CHAPTER 12

Working with Files

If your computer could only ever process data stored within the main memory of the machine, the scope and variety of applications that you could deal with would be severely limited. Virtually all serious business applications require more data than would fit into main memory and depend on the ability to process data that's stored on an external device, such as a fixed disk drive. In this chapter, you'll explore how you can process data stored in files on an external device.

C provides a range of functions in the header file `<stdio.h>` for writing to and reading from external devices. The external device you would use for storing and retrieving data is typically a fixed disk drive, but not exclusively. Because, consistent with the philosophy of the C language, the library facilities that you'll use for working with files are device-independent, so they apply to virtually any external storage device. However, I'll assume in the examples in this chapter that we are dealing with disk files.

In this chapter you'll learn the following:

- What a file is in C
- How files are processed
- How to write and read formatted files and binary files
- How to retrieve data from a file by direct random access to the information
- How to use temporary work files in a program
- How to update binary files
- How to write a file viewer program

The Concept of a File

With all the examples you've seen up to now, any data that the user enters when the program is executed is lost once the program finishes running. At the moment, if the user wants to run the program with the same data, he or she must enter it again each time. There are a lot of occasions when this is not only inconvenient, but also makes the programming task impossible.

If you want to maintain a directory of names, addresses, and telephone numbers, for instance, a program in which you have to enter all the names, addresses, and telephone numbers each time you run it is worse than useless! The answer is to store data on permanent storage that continues to be maintained after your computer is switched off. As I'm sure you know, this storage is called a **file**, and a file is usually stored on a hard disk.

You're probably familiar with the basic mechanics of how a disk works. If so, this can help you recognize when a particular approach to file usage is efficient and when it isn't. On the other hand, if you know nothing about disk file mechanics, don't worry at this point. There's nothing in the concept of file processing in C that depends on any knowledge of physical storage devices.

A file is essentially a serial sequence of bytes, as illustrated in Figure 12-1.

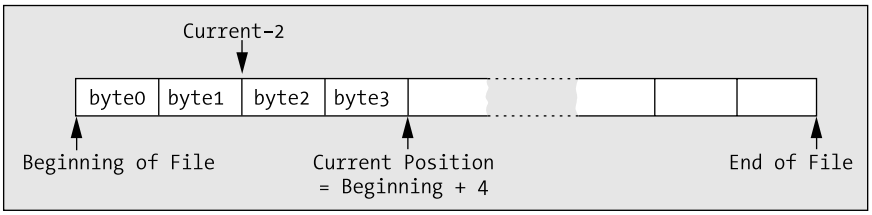

Figure 12-1. *Structure of a file*

Positions in a File

A file has a beginning and an end, and it has a **current position**, typically defined as so many bytes from the beginning, as Figure 12-1 illustrates. The current position is where any file action (a read from the file or a write to the file) will take place. You can move the current position to any other point in the file. A new current position can be specified as an offset from the beginning of the file or, in some circumstances, as a positive or negative offset from the previous current position.

File Streams

The C library provides functions for reading and writing to or from data streams. A **stream** is an abstract representation of any external source or destination for data, so the keyboard, the command line on your display, and files on disk are all examples of streams. You therefore use the same input/output functions for reading and writing any external device that is mapped to a stream.

There are two ways of writing data to a stream that is a disk file. Firstly, you can write a file as a **text file**, in which case data is written as a characters organized as lines, where each line is terminated by a newline character. Obviously, binary data such as values of type `int` or type `double` have to be converted to characters to allow them to be written to a text file, and you've already seen how this formatting is done with the `printf()` function. Secondly you can write a file as a **binary file**. Data that is written to a binary file is always written as a series of bytes, exactly as it appears in memory, so a value of type `double` for example would be written as the 8 bytes that appear in memory.

Of course, you can write any data you like to a file, but once a file has been written, it just consists of a series of bytes on disk. Regardless of whether you write a file as a binary file or as a text file, it ultimately ends up as just a series of bytes, whatever the data is. This means that when the file is read, the program must know what sort of data the file represents. You've seen many times now that exactly what a series of bytes represents is dependent upon how you interpret it. A sequence of 12 bytes in a binary file could be 12 characters, 12 8-bit signed integers, 12 8-bit unsigned integers, 6 16-bit signed integers, a 32-bit integer followed by an 8-byte floating-point value, and so on. All of these will be more or less valid interpretations of the data, so it's important that a program that is reading a file has the correct assumptions about how it was written.

Accessing Files

The files that are resident on your disk drive each have a name, and the rules for naming files will be determined by your operating system. When you write a program to process a file, it would not be particularly convenient if the program would only work with a specific file with a particular name. If it did, you would need to produce a different program for each file you might want to process. For this reason, when you process a file in C, your program references a file through a **file pointer**. A file

pointer is an abstract pointer that is associated with a particular file when the program is run so that the program can work with different files on different occasions. A file pointer points to a `struct` that represents a stream. In the examples in this chapter, I'll use Microsoft Windows file names. If you're using a different operating system environment, such as UNIX, you'll need to adjust the names of the files appropriately.

If you want to use several files simultaneously in a program, you need a separate file pointer for each file, although as soon as you've finished using one file, you can associate the file pointer you were using with another file. So if you need to process several files, but you'll be working with them one at a time, you can do it with one file pointer.

Opening a File

You associate a specific external file name with an internal file pointer variable through a process referred to as **opening a file**. You open a file by calling the standard library function `fopen()`, which returns the file pointer for a specific external file. The function `fopen()` is defined in `<stdio.h>`, and it has this prototype:

```
FILE *fopen(char *name, char *mode);
```

The first argument to the function is a pointer to a string that is the name of the external file that you want to process. You can specify the name explicitly as an argument, or you can use an array, or a variable of type pointer to `char` that contains the address of the character string that defines the file name. You would typically obtain the file name through some external means, such as from the command line when the program is started, or you could arrange to read it in from the keyboard. Of course, you can also define a file name as a constant at the beginning of a program when the program always works with the same file.

The second argument to the `fopen()`function is a character string called the **file mode** that specifies what you want to do with the file. As you'll see, this spans a whole range of possibilities, but for the moment I'll introduce just three file modes (which nonetheless comprise the basic set of operations on a file). Table 12-1 lists these three file modes.

Table 12-1. *File Modes*

Mode	Description
`"w"`	Open a text file for **w**rite operations. If the file exists, its current contents are discarded.
`"a"`	Open a text file for **a**ppend operations. All writes are to the end of the file.
`"r"`	Open a text file for **r**ead operations.

■ **Note** Notice that a file mode specification is a character string between double quotes, not a single character between single quotes.

These three modes only apply to text files that are files that are written as characters. You can also work with binary files that are written as a sequence of bytes and I'll discuss that in the section "Binary File Input and Output" later in this chapter. Assuming the call to `fopen()` is successful, the function returns a pointer of type `File *` that you can use to reference the file in further input/output

operations, using other functions in the library. If the file cannot be opened for some reason, fopen() returns a null pointer.

■ **Note** The pointer returned by fopen() is referred to as a **file pointer**, or a **stream pointer**.

So a call to fopen() does two things for you: it creates a file pointer that identifies the specific file on disk that your program is going to operate on, and it determines what you can do with that file within your program.

The pointer that's returned by fopen() is of type FILE * or "pointer to FILE," where FILE specifies a structure type that has been predefined in the header file <stdio.h> through a typedef. The structure that a file pointer points to will contain information about the file. This will be such things as the open mode you specified, the address of the buffer in memory to be used for data, and a pointer to the current position in the file for the next operation. You don't need to worry about the contents of this structure in practice. It's all taken care of by the input/output functions. However, if you really want to know about the FILE structure, you can browse through the library header file.

As I mentioned earlier, when you want to have several files open at once, they must each have their own file pointer variable declared, and you open each of them with a separate call to fopen () with the value that is returned stored in a separate file pointer. There's a limit to the number of files you can have open at one time that will be determined by the value of the constant FOPEN_MAX that's defined in <stdio.h>. FOPEN_MAX is an integer that specifies the maximum number of streams that can be open at one time. The C language standard requires that the value of FOPEN_MAX be at least 8, including stdin, stdout and stderr. Thus, as a minimum, you will be able to be working with up to 5 files simultaneously.

If you want to write to an existing text file with the name myfile.txt, you would use these statements:

```
FILE *pfile = fopen("myfile.txt", "w");  /* Open file myfile.txt to write it */
```

This statement opens the file and associates the physical file specified by the file name myfile.txt with your internal pointer pfile. Because you've specified the mode as "w", you can only write to the file; you can't read from it. The string that you supply as the first argument is limited to a maximum of FILENAME_MAX characters, where FILENAME_MAX is defined in the <stdio.h> header file. This value is usually sufficiently large enough that it isn't a real restriction.

If a file with the name myfile.txt does not already exist, the call to the function fopen() in the previous statement will create a new file with this name. Because you have just provided the file name without any path specification as the first argument to the fopen() function, the file is assumed to be in the current directory, and if the file is not found there, that's where it will be created. You can also specify a string that is the full path and name for the file, in which case the file will be assumed to be at that location and a new file will be created there if necessary. Note that if the directory that's supposed to contain the file doesn't exist when you specify the file path, neither the directory nor the file will be created and the fopen() call will fail. If the call to fopen() does fail for any reason, NULL will be returned. If you then attempt further operations with a NULL file pointer, it will cause your program to terminate.

■ **Note** So here you have the facility to create a new text file. Simply call fopen() with mode "w" and the first argument specifying the name you want to assign to the new file.

On opening a file for writing, the file is positioned at the beginning of any existing data for the first operation. This means that any data that was previously written to the file will be overwritten when you initiate any write operations.

If you want to add to an existing text file rather than overwrite it, you specify mode `"a"`, which is the append mode of operation. This positions the file at the end of any previously written data. If the file specified doesn't exist, as in the case of mode `"w"`, a new file will be created. Using the file pointer that you declared previously, to open the file to add data to the end, use the following statement:

```
pfile = fopen("myfile.txt", "a");      /* Open file myfile.txt to add to it */
```

When you open a file in append mode, all write operations will be at the end of the data in the file on each write operation. In other words, all write operations append data to the file and you cannot update the existing contents in this mode.

If you want to read a file, once you've declared your file pointer, open it using this statement:

```
pfile = fopen("myfile.txt", "r");
```

Because you've specified the mode argument as `"r"`, indicating that you want to read the file, you can't write to this file. The file position will be set to the beginning of the data in the file.

Clearly, if you're going to read the file, it must already exist. If you inadvertently try to open a file for reading that doesn't exist, `fopen()` will return `NULL`. It's therefore a good idea to check the value returned from `fopen()` in an `if` statement, to make sure that you really are accessing the file you want.

Renaming a File

There are many circumstances in which you'll want to rename a file. You might be updating the contents of a file by writing a new, updated file, for instance. You'll probably want to assign a temporary name to the new file while you're creating it, and then change the name to that of the old file once you've deleted it. Renaming a file is very easy. You just use the `rename()` function, which has the following prototype:

```
int rename(const char *oldname, const char *newname);
```

The integer that's returned will be 0 if the name change is successful, and nonzero otherwise. The file must be closed when you call `rename()`, otherwise the operation will fail.

Here's an example of using the `rename()` function:

```
if(rename( "C:\\temp\\myfile.txt", "C:\\temp\\myfile_copy.txt"))
  printf("Failed to rename file.");
else
  printf("File renamed successfully.");
```

The preceding code fragment will change the name of the `myfile.txt` file in the `temp` directory on drive C to `myfile_copy.text`. A message will be produced that indicates whether the name change succeeded. Obviously, if the file path is incorrect or the file doesn't exist, the renaming operation will fail.

■ **Caution** Note the double backslash in the file path string. If you forget to use the escape sequence for a backslash when specifying a Microsoft Windows file path you won't get the file name that you want.

Closing a File

When you've finished with a file, you need to tell the operating system that this is the case and free up your file pointer. This is referred to as **closing** a file. You do this by calling the `fclose()`function which accepts a file pointer as an argument and returns a value of type `int`, which will be `EOF` if an error occurs and 0 otherwise. The typical usage of the `fclose()` function is as follows:

```
fclose(pfile);                          /* Close the file associated with pfile */
```

The result of executing this statement is that the connection between the pointer, `pfile`, and the physical file name is broken, so `pfile` can no longer be used to access the physical file it represented. If the file was being written, the current contents of the output buffer are written to the file to ensure that data isn't lost.

■ **Note** `EOF` is a special character called the **end-of-file character**. In fact, the symbol `EOF` is defined in `<stdio.h>` and is usually equivalent to the value –1. However, this isn't necessarily always the case, so you should use `EOF` in your programs rather than an explicit value. EOF generally indicates that no more data is available from a stream.

It's good programming practice to close a file as soon as you've finished with it. This protects against output data loss, which could occur if an error in another part of your program caused the execution to be stopped in an abnormal fashion. This could result in the contents of the output buffer being lost, as the file wouldn't be closed properly. You must also close a file before attempting to rename it or remove it.

■ **Note** Another reason for closing files as soon as you've finished with them is that the operating system will usually limit the number of files you may have open at one time. Closing files as soon as you've finished with them minimizes the chances of you falling afoul of the operating system in this respect.

There is a function in `<stdio.h>` that will force any unwritten data left in a buffer to be written to a file. This is the function `fflush()`, which you've already used in previous chapters to flush the input buffer. With your file pointer `pfile`, you could force any data left in the output buffer to be written to the file by using this statement:

```
fflush(pfile);
```

The `fflush()` function returns a value of type `int`, which is normally 0 but will be set to `EOF` if an error occurs.

Deleting a File

Because you have the ability to create a file in your code, at some point you'll want to be able to delete a file programmatically, too. The `remove()` function that's declared in `<stdio.h>` does this. You use it like this:

```
remove("pfile.txt");
```

This will delete the file from the current directory that has the name `pfile.txt`. Note that the file should not be open when you call `remove()` to delete it. If the file is open, the effect of calling remove is implementation-defined, so consult your library documentation.

You always need to double-check any operations on files, but you need to take particular care with operations that delete files.

Writing to a Text File

Once you've opened a file for writing, you can write to it any time from anywhere in your program, provided you have access to the pointer for the file that has been set by `fopen()`. So if you want to be able to access a file from anywhere in a program that contains multiple functions, you need to ensure the file pointer has global scope or arrange for it to be passed as an argument to any function that accesses the file.

■ **Note** As you'll recall, to ensure that the file pointer has global scope you place the declaration for it outside of all of the functions, usually at the beginning of the source file.

The simplest write operation is provided by the function `fputc()`, which writes a single character to a text file. It has the following prototype:

```
int fputc(int c, FILE *pfile);
```

The `fputc()` function writes the character specified by the first argument to the file defined by the second argument, which is a file pointer. If the write is successful, it returns the character that was written. Otherwise it returns `EOF`.

In practice, characters aren't written to the physical file one by one. This would be extremely inefficient. Hidden from your program and managed by the output routine, output characters are written to an area of memory called a **buffer** until a reasonable number have been accumulated; they are then all written to the file in one go. This mechanism is illustrated in Figure 12-2.

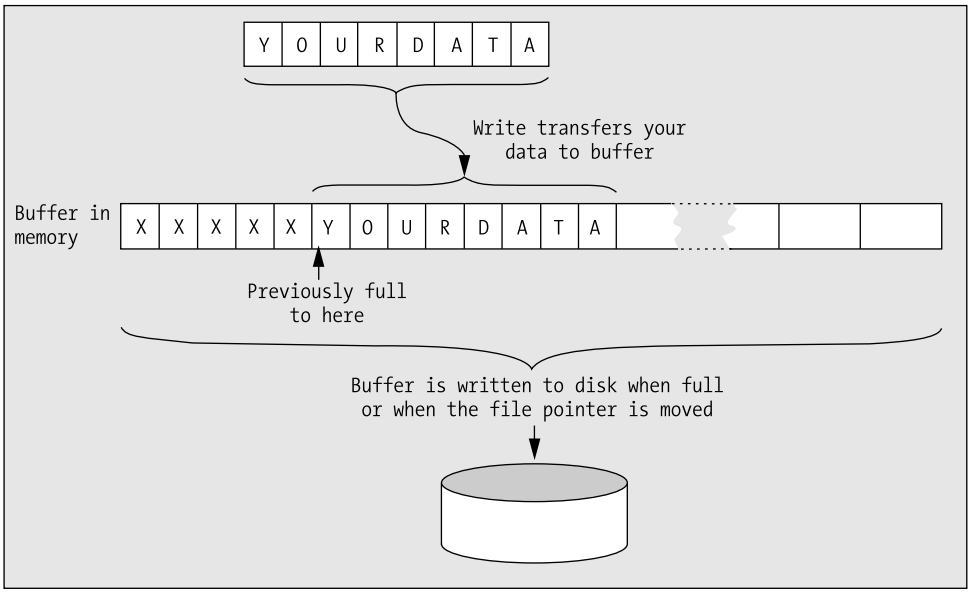

Figure 12-2. *Writing a file*

Note that the putc()function is equivalent to fputc(). It requires the same arguments and the return type is the same. The difference between them is that putc() may be implemented in the standard library as a macro, whereas fputc() is definitely a function.

Reading from a Text File

The fgetc() function is complementary to fputc() and reads a character from a text file that has been opened for reading. It takes a file pointer as its only argument and returns the character read as type int if the read is successful; otherwise, it returns EOF. The typical use of fgetc() is illustrated by the following statement:

```
mchar = fgetc(pfile);                    /* Reads a character into mchar */
```

You're assuming here that the variable mchar has been declared to be of type int.

Behind the scenes, the actual mechanism for reading a file is the inverse of writing to a file. A whole block of characters is read into a buffer in one go. The characters are then handed over to your program one at a time as you request them, until the buffer is empty, whereupon another block is read. This makes the process very fast, because most fgetc() operations won't involve reading the disk but simply moving a character from the buffer in main memory to the place where you want to store it.

Note that the function getc() that's equivalent to fgetc() is also available. It requires an argument of type FILE* and returns the character read as type int, so it's virtually identical to fgetc(). The only difference between them is that getc() may be implemented as a macro, whereas fgetc() is a function.

■ **Caution** Don't confuse the function getc() with the function gets(). They're quite different in operation: getc() reads a single character from the stream specified by its argument, whereas gets() reads a whole line of input from the standard input stream, which is the keyboard. You've already used the gets() function in previous chapters for reading a string from the keyboard.

TRY IT OUT: USING A SIMPLE FILE

You now have enough knowledge of the file input/output capabilities in C to write a simple program that writes a file and then reads it. So let's do just that:

```
/* Program 12.1 Writing a file a character at a time */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

const int LENGTH = 80;                /* Maximum input length   */

int main(void)
{
  char mystr[LENGTH];                 /* Input string           */
  int lstr = 0;                       /* Length of input string */
  int mychar = 0;                     /* Character for output   */
  FILE *pfile = NULL;                 /* File pointer           */
  char *filename = "C:\\myfile.txt";
```

```
  printf("\nEnter an interesting string of less than 80 characters:\n");
  fgets(mystr, LENGTH, stdin);        /* Read in a string       */

  /* Create a new file we can write */

  if(!(pfile = fopen(filename, "w")))
  {
    printf("Error opening %s for writing. Program terminated.", filename);
    exit(1);
  }

  lstr = strlen(mystr);
  for(int i = lstr-1 ; i >= 0 ; i--)
    fputc(mystr[i], pfile);           /* Write string to file backward  */

  fclose(pfile);                      /* Close the file                 */

  /* Open the file for reading */
  if(!(pfile = fopen(filename, "r")))
{
    printf("Error opening %s for reading. Program terminated.", filename);
    exit(1);
  }

  /* Read a character from the file and display it */
  while((mychar = fgetc(pfile)) != EOF)
    putchar(mychar);                  /* Output character from the file */
  putchar('\n');                      /* Write newline                  */

  fclose(pfile);                      /* Close the file                 */
  remove(filename);                   /* Delete the physical file       */
  return 0;
}
```

Here's an example of some output from this program:

```
Enter an interesting string.
Too many cooks spoil the broth.
.htorb eht liops skooc ynam ooT
```

How It Works

The name of the file that you're going to work with is defined by this statement:

```
char *filename = "C:\\myfile.txt";
```

This statement defines the file with the name `myfile.txt` on drive C with the Microsoft Windows notation for file names. As I noted earlier, you must use the escape sequence `'\\'` to get a backslash character. If you forget to do this and just use a single backslash, the compiler will think that you're writing an escape sequence `'\m'` in this case, which it won't recognize as valid.

Before running this program—or indeed any of the examples working with files—do make sure you don't have an existing file with the same name and path. If you have a file with the same name as that used in the example, you should change the initial value for `filename` in the example; otherwise, your existing file will be overwritten.

After displaying a prompt, the program reads a string from the keyboard. It then executes the following statements:

```
if(!(pfile = fopen(filename, "w")))
{
  printf("Error opening %s for writing. Program terminated.", filename);
  exit (1);
}
```

The condition in this `if` statement calls `fopen()` to create the new file `myfile.txt` on drive C, opens it for writing, and stores the pointer that is returned in `pfile`. The second argument to `fopen()` determines the mode as writing the file. The block of statements will be executed if `fopen()` returns `NULL`, so in this case you display a message and call the `exit()` function that is declared in `<stdlib.h>` for an abnormal end to the program.

After determining the `length` of the string using `strlen()` and storing the result in `lstr`, you have a loop defined by these statements:

```
for(int i = lstr-1 ; i >= 0 ; i--)
  fputc(mystr[i], pfile);         /* Write string to file backward  */
```

The loop index is varied from a value corresponding to the last character in the string `lstr-1` back to 0. Therefore, the `putc()` function call within the loop writes to the new file character-by-character, in reverse order. The particular file you're writing is specified by the pointer `pfile` as the second argument to the function call.

After closing the file with a call to `fclose()`, it's reopened in reading mode by these statements:

```
if(!(pfile = fopen(filename, "r")))
{
  printf("Error opening %s for reading. Program terminated.", filename);
  exit(1);
}
```

The mode specification `"r"` indicates that you intend to read the file, so the file position will be set to the beginning of the file. You have the same check for a `NULL` return value as when you wrote the file.

Next, you use the `getc()` function to read characters from the file within the `while` loop condition:

```
while((mychar = fgetc(pfile)) != EOF)
  putchar(mychar);                /* Output character from the file */
```

The file is read character-by-character. The read operation actually takes place within the loop continuation condition. As each character is read, it's displayed on the screen using the function `putc()` within the loop. The process stops when `EOF` is returned by `getc()` at the end of the file.

The last two statements before the return in the `main()` function are the following:

```
fclose(pfile);                    /* Close the file               */
remove(filename);                 /* Delete the physical file     */
```

These statements provide the necessary final tidying up, now that you've finished with the file. After closing the file, the program calls the `remove()` function, which will delete the file identified by the argument. This avoids cluttering up the disk with stray files. If you want to check the contents of the file that was written using a text editor, just remove or comment out the call to `remove()`.

Writing Strings to a Text File

Analogous to the `puts()` function for writing a string to `stdout`, you have the `fputs()` function for writing a string to a text file. Its prototype is as follows:

```
int fputs(char *pstr, FILE *pfile);
```

The first argument is a pointer to the character string that's to be written to the file, and the second argument is a file pointer. The operation of the function is slightly odd, in that it continues to write characters from a string until it reaches a '\0' character, which it doesn't write to the file. This can complicate reading back variable-length strings from a file that have been written by fputs(). It works this way because it's a character write operation, not a binary write operation, so it's expecting to write a line of text that has a newline character at the end. A newline character isn't required by the operation of the function, but it's very helpful when you want to read the file back (using the complementary fgets() function, as you'll see).

The fputs() function returns EOF if an error occurs, and 0 under normal circumstances. You use it in the same way as puts(), for example

```
fputs("The higher the fewer", pfile);
```

This will output the string appearing as the first argument to the file pointed to by pfile.

Reading Strings from a Text File

Complementing fputs() is the function fgets() for reading a string from a text file. It has the following prototype:

```
char *fgets(char *pstr, int nchars, FILE *pfile);
```

The fgets() function has three parameters. The function will read a string into the memory area pointed to by pstr, from the file specified by pfile. Characters are read from the file until either a '\n' is read or nchars-1 characters have been read from the file, whichever occurs first.

If a newline character is read, it's retained in the string. A '\0' character will be appended to the end of the string in any event. If there is no error, fgets() will return the pointer, pstr; otherwise, NULL is returned. The second argument to this function enables you to ensure that you don't overrun the memory area that you've assigned for input in your program. To prevent the capacity of your data input area from being exceeded, just specify the length of the area or the array that will receive the input data as the second argument to the function.

TRY IT OUT: TRANSFERRING STRINGS TO AND FROM A TEXT FILE

You can exercise the functions to transfer strings to and from a text file in an example that also uses the append mode for writing a file:

```
/* Program 12.2  As the saying goes...it comes back! */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

const int LENGTH = 80;                  /* Maximum input length */

int main(void)
{
  char *proverbs[] =
```

```
           {  "Many a mickle makes a muckle.\n",
              "Too many cooks spoil the broth.\n",
              "He who laughs last didn't get the joke in"
                                  " the first place.\n"
           };

char more[LENGTH];                  /* Stores a new proverb */
FILE *pfile = NULL;                 /* File pointer         */
char *filename = "C:\\myfile.txt";

/* Create a new file( if myfile.txt does not exist */
if(!(pfile = fopen(filename, "w")))    /* Open the file to write it */
{
  printf("Error opening %s for writing. Program terminated.", filename);
  exit(1);
}

/* Write our first three sayings to the file. */
int count = sizeof proverbs/sizeof proverbs[0];
for(int i = 0 ; i < count ; i++)
  fputs(proverbs[i], pfile);

fclose(pfile);                      /* Close the file */

/* Open the file to append more proverbs */
if(!(pfile = fopen(filename, "a")))
{
  printf("Error opening %s for writing. Program terminated.", filename);
  exit(1);
}

printf("Enter proverbs of less than 80 characters or press Enter to end:\n");
while(true)
{
  fgets(more, LENGTH, stdin);          /* Read a proverb           */
  if(more[0] == '\n')                  /* If its empty line        */
    break;                             /* end input operation      */
  fputs(more, pfile);                  /* Write the new proverb    */
}

fclose(pfile);                         /* Close the file           */

if(!(pfile = fopen(filename, "r")))    /* Open the file to read it */
{
  printf("Error opening %s for writing. Program terminated.", filename);
  exit(1);
}

/* Read and output the file contents */
printf("The proverbs in the file are:\n\n");
while(fgets(more, LENGTH, pfile))      /* Read a proverb */
 printf("%s", more);                   /* and display it */
```

```
  fclose(pfile);                          /* Close the file */
  remove(filename);                       /* and remove it  */
  return 0;
}
```

Here is some sample output from this program:

```
Enter proverbs of less than 80 characters or press Enter to end:
Least said, soonest mended.
A nod is as good as a wink to a blind horse.

The proverbs in the file are:

Many a mickle makes a muckle.
Too many cooks spoil the broth.
He who laughs last didn't get the joke in the first place.
Least said, soonest mended.
A nod is as good as a wink to a blind horse.
```

How It Works

You initialize the array of pointers, `proverbs[]`, in the following statement:

```
char *proverbs[] =
           {  "Many a mickle makes a muckle.\n",
              "Too many cooks spoil the broth.\n",
              "He who laughs last didn't get the joke in"
                                  " the first place.\n"
           };
```

You specify the three sayings as initial values for the array elements, and this causes the compiler to allocate the space necessary to store each string.

You have a further declaration of an array that will store a proverb that will be read from the keyboard:

```
char more[LENGTH];                    /* Stores a new proverb */
```

This initializes a conventional `char` array with another proverb. You also include `'\n'` at the end for the same reason as before.

After creating and opening a file on drive C for writing, the program writes the initial three proverbs to the file in a loop:

```
int count = sizeof proverbs/sizeof proverbs[0];
for(int i = 0 ; i < count ; i++)
  fputs(proverbs[i], pfile);
```

The contents of each of the memory areas pointed to by elements of the `proverbs[]` array are written to the file in the `for` loop using the function `fputs()`. This function is extremely easy to use; it just requires a pointer to the string as the first argument and a pointer to the file as the second.

The number of proverbs in the array is calculated by the following expression:

```
sizeof proverbs/sizeof proverbs[0]
```

The expression `sizeof proverbs` will evaluate to the total number of bytes occupied by the complete array, and `sizeof proverbs[0]` will result in the number of bytes required to store a single pointer in one element of the array. Therefore, the whole expression will evaluate to the number of elements in the pointer array. You could

have manually counted how many initializing strings you supplied, of course, but doing it this way means that the correct number of iterations is determined automatically, and this expression will still be correct even if the array dimension is changed by adding more initializing strings.

Once the first set of proverbs has been written, the file is closed and then reopened with this statement:

```
if(!(pfile = fopen(filename, "a")))
{
  printf("Error opening %s for writing. Program terminated.", filename);
  exit(1);
}
```

Because you have the mode specified as `"a"`, the file is opened in append mode. Note that the current position for the file is automatically set to the end of the file in this mode, so that subsequent write operations will be appended to the end of the existing data in the file.

After prompting for input, you read more proverbs from the keyboard and write them to the file with the following statements:

```
while(true)
{
  fgets(more, LENGTH, stdin);          /* Read a proverb           */
  if(more[0] == '\n')                  /* If its empty line        */
    break;                             /* end input operation      */
  fputs(more, pfile);                  /* Write the new proverb    */
}
```

Each additional proverb that's stored in the `more` array is written to the file using `fputs()`. As you can see, the function `fputs()` is just as easy to use with an array as it is with a pointer. Because you're in append mode, each new proverb will be added at the end of the existing data in the file. The loop terminates when an empty line is entered. An empty line will result in a string containing just `'\n'` followed by the string terminator.

Having written the file, you close it and then reopen it for reading, using the mode specifier `"r"`. You then have the following loop:

```
while(fgets(more, LENGTH, pfile))      /* Read a proverb */
 printf("%s", more);                   /* and display it */
```

You read strings successively from the file into the `more` array within the loop continuation condition. After each string is read, you display it on the screen by the call to `printf()` within the loop. The reading of each proverb by `fgets()` is terminated by detecting the `'\n'` character at the end of each string. The loop terminates when the function `fgets()` returns `NULL`.

Finally, the file is closed and then deleted using the function `remove()` in the same fashion as the previous example.

Formatted File Input and Output

Writing characters and strings to a text file is all very well as far as it goes, but you normally have many other types of data in your programs. To write numerical data to a text file, you need something more than you've seen so far, and where the contents of a file are to be human readable, you need a character representation of the numerical data. The mechanism for doing just this is provided by the functions for formatted file input and output.

Formatted Output to a File

You already encountered the function for formatted output to a file when I discussed standard streams back in Chapter 10. It's virtually the same as the `printf()` statement, except that there's one extra parameter and a slight name change. Its typical usage is the following:

```
fprintf(pfile, "%12d%12d%14f", num1, num2, fnum1);
```

As you can see, the function name has an additional `f` (for *file*), compared with `printf()`, and the first argument is a file pointer that specifies the destination of the data to be written. The file pointer obviously needs to be set through a call to `fopen()` first. The remaining arguments are identical to that of `printf()`. This example writes the values of the three variables `num1`, `num2`, and `num3` to the file specified by the file pointer `pfile`, under control of the format string specified as the second argument. Therefore, the first two variables are of type `int` and are to be written with a field width of 12, and the third variable is of type `float` and is to be written to the file with a field width of 14.

Formatted Input from a File

You get formatted input from a file by using the function `fscanf()`. To read three variable values from a file `pfile` you would write this:

```
fscanf(pfile, "%12d%12d%14f", &num1, &num2, &fnum1);
```

This function works in exactly the same way as `scanf()` does with `stdin`, except that here you're obtaining input from a file specified by the first argument. The same rules govern the specification of the format string and the operation of the function as apply to `scanf()`. The function returns `EOF` if an error occurs such that no input is read; otherwise, it returns the number of values read as a value of type `int`.

TRY IT OUT: USING FORMATTED INPUT AND OUTPUT FUNCTIONS

You can demonstrate the formatted input and output functions with an example that will also show what's happening to the data in these operations:

```
/* Program 12.3 Messing about with formatted file I/O */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  long num1 = 234567L;                    /* Input values...              */
  long num2 = 345123L;
  long num3 = 789234L;

  long num4 = 0L;                         /* Values read from the file... */
  long num5 = 0L;
  long num6 = 0L;

  float fnum = 0.0f;                      /* Value read from the file     */
  int   ival[6] = { 0 };                  /* Values read from the file    */
  FILE *pfile = NULL;                     /* File pointer                 */
  char *filename = "C:\\myfile.txt";
```

```
  pfile = fopen(filename, "w");                /* Create file to be written */
  if(pfile == NULL)
  {
    printf("Error opening %s for writing. Program terminated.", filename);
    exit(1);
  }
  fprintf(pfile, "%6ld%6ld%6ld", num1, num2, num3);   /* Write file          */
  fclose(pfile);                                      /* Close file          */
  printf("\n %6ld %6ld %6ld", num1, num2, num3);   /* Display values written */

  pfile = fopen(filename, "r");                       /* Open file to read   */
  fscanf(pfile, "%6ld%6ld%6ld", &num4, &num5 ,&num6); /* Read back           */
  printf("\n %6ld %6ld %6ld", num4, num5, num6);      /* Display what we got */

  rewind(pfile);                          /* Go to the beginning of the file */
  fscanf(pfile, "%2d%3d%3d%3d%2d%2d%3f", &ival[0], &ival[1], /* Read it again */
                    &ival[2], &ival[3], &ival[4] , &ival[5], &fnum);
  fclose(pfile);                                     /* Close the file and    */
  remove(filename);                                  /* delete physical file. */

  /* Output the results */
  printf("\n");
  for(int i = 0 ; i < 6 ; i++ )
    printf("%sival[i] = %d", i == 4 ? "\n\t" : "\t", i, ival[i]);
  printf("\nfnum = %f\n", fnum);
  return 0;
}
```

The output from this example is the following:

```
 234567 345123 789234
 234567 345123 789234
        ival[i] = 0     ival[i] = 1     ival[i] = 2     ival[i] =3
        ival[i] = 4     ival[i] = 5
fnum = 234.000000
```

How It Works

This example writes the values of `num1`, `num2`, and `num3`, which are defined and assigned values in their declaration, to the file `myfile.txt` on drive C. This is referenced through the pointer `pfile`. The file is closed and reopened for reading, and the values are read from the file in the same format as they are written. You then have the following statement:

```
rewind(pfile);
```

This statement calls the `rewind()` function, which simply moves the current position back to the beginning of the file so that you can read it again. You could have achieved the same thing by closing the file then reopening it again, but with `rewind()` you do it with one function call and the operation will be a lot faster.

Having repositioned the file, you read the file again with this statement:

```
fscanf(pfile, "%2d%3d%3d%3d%2d%2d%3f", &ival[0], &ival[1], /* Read it again */
                    &ival[2], &ival[3], &ival[4] , &ival[5], &fnum);
```

This statement reads the same data into the array `ival[]` and the variable `fnum`, but with different formats from those that you used for writing the file. You can see from the effects of this that the file consists of just a string of characters once it has been written, exactly the same as the output to the screen from `printf()`.

■ **Note** You can lose information if you choose a format specifier that outputs fewer digits precision than the stored value holds.

You can see that the values you get back from the file when you read it will depend on both the format string that you use and the variable list that you specify in the `fscanf()` function.

None of the intrinsic source information that existed when you wrote the file is necessarily maintained. Once the data is in the file, it's just a sequence of bytes in which the meaning is determined by how you interpret them. This is demonstrated quite clearly by this example, in which you've converted the original three values into eight new values.

Lastly, you leave everything neat and tidy in this program by closing the file and using the function `remove()` to delete it.

Dealing with Errors

The examples in this book have included minimal error checking and reporting because the code for comprehensive error checking and reporting tends to take up a lot of space in the book and make the programs look rather more complicated than they really are. In real-world programs, however, it's essential that you do as much error checking and reporting as you can.

Generally, you should write your error messages to `stderr`, which is automatically available to your program and always points to your display screen. Even though `stdout` may be redirected to a file by an operating system command, `stderr` continues to be assigned to the screen. It's important to check that a file you want to read does in fact exist and you have been doing this in the examples, but there's more that you can do. First of all, you can write error messages to `stderr` rather than `stdin`, for example

```
char *filename = "C:\\MYFILE.TXT";     /* File name    */
FILE *pfile = NULL;                    /* File pointer */

if(!(pfile = fopen(filename, "r")))
{
  fprintf(stderr, "\nCannot open %s to read it.", filename);
  exit(1);
}
```

The merit of writing to `stderr` is that the output will always be directed to the display and it will always be written immediately to the display device. This means that you will always see the output directed to `stderr`, regardless of what happens in the program. The `stdin` stream is buffered, so there is the risk that data could be left in the buffer and never displayed if your program crashes. Terminating a program by calling `exit()` ensures that output stream buffers will be flushed so output will be written to the ultimate destination. The stream `stdin` can be redirected to a file, but `stderr` can't be redirected simply to ensure that the output always occurs.

Knowing that some kind of error occurred is useful, but you can do more than this. The `perror()` function outputs a string that you pass as an argument plus an implementation-defined error message corresponding to the error that occurred. You could therefore rewrite the previous fragment as follows:

```
if(!(pfile = fopen(myfile, "r")))
{
  perror(strcat("Error opening ", filename));
  exit(1);
}
```

This will output your message consisting of the file name appended to the first argument to `strcat()`, plus a system-generated message relating to the error. The output will be written to `stderr`.

If an error occurs when you're reading a file, you can check whether the error is due to reaching the end of file. The `feof()` function will return a nonzero integer if the end of file has been reached, so you can check for this with statements such as these:

```
if(feof(pfile))
  printf("End of file reached.");
```

Note that I didn't write the message to `stderr` here because reaching the end of the file isn't necessarily an error.

The `ferror()` function returns a nonzero integer if an error occurrs with an operation on the stream that's identified by the file pointer that you pass as the argument. Calling this function enables you to establish positively that an error did occur. The `<errno.h>` header file defines a value with the name `errno` that may indicate what kind of file error has occurred. You need to read the documentation for your C implementation to find out the specifics of this. The value of `errno` may be set for errors other than just file operations.

You should always include some basic error checking and reporting code in all of your programs. Once you've written a few programs, you'll find that including some standard bits of code for each type of operation warranting error checks is no hardship. With a standard approach, you can copy most of what you need from one program to another.

Further Text File Operation Modes

Text mode is the default mode of operation with the open modes you have seen up to now, but in earlier versions of C you could specify explicitly that a file is to be opened in text mode. You could do this by adding `t` to the end of the existing specifiers. This gives you the mode specifiers `"wt"`, `"rt"`, and `"at"` in addition to the original three. I am only mentioning this because you may come across it in other C programs. Although most compilers will support this, it's not specifically part of the current C standard so it is best not to use this option in your code.

You can also open a text file for update—that is, for both reading and writing—using the specifier `"r+"`. You can also specify the open mode as `"w+"` if you want to both read and write a new file, or when you want to discard the original contents of an existing file before you start. Opening a file with the mode `"w+"` truncates the length of an existing file to zero, so only use this mode when you want to discard the current file contents. In older programs you may come across these modes written as `"rt+"` or `"r+t"` and `"wt+"` or `"w+t"`.

As I've said, in update mode you can both read and write a text file. However, you can't write to the file immediately after reading it or read from the file immediately after writing it, unless the `EOF` has been reached or the position in the file has been changed by some means. (This involves calling a function such as `rewind()` or some other function that modifies the file position.) The reason for this is that writing to a file doesn't necessarily write the data to the external device. It simply transfers

it to a buffer in memory that's written to the file once it's full, or when some other event causes it to be written. Similarly, the first read from a file will fill a buffer area in memory, and subsequent reads will transfer data from the buffer until it's empty, whereupon another file read to fill the buffer will be initiated. This is illustrated in Figure 12-3.

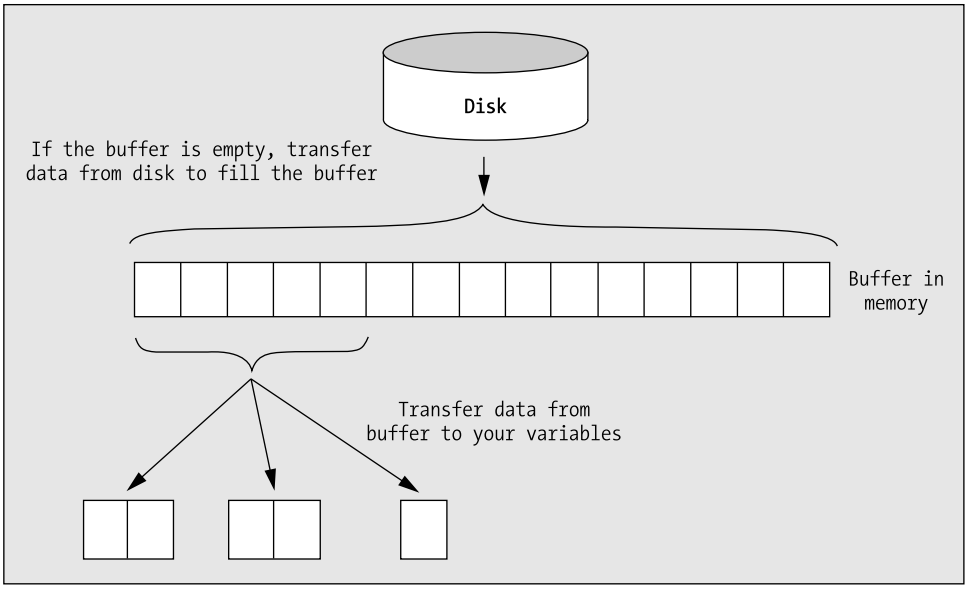

Figure 12-3. *Buffered input operations*

This means that if you were able to switch immediately from write mode to read mode, data would be lost because it would be left in the buffer. In the case of switching from read mode to write mode, the current position in the file may be different from what you imagine it to be, and you may inadvertently overwrite data on the file. A switch from read to write or vice versa, therefore, requires an intervening event that implicitly flushes the buffers. The `fflush()` function will cause the bytes remaining in an output buffer for the stream you pass as the argument to be written to an output file.

Binary File Input and Output

The alternative to text mode operations on a file is **binary mode**. In this mode, no transformation of the data takes place, and there's no need for a format string to control input or output, so it's much simpler than text mode. The binary data as it appears in memory is transferred directly to the file. Characters such as `'\n'` and `'\0'` that have specific significance in text mode are of no consequence in binary mode.

Binary mode has the advantage that no data is transformed or precision lost, as can happen with text mode due to the formatting process. It's also somewhat faster than text mode because no transformation operations are performed. The two modes are contrasted in Figure 12-4.

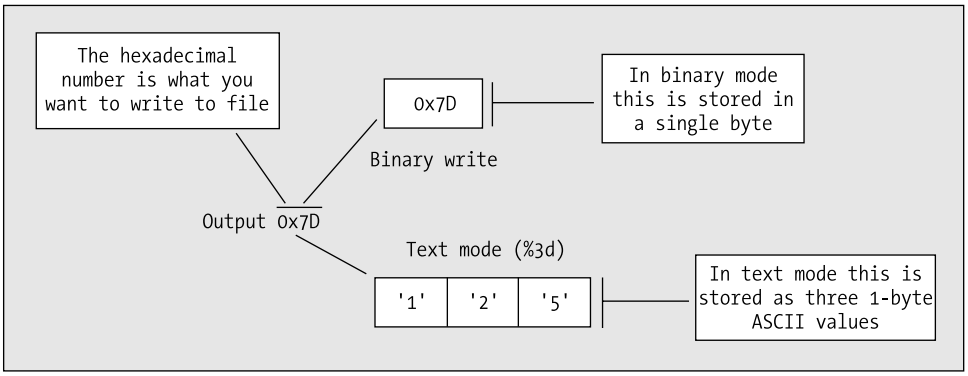

Figure 12-4. *Contrasting binary mode and text mode*

Specifying Binary Mode

You specify binary mode by appending `b` to the basic open mode specifiers I introduced initially. Therefore, you have the additional open mode specifiers `"wb"` for writing a binary file, `"rb"` to read a binary file, `"ab"` to append data to the end of a binary file, and `"rb+"` to enable reading and writing of a binary file.

Because binary mode involves handling the data to be transferred to and from the file in a different way from text mode, you have a new set of functions to perform input and output.

Writing a Binary File

You use the `fwrite()` function to write a binary file. This is best explained with an example of its use. Suppose that you open the file to be written with the following statements:

```
  char *filename = "myfile.bin";
FILE *pfile = fopen(filename, "wb");
```

The `filename` variable points to the string that defines the name of the file, and `pfile` is a variable to store a pointer to an object of type `FILE` as before.

You could write to the file with these statements:

```
long pdata[] = {2L, 3L, 4L};
int num_items = sizeof(pdata)/sizeof(long);
FILE *pfile = fopen(filename, "wb");
size_t wcount = fwrite(pdata, sizeof(long), num_items, pfile);
```

The `fwrite()` function operates on the principle of writing a specified number of binary data items to a file, where each item is a given number of bytes `long`. The first argument, `pdata`, is a pointer containing the starting address in memory of where the data items to be written are stored. The second argument specifies the size in bytes of each item to be written. The third argument, `num_items`, defines a count of the number of items to be written to the file. The file to which the data is to be transferred is identified by the last argument, `pfile`. The function `fwrite()` returns the count of the number of items actually written as a value of type `size_t`. If the operation is unsuccessful for some reason, this value will be less than `num_items`.

Note that there is no check that you opened the file in binary mode when you call the `fwrite()` function. The write operation will write binary data to a file that you open in text mode. Equally, there is nothing to prevent you from writing text data to a binary file. Of course, if you do this a considerable amount of confusion is likely to result.

The return value and the second and third arguments to the function are all of the same type as that returned by the `sizeof` operator. This is defined as type `size_t`, which you probably remember is an unsigned integer type.

The code fragment above uses the `sizeof` operator to specify the size in bytes of the objects to be transferred and also determines the number of items to be written using the expression `sizeof(pdata)/sizeof(long)`. This is a good way of specifying these values when this is possible, because it reduces the likelihood of error. Of course, in a real context, you should also check the return value in `wcount` to be sure the write is successful.

The `fwrite()` function is geared to writing a number of binary objects of a given length to a file. You can write in units of your own structures as easily as you can write values of type `int`, values of type `double`, or sequences of individual bytes.

This doesn't mean that the values you write in any given output operation all have to be of the same type. You might allocate some memory using `malloc()`, for instance, into which you assemble a sequence of data items of different types and lengths. You could then write the whole block of memory in one go as a sequence of bytes. Of course, when you come to read them back, you need to know the precise sequence and types for the values in the file if you are to make sense of them.

Reading a Binary File

You use the `fread()` function to read a binary file once it has been opened in read mode. Using the same variables as in the example of writing a binary file, you could read the file using a statement such as this:

```
size_t wcount = fread( pdata, sizeof(long), num_items, pfile);
```

This operates exactly as the inverse of the write operation. Starting at the address specified by `data`, the function reads `num_items` objects, each occupying the number of bytes specified by the second argument. The function returns the count of the number of items that were read. If the read isn't completely successful, the count will be less than the number of objects requested.

TRY IT OUT: READING A BINARY FILE

You can apply the binary file operations to a version of the Program 7.11 you saw in Chapter 7 for calculating primes. This time, you'll use a disk file as a buffer to calculate a larger number of primes. You can make the program automatically spill primes on to a disk file if the array assigned to store the primes is insufficient for the number of primes requested. In this version of the program to find primes, you'll improve the checking process a little.

In addition to the `main()`function that will contain the prime finding loop, you'll write a function to test whether a value is prime called `test_prime()`, a helper function that will check a given value against a block of primes called `check()`, and a function called `put_primes()`, which will retrieve the primes from the file and display them.

As this program consists of several functions and will work with variables at global scope, let's take a look at it piece by piece. We'll start with the function prototypes and global data before we look at detail of the functions:

```
/* Program 12.4  A prime example using binary files */
#include <stdio.h>
#include <stdlib.h>
#include <math.h >                    /* For square root function sqrt() */

/* Function prototypes */
int test_prime(unsigned long long N);
void put_primes(void);
int check(unsigned long long buffer[], size_t count, unsigned long long N);
```

```
/* Global data */
const unsigned int MEM_PRIMES = 100;  /* Count of number of primes in memory */

struct
{
  char *filename;                                  /* File name for primes   */
  FILE *pfile;                                     /* File stream pointer    */
  int nrec;                                        /* Number of file records */

  unsigned long long primes[MEM_PRIMES];           /* Array to store primes  */
  size_t index;                    /* Index of free location in array primes */

} global = { "C:\\myfile.bin",                     /* Physical file name     */
             NULL,                                 /* File pointer value     */
             0,                                    /* File record count      */
             {2ULL, 3ULL, 5ULL},                   /* primes array values    */
             3                                     /* Number of primes       */
           };

int main(void)
{
  /* Code for main()... */
}

/*********************************************************************
 * Function to test if a number, N, is prime using primes in       *
 * memory and on file                                              *
 * First parameter N - value to be tested                          *
 * Return value - a positive value for a prime, zero otherwise     *
 *********************************************************************/
int test_prime(unsigned long long N)
{
  /* Code for test_prime()... */
}

/*********************************************************************
 * Function to check whether an integer, N, is divisible by any    *
 * of the elements in the array pbuffer up to the square root of N.*
 * First parameter buffer - an array of primes                     *
 * second parameter count - number of elements in pbuffer          *
 * Third parameter N - the value to be checked                     *
 * Return value - 1 if N is prime, zero if N is not a prime,       *
 *               -1 for more checks                                *
 *********************************************************************/
int check(unsigned long long buffer[], size_t count, unsigned long long N)
{
  /* Code for check()... */
}
```

```
/*****************************************
 * Function to output primes from the file *
 *****************************************/
void put_primes(void)
{
  /* Code for put_primes()... */
}
```

After the usual `#include` statements, you have the prototypes for three functions used in the program:

```
int test_prime(unsigned long long N);
void put_primes(void);
int check(unsigned long long buffer[], size_t count, unsigned long long N);
```

You write just the parameter types within the prototypes without using a parameter name. Function prototypes can be written either with or without parameter names, but the parameter types must be specified. Generally, it's better to include names, because they give a clue to the purpose of the parameter. The names in the prototype can be different from the names used in the definition of the function, but you should only do this if it helps to make the code more readable. To allow the maximum range of possible prime values you'll store them as values of type `unsigned long long`.

It is often convenient to define variables at global scope when they need to be accessed by several functions. This avoids the need for long parameter lists for the functions where you would pass the data as arguments in each function call. However, the more names you declare at global scope, the greater the risk of collisions with local names in your program or names used in the standard libraries, so it is always a good idea to keep the number of names at global scope to a minimum. One way to reduce the number of names at global scope is to put your global variables inside a `struct`. Here we have defined a `struct` without a type name, and the `struct` variable has the name `global`. All the variables defined as members of the `struct` therefore do not appear at global scope because they must be qualified by the name `global`.

The first three members of the `struct` are `filename` that points to the name of the file that will store primes, the file stream pointer, `pfile`, and the `nrec` variable to store the number of records in the file. You then have the `primes` array that will hold up to `MEM_PRIMES` values in memory before they need to be written to the file followed by the `index` variable that records the current free element in the `primes` array.

You can see how we initialize the members of the `struct` using an initializer list. Note how the initial values for three elements in the `primes` array appear between braces; because there are only three initializers for the array, the remaining elements will be set to 0.

Here's the definition for `main()`:

```
int main(void)
{
  unsigned long long trial = 5ULL;                      /* Prime candidate */
  unsigned long num_primes = 3UL;                       /* Prime count     */
  unsigned long total = 0UL;                            /* Total required  */

  printf("How many primes would you like?  ");
  scanf("%lu", &total);                 /* Total is how many we need to find */
  total = total<4UL ? 4UL:total;        /* Make sure it is at least 4        */
```

```
  /* Prime finding and storing loop */
  while(num_primes < total)             /* Loop until we get total required  */
  {
    trial += 2ULL;                      /* Next value for checking           */
    if(test_prime(trial))               /* Check if trial is prime           */
    {                                   /* Positive value means prime        */
       global.primes[global.index++] = trial;  /* so store it                */
      num_primes++;                     /* Increment total number of primes  */

      if(global.index == MEM_PRIMES)    /* Check if array is full            */
      {
        /* File opened OK?   */
        if(!(global.pfile = fopen(global.filename, "ab")))
        { /* No, so explain and end the program */
          printf("\nUnable to open %s to append\n", global.filename);
          exit(1);
        }
        /* Write the array    */
        fwrite(global.primes, sizeof(unsigned long long),
                                            MEM_PRIMES, global.pfile);

        fclose(global.pfile);           /* Close the file                */
        global.index = 0U;              /* Reset count of primes in memory */
        global.nrec++;                  /* Increment file record count     */
      }
    }
  }

  if(total>MEM_PRIMES)                  /* If we wrote some to file         */
    put_primes();                       /* Display the contents of the file */
  if(global.index)                      /* Display any left in memory       */
    for(size_t i = 0; i<global.index ; i++)
    {
      if(i%5 == 0)
        printf("\n");                               /* Newline after five     */
      printf("%12llu", global.primes[i]);           /* Output a prime         */
    }

  if(total>MEM_PRIMES)                              /* Did we need a file?    */
    if(remove(global.filename))                     /* then delete it.        */
      printf("\nFailed to delete %s\n", global.filename); /* Delete failed   */
    else
      printf("\nFile %s deleted.\n", global.filename);    /* Delete OK       */
  return 0;
}
```

How It Works

The definition of the function `main()` follows the global declarations. When the program executes, you enter the number of primes you want to find, and this value controls the loop for testing prime candidates. Checking for a prime is performed by the function `test_prime()`, which is called in the `if` statement condition within the loop. The function returns 1 if the value tested is prime, and 0 otherwise. If a prime is found, then you execute these statements:

```
    global.primes[global.index++] = trial;   /* so store it               */
    num_primes++;                     /* Increment total number of primes  */
```

The first statement stores the prime that you've found in the `global.primes[]` array. You keep track of how many primes you have in total with the variable `num_primes`, and the `struct` member variable `global.index` records how many you have in memory at any given time.

Every time you find a prime and add it to the `primes[]` array, you perform the following check:

```
    if(global.index == MEM_PRIMES)    /* Check if array is full           */
    {
      /* File opened OK?   */
      if(!(global.pfile = fopen(global.filename, "ab")))
      { /* No, so explain and end the program */
        printf("\nUnable to open %s to append\n", global.filename);
        exit(1);
      }
      /* Write the array    */
      fwrite(global.primes, sizeof(unsigned long long),
                                          MEM_PRIMES, global.pfile);

      fclose(global.pfile);              /* Close the file                */
      global.index = 0U;                 /* Reset count of primes in memory */
      global.nrec++;                     /* Increment file record count     */
    }
```

If you've filled the `global.primes` array, the `if` condition will be true and you'll execute the associated statement block. In this case, the file is opened in binary mode to append data. The first time this occurs, a new file will be created. On subsequent calls of the `fopen()` function file, the existing file will be opened with the current position set at the end of any existing data in the file, ready for the next block to be written. After writing a block, the file is closed, as it will be necessary to reopen it for reading in the function that performs the checking of prime candidates.

Finally in this group of statements, the count of the number of primes in memory is reset to 0 because they've all been safely stowed away, and the count of the number of blocks of primes written to the file is incremented.

When sufficient primes have been found to fulfill the number requested, you display the primes with the following statements:

```
  if(total>MEM_PRIMES)                 /* If we wrote some to file        */
    put_primes();                      /* Display the contents of the file */
  if(global.index)                     /* Display any left in memory      */
    for(size_t i = 0; i<global.index ; i++)
    {
      if(i%5 == 0)
        printf("\n");                            /* Newline after five    */
      printf("%12llu", global.primes[i]);        /* Output a prime        */
    }
```

It's quite possible that the number of primes requested can be accommodated in memory, in which case you won't write to a file at all. You must therefore check whether `total` exceeds `MEM_PRIMES` before calling the function `put_primes()` that outputs the primes in the file. If the value of `global.index` is positive, there are primes in the `global.primes` array that haven't been written to the file. In this case, you display these in the `for` loop, five to a line.

Finally in `main()`, you remove the file from the disk with the following statements:

```
  if(total>MEM_PRIMES)                          /* Did we need a file?    */
    if(remove(global.filename))                 /* then delete it.        */
      printf("\nFailed to delete %s\n", global.filename); /* Delete failed   */
    else
      printf("\nFile %s deleted.\n", global.filename);    /* Delete OK       */
```

The first `if` ensures that you don't attempt to delete the file if you didn't create one.

The implementation of the function to check whether a value is prime is as follows:

```
int test_prime(unsigned long long N)
{
  unsigned long long buffer[MEM_PRIMES]; /* local buffer for primes from file */

  int k = 0;

  if(global.nrec > 0)                           /* Have we written records? */
  {
    if(!(global.pfile = fopen(global.filename, "rb"))) /* Then open the file */
    {
      printf("\nUnable to open %s to read\n", global.filename);
      exit(1);
    }

    for(size_t i = 0; i < global.nrec ; i++)
    { /* Check against primes in the file first */
      /* Read primes */
      fread(buffer, sizeof( long long), MEM_PRIMES, global.pfile);
      if((k = check(buffer, MEM_PRIMES, N)) >= 0)  /* Prime or not?          */
      {
        fclose(global.pfile);                   /* Yes, so close the file */
        return k;                               /* 1 for prime, 0 for not */
      }
    }
    fclose(global.pfile);                       /* Close the file         */
  }

  /* Check against primes in memory */
  return check(global.primes, global.index, N);
}
```

The `test_prime()` function accepts a candidate value as an argument and returns 1 if it's prime and 0 if it isn't. If you've written anything to the file, this will be indicated by a positive value of `global.nrec`. In this case, the primes in the file need to be used as divisors first, because they are lower than those currently in memory as you compute them in sequence.

■ **Note** As you may remember, a prime is a number with no factors other than 1 and itself. It's sufficient to check whether a number is divisible by any of the primes less than the square root of the number to verify that it's prime. This follows from the simple logic that any exact divisor greater than the square root must have an associated factor (the result of the division) that's less than the square root.

To read the file, the function executes this statement:

```
fread(buffer, sizeof( long long), MEM_PRIMES, global.pfile);
```

This reads one block of primes from the file into the array `buffer`. The second argument defines the size of each object to be read, and `MEM_PRIMES` defines the number of objects of the specified size to be read.

Having read a block, the following check is executed:

```
if((k = check(buffer, MEM_PRIMES, N)) >= 0)  /* Prime or not?         */
{
  fclose(global.pfile);                      /* Yes, so close the file */
  return k;                                  /* 1 for prime, 0 for not */
}
```

Within the `if` condition, the `check()` function is called to determine if any of the array elements divide into the prime candidate with no remainder. This function returns 0 if an exact division is found, indicating the candidate isn't prime. If no exact division is found with primes up to the square root of the candidate value, 1 is returned, indicating that the candidate must be prime. Whatever value is returned from `check()`, in both cases you've finished checking so the file is closed and the same value is returned to `main()`.

The value –1 is returned from `check()` if no exact division has been found, but the square root of the test value hasn't been exceeded. You don't need to check for the –1 return explicitly, because it's the only possibility left if the value returned from `check()` isn't 0 or 1. In this case, the next block, if there is one, is read from the file in the next iteration of the `for` loop.

If the contents of the file have been exhausted without determining whether `N` is prime, the `for` loop will end, and you'll close the file and execute the following statement:

```
return check(global.primes, global.index, N);
```

Here, the test value is tried against any primes in the `primes` array in memory by the function `check()`. If a prime is found, the `check()` function will return 1; otherwise, the function will return 0. The value that's returned by the `check()` function will be returned to `main()`.

The code for the `check()` function is as follows:

```
int check(unsigned long long buffer[], size_t count, unsigned long long N)
{
  /* Upper limit */
  unsigned long long root_N = (unsigned long long)(1.0 + sqrt(N));

  for(size_t i = 0 ; i<count ; i++)
  {
    if(N % buffer[i] == 0ULL )              /* Exact division?             */
      return 0;                             /* Then not a prime           */

    if(buffer[i] > root_N)                  /* Divisor exceeds square root? */
      return 1;                             /* Then must be a prime       */
  }
  return -1;                                /* More checks necessary...   */
}
```

The role of this function is to check if any of the primes contained in the `buffer` array that's passed as the first argument divide exactly into the test value supplied as the second argument. The local variable in the function is declared in this statement:

```
  /* Upper limit */
  unsigned long long root_N = (unsigned long long)(1.0 + sqrt(N));
```

The integer variable, `root_N`, will hold the upper limit for divisors to be checked against the trial value. Only divisors less than the square root of the test value `N` are tried.

The checking is done in the `for` loop:

```
  for(size_t i = 0 ; i<count ; i++)
  {
    if(N % buffer[i] == 0UL )               /* Exact division?             */
      return 0;                             /* Then not a prime            */

    if(buffer[i] > root_N)                  /* Divisor exceeds square root? */
      return 1;                             /* Then must be a prime        */
  }
```

This loop steps through each of the divisors in the `buffer` array. If an exact divisor for `N` is found, the function will end and return 0 to indicate that the value isn't prime. If you arrive at a divisor that's greater than `root_N`, you've tried all those lower than this value, so `N` must be prime and the function returns 1. If the loop ends without executing a return statement then you haven't found an exact divisor but you haven't tried all values up to `root_N`. In this case the function returns –1 to indicate there's more checking to be done.

The last function that you need to define will output all the primes to the file:

```
void put_primes(void)
{
  unsigned long long buffer[MEM_PRIMES];      /* Buffer for a block of primes */

  if(!(global.pfile = fopen( global.filename, "rb"))) /* Open the file        */
  {
    printf("\nUnable to open %s to read primes for output\n", global.filename);
    exit(1);
  }

  for (size_t i = 0U ; i< global.nrec ; i++)
  {
    /* Read a block of primes   */
    fread(buffer, sizeof( unsigned long long), MEM_PRIMES, global.pfile);

    for(size_t j = 0 ; j<MEM_PRIMES ; j++)            /* Display the primes */
    {
      if(j%5 == 0U)                                   /* Five to a line     */
        printf("\n");
      printf("%12llu", buffer[j]);                    /* Output a prime     */
    }
  }
  fclose(global.pfile);                               /* Close the file     */
}
```

The operation of the `put_primes()` function is very simple. Once the file is opened, blocks of primes are read into the `buffer` array and as each is read, the `for` loop outputs the values to the screen, five to a line with a field width of 12. After all records have been read, the file is closed.

To run the program, you need to assemble all the functions that you've described into a single file and compile it. You'll be able to get as many primes as your computer and your patience permit.

A disadvantage of this program is that when you have a large number of primes, the output whizzes by on the screen before you can inspect it. You can do several things to fix this. You can write the output to the printer for a permanent record, instead of writing it to the screen. Or perhaps you can arrange for the program to display a prompt and wait for the user to press a key, between the output of one block and the next.

Moving Around in a File

For many applications, you need to be able to access data in a file other than in the sequential order you've used up to now. You can always find some information that's stored in the middle of a file by reading from the beginning and continuing in sequence until you get to what you want. But if you've written a few million items to the file, this may take some time.

Of course, to access data in random sequence requires that you have some means of knowing where the data that you would like to retrieve is stored in the file. Arranging for this is a complicated topic in general. There are many different ways of constructing pointers or indexes to make direct access to the data in a file faster and easier. The basic idea is similar to that of an index to a book. You have a table of keys that identify the contents of each record in the file you might want, and each key has an associated position in the file defined that records where the data is stored.

Let's look at the basic tools in the library that you need to enable you to deal with this kind of file input/output.

■ **Note** You cannot update a file in append mode. Regardless of any operations you may invoke to move the file position, all writes will be to the end of the existing data.

File Positioning Operations

There are two aspects to file positioning: finding out where you are at a given point in a file, and moving to a given point in a file. The former is basic to the latter: if you never know where you are, you can never decide how to get to where you want to go.

A random position in a file can be accessed regardless of whether the file concerned was opened in binary mode or in text mode. However, working with text mode files can get rather complicated in some environments, particularly Microsoft Windows. This is because the number of characters recorded in the file can be greater than the number of characters you actually write to the file. This is because a newline (`'\n'` character) in memory can translate into two characters when written to a file in text mode (carriage return, `CR`, followed by linefeed, `LF`). Of course, your C library function for reading the information sorts everything out when you read the data back. A problem only arises when you think that a point in the file is 100 bytes from the beginning. Whether writing 100 characters to a file in text mode results in 100 bytes actually appearing in the file depends on whether the data includes newline characters. If you subsequently want to write some different data that is the same length in memory as the original data written to the file, it will only be the same length on the file if it contains the same number of `'\n'` characters.

Thus writing to text files randomly is best avoided. For this reason, I'll sidestep the complications of moving about in text files and concentrate the examples on the much more useful—and easier—context of randomly accessing the data in binary files.

Finding Out Where You Are

You have two functions to tell you where you are in a file, both of which are very similar but not identical. They each complement a different positioning function. The first is the function `ftell()`, which has the prototype

```
long ftell(FILE *pfile);
```

This function accepts a file pointer as an argument and returns a `long` integer value that specifies the current position in the file. This could be used with the file that's referenced by the pointer `pfile` that you've used previously, as in the following statement:

```
fpos = ftell(pfile);
```

The `fops` variable of type `long` now holds the current position in the file and, as you'll see, you can use this in a function call to return to this position at any subsequent time. The value is actually the offset in bytes from the beginning of the file.

The second function providing information on the current file position is a little more complicated. The prototype of the function is the following:

```
int fgetpos(FILE *pfile, fpos_t *position);
```

The first parameter is your old friend the file pointer. The second parameter is a pointer to a type that's defined in `<stdio.h>` called `fpos_t`. `fpos_t` will be a type other than an array type that is able to record every position within a file. It is typically an integer type and with my library it is type `long`. If you're curious about what type `fpos_t` is on your system, then have a look at it in `<stdio.h>`.

The `fgetpos()` function is designed to be used with the positioning function `fsetpos()`, which I'll come to very shortly. The function `fgetpos()` stores the current position and file state information for the file in `position` and returns 0 if the operation is successful; otherwise, it returns a nonzero integer value. You could declare a variable `here` to be of type `fpos_t` with a statement such as this:

```
fpos_t here = 0;
```

You could now record the current position in the file with the statement

```
fgetpos(pfile, &here);
```

This records the current file position in the variable `here` that you have defined.

■ **Caution** Note that you must declare a variable of type `fpos_t`. It's no good just declaring a pointer of type `fpos_t*`, as there won't be any memory allocated to store the position data.

Setting a Position in a File

As a complement to `ftell()`, you have the function `fseek()`, which has the following prototype:

```
int fseek(FILE *pfile, long offset, int origin);
```

The first parameter is a pointer to the file that you're repositioning. The second and third parameters define where you want to go in the file. The second parameter is an offset from a reference point specified by the third parameter. The reference point can be one of three values that are specified by the predefined names `SEEK_SET`, which defines the beginning of the file; `SEEK_CUR`, which defines the current position in the file; and `SEEK_END`, which, as you might guess, defines the end of the file. Of course, all three values are defined in the header file `<stdio.h>`. For a text mode file, the