fclose(fp);
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

We will define a file pointer is defined by the name of `fp` and a string by the name of `buffer` of size 255. We will open the sequential file, whose name is supplied through the command-line argument, in read-only mode and set the `fp` file pointer to point at it. If the file cannot be opened in read-only mode because the file does not exist or because of a lack of enough permissions, an error message will be displayed and the program will terminate.

If the file opens successfully in read-only mode, set a `while` loop to execute until the end of the file is reached. Use the `fgets` function to read one line at a time from the file; the line read from the file is assigned to the `buffer` string. Then, display the content in the `buffer` string onscreen. When the end of the file has been reached, the `while` loop will terminate and the file pointer is closed to release all the resources allocated to the file.

Let's use GCC to compile the `readtextfile.c` program, as shown in the following statement:

```
D:\CBook>gcc readtextfile.c –o readtextfile
```

If you get no errors or warnings, this means that the `readtextfile.c` program has been compiled into an executable file, `readtextfile.exe`. Assuming the file whose content we want to read is textfile.txt, let's run the executable file, `readtextfile.exe`:

```
D:\CBook>readtextfile textfile.txt
I am trying to create a sequential file. it is through C programming.
It is very hot today. I have a cat.  do you like animals?    It might
rain. Thank you. bye
```

Voila! We've successfully read the content from our sequential file and displayed it onscreen.

Now, let's move on to the next recipe!

Creating a random file and entering some data into it

In this recipe, we will be creating a random file and we will enter some lines of text into it. The random files are structured, and the content in the random file is written via structures. The benefit of creating a file using structures is that we can compute

the location of any structure directly and can access any content in the file randomly. The name of the file to be created is passed through command-line arguments.

How to do it…

The following are the steps to create a random file and enter a few lines of text in it. You can enter any number of lines as desired; simply type `stop`, followed by the *Enter* key, when you are done:

1. Define a structure consisting of a string member:

```
struct data{
    char str[ 255 ];
};
```

2. Open a random file in write-only mode and point to it with a file pointer:

```
fp = fopen (argv[1], "wb");
```

3. The program will terminate if the file cannot be opened in write-only mode:

```
if (fp == NULL) {
    perror ("An error occurred in creating the file\n");
    exit(1);
}
```

4. Enter the file contents when prompted and store it into the structure members:

```
printf("Enter file content:\n");
gets(line.str);
```

5. If the text entered is not `stop`, the structure containing the text is written into the file:

```
while(strcmp(line.str, "stop") !=0){
    fwrite( &line, sizeof(struct data), 1, fp );
```

6. Steps 4 and 5 are repeated until you enter `stop`.
7. When you enter `stop`, the file pointed at by the file pointer is closed to release the resources allocated to the file:

```
fclose(fp);
```

The `createrandomfile.c` program for creating a random file is as follows:

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

struct data{
    char str[ 255 ];
};

void main (int argc, char* argv[])
{
    FILE *fp;
    struct data line;
    fp = fopen (argv[1], "wb");
    if (fp == NULL) {
        perror ("An error occurred in creating the file\n");
        exit(1);
    }
    printf("Enter file content:\n");
    gets(line.str);
    while(strcmp(line.str, "stop") !=0){
        fwrite( &line, sizeof(struct data), 1, fp );
        gets(line.str);
    }
    fclose(fp);
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

Let's start by defining a structure by the name `data` consisting of a member `str`, which is a string variable of size 255. Then we will define a file pointer by the name `fp`, followed by a variable line as a type data structure, so the line becomes a structure with a member called `str`. We will open a random file, whose name is supplied through a command-line argument, in write-only mode and set the `fp` file pointer to point at it. If the file cannot be opened in write-only mode for any reason, an error message will be displayed and the program will terminate.

You will be prompted to enter the file contents. The text you enter will be assigned to the `str` member of the line structure. Because you are supposed to enter `stop` to indicate that you have finished entering data in the file, the text you entered will be compared with the `stop` string. If the text entered is not `stop`, it is written into the file pointed at by the `fp` file pointer.

Because it is a random file, the text is written into the file through the structure line. The `fwrite` function writes the number of bytes equal to the size of the structure line into the file pointed at by the `fp` pointer at its current position. The text in the `str` member of the line structure is written into the file. When the text entered by the user is `stop`, the file pointed at by file pointer, the `fp` file pointer is closed.

Let's use GCC to compile the `createrandomfile.c` program, as shown in the following statement:

```
D:\CBook>gcc createrandomfile.c –o createrandomfile
```

If you get no errors or warnings, this means that the `createrandomfile.c` program has been compiled into an executable file, `createrandomfile.exe`. Assuming that we want to create a random file with the name `random.data`, let's run the executable file, `createrandomfile.exe`:

```
D:\CBook>createrandomfile random.data
Enter file content:
This is a random file. I am checking if the code is working
perfectly well. Random file helps in fast accessing of
desired data. Also you can access any content in any order.
stop
```

Voila! We've successfully created a random file and entered some data in it.
Now let's move on to the next recipe!

Reading content from a random file and displaying it onscreen

In this recipe, we will be reading the contents of a random file and will be displaying it on screen. Because the content in the random file comprises records, where the size of the record is already known, any record from the random file can be picked up randomly; hence, this type of file gets the name of *random file*. To access the *n*th record from a random file, we don't have to read all the *n*-1 records first as we would do in a sequential file. We can compute the location of that record and can access it directly. The name of the file to be read is passed through command-line arguments.

How to do it…

1. Define a structure consisting of a string member:

```
struct data{
    char str[ 255 ];
};
```

2. Open a random file in read-only mode and point at it with a file pointer:

```
fp = fopen (argv[1], "rb");
```

3. The program will terminate if the file cannot be opened in read-only mode:

```
if (fp == NULL) {
    perror ("An error occurred in opening the file\n");
    exit(1);
}
```

4. Find the total number of bytes in the file. Divide the retrieved total number of bytes in the file by the size of one record to get the total number of records in the file:

```
fseek(fp, 0L, SEEK_END);
n = ftell(fp);
nol=n/sizeof(struct data);
```

5. Use a `for` loop to read one record at a time from the file:

```
for (i=1;i<=nol;i++)
fread(&line,sizeof(struct data),1,fp);
```

6. The content read from the random file is via the structure defined in step 1. Display the contents of the file by displaying the file content assigned to structure members:

```
puts(line.str);
```

7. The end of the file is reached when the `for` loop has finished. Close the file pointer to release the resources allocated to the file:

```
fclose(fp);
```

The `readrandomfile.c` program for reading the content from a random file is as follows:

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

struct data{
    char str[ 255 ];
};

void main (int argc, char* argv[])
{
    FILE *fp;
    struct data line;
    int n,nol,i;
    fp = fopen (argv[1], "rb");
    if (fp == NULL) {
        perror ("An error occurred in opening the file\n");
        exit(1);
    }
    fseek(fp, 0L, SEEK_END);
    n = ftell(fp);
    nol=n/sizeof(struct data);
    rewind(fp);
    printf("The content in file is :\n");
    for (i=1;i<=nol;i++)
    {
        fread(&line,sizeof(struct data),1,fp);
        puts(line.str);
    }
    fclose(fp);
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

We will define a structure by the name, `data`, consisting of a member, `str`, which is a string variable of size 255. Then, a file pointer is defined by the name of `fp` and a variable line by the type data structure, so the line becomes a structure with a member called `str`. We will open a random file, whose name is supplied through the command-line argument, in read-only mode and set the `fp` file pointer to point at it. If the file cannot be opened in read-only mode for any reason, an error message will be displayed and the program will terminate. The file error can occur if any non-existing file is referred to, or if the file does not have enough permissions. If the file opens in read-only mode successfully, the next step is to find the total count of the number of records in the file. For this, the following formula is applied:

Total number of bytes in the file/size of one record

To find the total number of bytes in the file, first move the file pointer to the end of the file by invoking the `fseek` function. Thereafter, using the `ftell` function, retrieve the total number of bytes consumed by the file. We will then divide the total number of bytes in the file by the size of one record to determine the total count of records in the file.

Now, we are ready to read one record at a time from the file, and to do so, we will move the file pointer to the beginning of the file. We will set a `for` loop to execute the same amount of times as the number of records in the file. Within the `for` loop, we will invoke the `fread` function to read one record at a time from the file. The text read from the file is assigned to the `str` member of the line structure. The content in the `str` member of the line structure is displayed onscreen. When the `for` loop terminates, the file pointed to by the `fp` file pointer is closed to release the resources allocated to the file.

Let's use GCC to compile the `readrandomfile.c` program , as shown in the following statement:

```
D:\CBook>gcc readrandomfile.c –o readrandomfile
```

If you get no errors or warnings, this means that the `readrandomfile.c` program has been compiled into an executable file, `readrandomfile.exe`. Assuming that we want to create a random file with the name `random.data`, let's run the executable file, `readrandomfile.exe`:

```
D:\CBook>readrandomfile random.data
The content in file is :
```

```
This is a random file. I am checking if the code is working
perfectly well. Random file helps in fast accessing of
desired data. Also you can access any content in any order.
```

Voila! We've successfully read the content from a random file and displayed it onscreen.

Now let's move on to the next recipe!

Decrypting the contents of an encrypted file

In this recipe, we will be reading an encrypted file. We will be decrypting its content and writing the decrypted contents into another sequential file. Both filenames, the encrypted one and the one with which we will save the decrypted version, are supplied through command-line arguments.

How to do it…

Two files are used in this program. One is opened in read-only mode and the other is opened in write-only mode. The contents of one file is read and decrypted, and the decrypted content is stored in another file. The following are the steps to decrypt an existing encrypted file and save the decrypted version into another file:

1. Open two files, one in read-only and the other in write-only mode:

```
fp = fopen (argv [1],"r");
fq = fopen (argv[2], "w");
```

2. If either of the files cannot be opened in their respective modes, the program will terminate after displaying an error message:

```
if (fp == NULL) {
    printf("%s file does not exist\n", argv[1]);
    exit(1);
}
if (fq == NULL) {
    perror ("An error occurred in creating the file\n");
    exit(1);
}
```

3. Set a `while` loop to execute. It will read one line at a time from the file to be read:

```
while (!feof(fp))
fgets(buffer, BUFFSIZE, fp);
```

4. The length of the line that is read from the file is computed:

```
n=strlen(buffer);
```

5. Set a `for` loop to execute. This will access all the characters of the line one by one:

```
for(i=0;i<n;i++)
```

6. Add the value of `45` to the ASCII value of each character to decrypt it. I assume that ASCII value `45` was deducted from each character to encrypt the file:

```
buffer[i]=buffer[i]+45;
```

7. The decrypted line is written into the second file:

```
fputs(buffer,fq);
```

8. When the `while` loop has finished (read, which is the file being decrypted), both the file pointers are closed to release the resources allocated to them:

```
fclose (fp);
fclose (fq);
```

The `decryptfile.c` program for decrypting an encrypted file is as follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BUFFSIZE 255
void main (int argc, char* argv[])
{
    FILE *fp,*fq;
    int  i,n;
    char buffer[BUFFSIZE];
    fp = fopen (argv [1],"r");
    if (fp == NULL) {
        printf("%s file does not exist\n", argv[1]);
        exit(1);
    }
    fq = fopen (argv[2], "w");                    if (fq == NULL) {
        perror ("An error occurred in creating the file\n");
        exit(1);
    }
    while (!feof(fp))
    {
        fgets(buffer, BUFFSIZE, fp);
```

```
            n=strlen(buffer);
            for(i=0;i<n;i++)
                buffer[i]=buffer[i]+45;
            fputs(buffer,fq);
        }
        fclose (fp);
        fclose (fq);
    }
```

Now, let's go behind the scenes to understand the code better.

How it works...

We will define two file pointers, `fp` and `fq`. We will open the first file that is supplied through the command-line argument in read-only mode, and the second file in write-only mode. If the files cannot be opened in the read-only mode and write-only mode, respectively, an error message will be displayed and the program will terminate. The file that opens in read-only mode is pointed at by the `fp` file pointer and the file that opens in write-only mode is pointed at by the file `fq` file pointer.

We will set a `while` loop to execute, which will read all the lines from the file pointed at by the `fp` pointer one by one. The `while` loop will continue to execute until the end of the file pointed at by `fp` is reached.

.Within the `while` loop, a line is read and is assigned to the `buffer` string variable. The length of the line is computed. We will then set a `for` loop is set to execute up until the end of the line; that is, each character of the line is accessed. We will add the value `45` to the ASCII value of each character to encrypt it. Thereafter, we will write the decrypted line into the second file. The file is pointed at by the `fq` file pointer. When the file from which the content was read is finished with, both the file pointers are closed to release the resources allocated to both the files.

Let's use GCC to compile the `decryptfile.c` program, as shown in the following statement:

```
D:\CBook>gcc decryptfile.c -o decryptfile
```

Assume the encrypted file is named `encrypted.txt`. Let's see the encrypted text in this file:

```
D:\CBook>type encrypted.txt
≤4@≤GEL<A:≤GB≤6E84G8≤4≤F8DH8AG<4?≤9<?8≤<G≤<F≤G;EBH:;≤≤CEB:E4@@<A:≤≤≤G≤<
F≤I8EL≤;BG≤GB74L≤;4I8≤4≤64G≤≤7B≤LBH≤?<>8≤4A<@4?F≤≤≤≤G≤@<:;G≤E4<A';4A>≤L
BH≤5L8
```

The preceding command is executed in Windows' Command Prompt.

If you get no errors or warnings while compiling the file, this means that the `decryptfile.c` program has been compiled into an executable file, `decryptfile.exe`. Assume that an encrypted file by the name of `encrypted.txt` already exists and you want to decrypt it into another file, `originalfile.txt`. So, let's run the executable, `decryptfile.exe`, to decrypt the `encrypted.txt` file:

```
D:\CBook>decryptfile encrypted.txt originalfile.txt
```

Let's see the contents of `orignalfile.txt` to see whether it contains the decrypted version of the file:

```
D:\CBook>type originalfile.txt
I am trying to create a sequential file. it is through C programming.
It is very hot today. I have a cat.  do you like animals?    It might
rain. Thank you. bye
```

Voila! You can see that `originalfile.txt` contains the decrypted file.