second argument must be a value returned by ftell() if you're to avoid getting lost. The third argument for text mode files must be SEEK_SET. So for text mode files, all operations with fseek() are performed with reference to the beginning of the file.

For binary files, the offset argument is simply a relative byte count. You can therefore supply positive or negative values for the offset when the reference point is specified as SEEK_CUR.

You have the fsetpos() function to go with fgetpos(). This has the rather straightforward prototype

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

The first parameter is a pointer to the file opened with fopen(), and the second is a pointer of the type you can see, where the value was obtained by calling fgetpos().

You can't go far wrong with this one really. You could use it with a statement such as this:

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

The variable here was previously set by a call to fgetpos(). As with fgetpos(), a nonzero value is returned on error. Because this function is designed to work with a value that is returned by fgetpos(), you can only use it to get to a place in a file that you've been before, whereas fseek() allows you to go to any specific position.

Note that the verb **seek** is used to refer to operations of moving the read/write heads of a disk drive directly to a specific position in the file. This is why the function fseek() is so named.

With a file that you've opened for update by specifying the mode as "rb+" or "wb+", for example, either a read or a write may be safely carried out on the file after executing either of the file positioning functions, fsetpos() or fseek(). This is regardless of what the previous operation on the file was.

## TRY IT OUT: RANDOM ACCESS TO A FILE

To exercise your newfound skills with files, you can create a revised version of Program 11.6 from the previous chapter to allow you to keep a dossier on family members. In this case, you'll create a file containing data on all family members, and then you'll process the file to output data on each member and that member's parents. The structures used only extend to a minimum range of members in each case. You can, of course, embellish these to hold any kind of scuttlebutt you like on your relatives.

Let's look at the function main() first:

```
/* Program 12.5 Investigating the family.*/
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

/* Global Data */
const int NAME_MAX = 20;

struct
{
  char *filename;                          /* Physical file name      */
  FILE *pfile;                             /* File pointer            */
} global = {"C:\\myfile.bin", NULL };
```

```
/* Structure types */
struct Date                                  /* Structure for a date        */
{
  int day;
  int month;
  int year;
};

typedef struct family                        /* Structure for family member */
{
  struct Date dob;
  char name[NAME_MAX];
  char pa_name[NAME_MAX];
  char ma_name[NAME_MAX];
}Family;

/* Function prototypes */
bool get_person(Family *pfamily);               /* Input function           */
void getname(char *name);                       /* Read a name              */
void show_person_data(void);                    /* Output function          */
void get_parent_dob(Family *pfamily);           /* Function to find pa & ma */

int main(void)
{
  Family member;                              /* Stores a family structure */

  if(!(global.pfile = fopen(global.filename, "wb")))
  {
    printf("\nUnable to open %s for writing.\n", global.filename);
    exit(1);
  }

  while(get_person(&member))                    /* As long as we have input */
    fwrite(&member, sizeof member, 1, pfile);   /*    write it away         */

  fclose(global.pfile);                   /* Close the file now its written */

  show_person_data();                     /* Show what we can find out      */

  if(remove(global.filename))
    printf("\nUnable to delete %s.\n", global.filename);
  else
    printf("\nDeleted %s OK.\n", global.filename);
  return 0;
}
```

### How It Works

After the `#include` statements, you have the global variables and structure definitions. You've seen all of these in previous examples. `Family` isn't a variable, but it has been declared as a type name for the structure `family`. This will allow you to declare `Family` type objects without having to use the keyword `struct`. Following the structure declarations, you have the prototypes for the three functions that you're using in addition to `main()`.

Because you want to try out the file-positioning functions in addition to the basic file read and write operations, the example has been designed to exercise these as well. You get the input on one person at a time in the `get_person()` function, where the data is stored in the member structure object. You write each structure to the file as soon as it is received, and the input process ceases when the function `get_person()` returns 0.

When the `while` loop ends, the input file is closed and the function `show_person_data()` is called. You use the file position getting and setting functions within this function. Lastly, the file is deleted from the disk by the function `remove()`.

The code for the input function, `get_person()`, is as follows:

```
/* Function to input data on Family members */
bool get_person(Family *temp)
{
  static char more = '\0';                   /* Test value for ending input */

  printf("\nDo you want to enter details of a%s person (Y or N)? ",
                                      more != '\0'?"nother " : "" );
  scanf(" %c", &more);

  if(tolower(more) == 'n')
    return false;

  printf("\nEnter the name of the person: ");
  getname(temp->name);                           /* Get the person's name */
  printf("\nEnter %s's date of birth (day month year); ", temp->name);
  scanf("%d %d %d", &temp->dob.day, &temp->dob.month, &temp->dob.year);

  printf("\nWho is %s's father? ", temp->name);
  getname(temp->pa_name);                        /* Get the father's name  */
  printf("\nWho is %s's mother? ", temp->name);
  getname(temp->ma_name);                        /* Get the mother's name  */
return true;
}
```

This function is fairly self-explanatory. None of the mechanisms involved in this function is new to you. An indicator, `more`, controls whether reading data continues, and it's set by the input following the first prompt. It's defined as `static` so the variable and its value persists in the program from one call of `get_person()` to the next. This allows the prompt to work correctly in selecting a slightly different message for the second and subsequent iterations.

If no data input takes place, which is triggered when **N** or **n** is entered in response to the initial prompt in the function, `false` is returned. If more data entry occurs, it's entered into the appropriate structure members and `true` is returned.

The names are read by the `getname()` function that can be implemented like this:

```
/* Read a name from the keyboard */
void getname(char *name)
{
  fflush(stdin);                             /* Skip whitespace        */
  fgets(name, NAME_MAX, stdin);
  int len = strlen(name);
  if(name[len-1] == '\n')                    /* If last char is newline */
    name[len-1] = '\0';                      /* overwrite it           */
}
```

This reads a name using `fgets()` to allow whitespace in the input and also to ensure the capacity of the storage for a name is not exceeded. If the input exceeds `NAME_MAX` characters, including the terminating null, the name will be truncated. The `fgets()` function stores the newline that arises when the Enter key is pressed, so when this is present as the last string character, you overwrite it with a terminating null.

The next function generates the output for each person, including the date of birth of both parents, if they've been recorded. The code for this function is as follows:

```
/* Function to output data on people on file   */
void show_person_data(void)
{
  Family member;                      /* Structure to hold data from file  */
  fpos_t current = 0;                 /* File position                     */

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

  /* Read data on person */
  while(fread(&member, sizeof member, 1, global.pfile))
  {
    fgetpos(global.pfile, &current);   /* Save current position            */
    printf("\n\n%s's father is %s, and mother is %s.",
             member.name, member.pa_name, member.ma_name);
    get_parent_dob(&member);           /* Get parent data                  */
    fsetpos(global.pfile, &current);   /* Position file to read next       */
  }
   fclose(global.pfile);               /* Close the file                   */
}
```

This function processes each structure in sequence from the file.

After declaring a variable of type `Family`, you declare a variable, `current`, with the following statement:

```
  fpos_t current = 0;                  /* File position                     */
```

This statement declares `current` as type `fpos_t`. This variable will be used to remember the current position in the file. The `get_parent_dob()` function is called later in this function, which also accesses the file. It's therefore necessary to remember the file position of the next structure to be read on each iteration before calling `get_parent_dob()`.

After opening the file for binary read operations, all of the processing takes place in a loop controlled by the following:

```
  while(fread(&member, sizeof member, 1, global.pfile))
```

This uses the technique of reading the file within the loop condition and using the value returned by the function `fread()` as the determinant of whether the loop continues. If the function returns 1, the loop continues, and when 0 is returned, the loop is terminated.

Within the loop you have these statements:

```
    fgetpos(global.pfile, &current);   /* Save current position            */
    printf("\n\n%s's father is %s, and mother is %s.",
             member.name, member.pa_name, member.ma_name);
    get_parent_dob(&member);           /* Get parent data                  */
    fsetpos(global.pfile, &current);   /* Position file to read next       */
```

First, the current position in the file is saved, and the parents of the current person stored in `member` are displayed. You then call the function `get_parent_dob()`, which will search the file for parent entries. On returning after the call to this function, the file position is unknown, so a call to `fsetpos()` is made to restore it to the position required for the next structure to be read. After all the structures have been processed, the `while` loop terminates and the file is closed.

The function to find the dates of birth for the parents of an individual is as follows:

```
/* Function to find parents' dates of birth. */
void get_parent_dob(Family *pmember)
{
  Family relative;                          /* Stores a relative         */
  int num_found = 0;                        /* Count of relatives found  */

  rewind(global.pfile);                     /* Set file to the beginning */

  /* Get the stuff on a relative */
  while(fread(&relative, sizeof(Family), 1, global.pfile))
  {
    if(strcmp(pmember->pa_name, relative.name) == 0)   /*Is it pa?        */
    { /* We have found dear old dad */
      printf("\n Pa was born on %d/%d/%d.",
            relative.dob.day, relative.dob.month, relative.dob.year);

      if(++num_found == 2)                  /* Increment parent count    */
        return;                             /* We got both so go home    */
    }
    else
      if(strcmp(pmember->ma_name, relative.name) == 0) /*Is it ma?        */
      { /* We have found dear old ma */
        printf("\n Ma was born on %d/%d/%d.",
               relative.dob.day, relative.dob.month, relative.dob.year);

        if(++num_found == 2)                /* Increment parent count    */
           return;                          /* We got both so go home    */
      }
  }
}
```

As the file has already been opened by the calling program, it's only necessary to set it back to the beginning with the `rewind()` function before beginning processing. The file is then read sequentially, searching each structure that's read for a match with either parent name. The search mechanism for the father is contained in the following statements:

```
    if(strcmp(pmember->pa_name, relative.name) == 0)   /*Is it pa?        */
    { /* We have found dear old dad */
      printf("\n Pa was born on %d/%d/%d.",
            relative.dob.day, relative.dob.month, relative.dob.year);
```

```
      if(++num_found == 2)                    /* Increment parent count    */
        return;                               /* We got both so go home    */
    }
```

The name entry for the father of the person indicated by `pmember` is compared with the name member in the structure object `relative`. If the father check fails, the function continues with an identical mother check.

If a parent is found, the date of birth information is displayed. A count is kept in `num_found` of the number of parents discovered in the file, and the function is exited if both have been found. The function ends in any event after all structures have been read from the file.

To run this program, you need to assemble the code for `main()` and the other functions into a single file. You can then compile and execute it. Of course, the example could be written equally well using `ftell()` and `fseek()` as positioning functions.

As in the previous examples in this chapter, the program uses a specific file name, on the assumption that the file doesn't already exist when the program is run. There's a way in C to create temporary files that get around this so let's look into that next.

# Using Temporary Work Files

Very often you need a work file just for the duration of a program. You use it only to store intermediate results and you can throw it away when the program is finished. The program that calculates primes in this chapter is a good example; you really only need the file during the calculation.

You have a choice of two functions to help with temporary file usage, and each has advantages and disadvantages.

## Creating a Temporary Work File

The first function will create a temporary file automatically. Its prototype is the following:

```
FILE *tmpfile(void);
```

The function takes no arguments and returns a pointer to the temporary file. If the file can't be created for any reason—for example, if the disk is full—the function returns `NULL`. The file is created and opened for update, so it can be written and read, but obviously it needs to be in that order. You can only ever get out what you have put in. The file is automatically deleted on exit from your program, so there's no need to worry about any mess left behind. You'll never know what the file is called, and because it doesn't last this doesn't matter.

The disadvantage of this function is that the file will be deleted as soon as you close it. This means you can't close the file, having written it in one part of the program, and then reopen it in another part of the program to read the data. You must keep the file open for as long as you need access to the data. A simple illustration of creating a temporary file is provided by these statements:

```
FILE pfile;                           /* File pointer                 */
pfile = tmpfile();                    /* Get pointer to temporary file */
```

## Creating a Unique File Name

The second possibility is to use a function that provides you with a unique file name. Whether this ends up as the name of a temporary file is up to you. The prototype for this function is the following:

```
char *tmpnam(char *filename);
```

If the argument to the function is NULL, the file name is generated in an internal static object, and a pointer to that object is returned. If you want the name stored in a char array that you declare yourself, it must be at least L_tmpnam characters long, where L_tmpnam is an integer constant that is defined in <stdio.h>. In this case, the file name is stored in the array that you specify as an argument, and a pointer to your array is also returned. If the function is unable to create a unique name, it will return NULL.

So to take the first possibility, you can create a unique file with the following statements:

```
FILE *pFile = NULL;
char *filename = tmpnam(NULL);
if(filename != NULL)
  pfile = fopen(filename, "wb+");
```

Here you declare your file pointer pfile and then your pointer filename that is initialized with the address of the temporary file name that the tmpnam() function returns. Because the argument to tmpnam() is NULL, the file name will be generated as an internal static object whose address will be placed in the pointer filename. As long as filename is not NULL you call fopen() to create the file with the mode "wb+". Of course, you can also create temporary text files, too.

Don't be tempted to write this:

```
pfile = fopen(tmpnam(NULL), "wb+");    /* Wrong!! */
```

Apart from the fact there is a possibility that tmpnam() may return NULL, you also no longer have access to the file name, so you can't use remove() to delete the file.

If you want to create the array to hold the file name yourself, you could write this:

```
FILE *pfile = NULL;
char filename[L_tmpnam];
if(tmpnam(filename) != NULL)
  pfile = fopen(filename, "wb+");
```

Remember, the assistance that you've obtained from the standard library is just to provide a unique name. It's your responsibility to delete any files created.

---

■ **Note** You should note that you'll be limited to a maximum number of unique names from this function in your program. You can access the maximum number through TMP_MAX that is defined in <stdio.h>.

---

# Updating Binary Files

You have three open modes that provide for updating binary files:

- The mode "r+b" (or you can write it as "rb+") opens an existing binary file for both reading and writing. With this open mode you can read or write anywhere in the file.
- The mode "w+b" (or you can write "wb+") truncates the length of an existing binary file to zero so the contents will be lost; you can then carry out both read and write operations but, obviously because the file length is zero, you must write something before you can read the file. If the file does not exist, a new file will be created when you call fopen() with mode "w+b".
- The third mode "a+b" (or "ab+") opens an existing file for update. This mode only allows write operations at the end of the file.

While you can write each of the open modes for updating binary files in either of two ways, I prefer to always put the + at the end because for me it is more obvious that the + is significant and means update. We can first put together an example that uses mode "wb+" to create a new file that we can then update using the other modes.

## TRY IT OUT: WRITING A BINARY FILE WITH AN UPDATE MODE

The file will contain names of people and their ages, the data being read from the keyboard. A name will be stored as a single string containing a first name and a second name. I have specified a full file path to the temp directory on drive C: in the code so you should check that this will be satisfactory on your system or change it to suit your environment. Note that if the directory in the path does not exist, the program will fail. Here's the code for the example:

```
/* Program 12.6 Writing a binary file with an update mode */
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

const int MAXLEN = 30;                    /* Size of name buffer    */

void listfile(char *filename);            /* List the file contents */

int main(void)
{
  const char *filename = "C:\\temp\\mydata.bin";
  char name[MAXLEN];                          /* Stores a name    */
  size_t length = 0;                          /* Length of a name */
  int age = 0;                                /* Person's age     */
  char answer = 'y';

  FILE *pFile = fopen(filename, "wb+");

  do
  {
    fflush(stdin);                            /* Remove whitespace */

    printf("\nEnter a name less than %d characters:", MAXLEN);
    gets(name);                               /* Read the name     */

    printf("Enter the age of %s: ", name);
    scanf(" %d", &age);                       /* Read the age      */

    /* Write the name & age to file */
    length = strlen(name);                    /* Get name length   */
    fwrite(&length, sizeof(length), 1, pFile);  /* Write name length */
    fwrite(name, sizeof(char), length, pFile);  /* then the name     */
    fwrite(&age, sizeof(age), 1, pFile);        /* then the age      */

    printf("Do you want to enter another(y or n)?  " );
    scanf("\n%c", &answer);
  } while(tolower(answer) == 'y');
```

```
  fclose(pFile);                              /* Close the file    */

  listfile(filename);                         /* List the contents */
  return 0;
}

/* List the contents of the binary file */
void listfile(char *filename)
{
  size_t length = 0;                          /* Name length       */
  char name[MAXLEN];                          /* Stores a name     */
  int age = 0;
  char format[20];                            /* Format string     */

  /* Create the format string for names up to MAXLEN long */
  sprintf(format, "\n%%-%ds Age:%%4d", MAXLEN);

  FILE *pFile = fopen(filename, "rb");        /* Open to read      */
  printf("\nThe contents of %s are:", filename);

  /* Read records as long as we read a length value */
  while(fread(&length, sizeof(length), 1, pFile) == 1)
  {
    if(length+1>MAXLEN)
    {
      printf("\nName too long.");
      exit(1);
    }
    fread(name, sizeof(char), length, pFile);   /* Read the name     */
    name[length] = '\0';                        /* Append terminator */
    fread(&age, sizeof(age), 1, pFile);         /* Read the age      */
    printf(format, name, age);                  /* Output the record */
  }
  fclose(pFile);
}
```

Here's some sample output from this program:

```
Enter a name less than 30 characters:Bill Bloggs
Enter the age of Bill Bloggs: 21
Do you want to enter another(y or n)?  y

Enter a name less than 30 characters:Yolande Dogsbreath
Enter the age of Yolande Dogsbreath: 27
Do you want to enter another(y or n)?  y

Enter a name less than 30 characters:Ned Nudd
Enter the age of Ned Nudd: 33
Do you want to enter another(y or n)?  y

Enter a name less than 30 characters:Binkie Huckerback
Enter the age of Binkie Huckerback: 18
Do you want to enter another(y or n)?  y
```

```
Enter a name less than 30 characters:Mary Dunklebiscuit
Enter the age of Mary Dunklebiscuit: 29
Do you want to enter another(y or n)?  n

The contents of C:\temp\mydata.bin are:
Bill Bloggs                    Age:  21
Yolande Dogsbreath             Age:  27
Ned Nudd                       Age:  33
Binkie Huckerback              Age:  18
Mary Dunklebiscuit             Age:  29
```

### How It Works

The file is opened for binary update operations with the mode specified as `"rb+"`. In this mode the file contents will be overwritten because the file length is truncated to zero. If the file does not exist, a file will be created. The data is read from the keyboard and the file is written in the `do-while` loop. The first statement in the loop flushes `stdin`:

```
fflush(stdin);                                /* Remove whitespace */
```

This is necessary because the read operation for a single character that appears in the loop condition will leave a newline character in `stdin` on all loop iterations after the first. If you don't get rid of this character, the read operation for the name will not work correctly because the newline will be read as an empty name string.

After reading a name and an age from the keyboard, the information is written to the file as binary data with these statements:

```
length = strlen(name);                      /* Get name length   */
fwrite(&length, sizeof(length), 1, pFile);  /* Write name length */
fwrite(name, sizeof(char), length, pFile);  /* then the name     */
fwrite(&age, sizeof(age), 1, pFile);        /* then the age      */
```

The names are going to vary in length and you have basically two ways to deal with this. You can write the entire name array to the file each time and not worry about the length of a name string. This is simpler to code but means that there would be a lot of spurious data in the file. The alternative is to adopt the approach used in the example. The length of each name string is written preceding the name itself, so to read the file you will first read the length, then read that number of characters from the file as the name. Note that the `'\0'` string terminator is not written to the file, so you have to remember to add this at the end of each name string when you read the file back.

The loop allows as many records as you want to be added to the file because it continues as long as you enter `'y'` or `'Y'` when prompted. When the loop ends you close the file and call the `listfile()` function that lists the contents of the file on `stdout`.

The `listfile()` function opens the file for binary read operations with the mode `"rb"`. In this mode the file pointer will be positioned at the beginning of the file and you can only read it.

The maximum length of a name is specified by the `MAXLEN` symbol so it would be helpful to use the format `%-MAXLENs` for outputting names. This would output a name left-justified in a field that has a width that is the maximum name length, so the names would line up nicely and they would always fit in the field. Of course, you can't really write this as part of the format string because the letters in the `MAXLEN` symbol name would be interpreted as just that, a sequence of letters, and not the value of the symbol. To achieve the required result, the `listfile()` function uses the `sprintf()` function to write to the `format` array to create a format string:

```
sprintf(format, "\n%%-%ds Age:%%4d", MAXLEN);
```

The `sprintf()` function works just like `printf()` except that the output is written to an array of `char` elements that you specify as the first argument. This operation therefore writes the value of `MAXLEN` to the `format` array, using the format string:

```
"\n%%-%ds Age:%%4d"
```

After the \n for a newline there is %% which specifies a single % symbol in the output. The - will appear next in the output followed by the value of MAXLEN formatted using the %d specification. This will be followed by s, then a space followed by Age:. Finally the output will contain a % character followed by 4d. Because the MAXLEN symbol is defined as 30, after executing the sprintf() function the format array will contain the following string:

```
"\n%-30s Age:%d"
```

The file is read and the contents listed on stdout in the while loop that is controlled by the value of an expression that reads the name length from the file:

```
while(fread(&length, sizeof(length), 1, pFile) == 1)
{
...
}
```

The call to fread() reads one item of sizeof(length) bytes into the location specified by &length. When the operation is successful the fread() function returns the number of items read but when the end-of-file is reached the function will return less than the number requested because there is not more data to be read. Thus when we reach the end of file, the loop will end.

An alternative way of recognizing when the end-of-file is reached is to code the loop like this:

```
while(true)
{
  fread(&length, sizeof(length), 1, pFile);
  /* Now check for end of file */
  if(feof(pFile))
    break;
...
}
```

The feof() function tests the end-of-file indicator for the stream specified by the argument and returns nonzero if the indicator is set. Thus when end-of-file is reached, the break will be executed and the loop will end.

After reading the length value from the file, you check that you have space to accommodate the name that follows with the following statements:

```
if(length+1>MAXLEN)
{
  printf("\nName too long.");
  exit(1);
}
```

Remember that the name in the file does not have a terminating '\0' character so you have to allow for that in the name array. Hence you compare length+1 with MAXLEN.

You read the name and age from the file with these statements:

```
fread(name, sizeof(char), length, pFile);   /* Read the name     */
name[length] = '\0';                        /* Append terminator */
fread(&age, sizeof(age), 1, pFile);         /* Read the age      */
```

Finally in the loop, you write the name and age to stdout using the format string that you created using the sprintf() function.

```
printf(format, name, age);                  /* Output the record */
```

## Changing the File Contents

We could revise and extend the previous example so that it uses the other two binary update modes. Let's add capability to update the existing records in the file as well as add records or delete the file. This program will be rather more complicated so it will be helpful to break the operations down into more functions. We will still write the file so the names are recorded as they are, so the records consisting of a name and an age will vary in length. This will provide an opportunity to see some of the complications this introduces when we want to change the contents of the file.

To give you an idea of where we are headed, let's look at the program in outline. The program will consist of the following nine functions:

`main()`: Controls overall operation of the program and allows the user to select from a range of operations on the file.

`listfile()`: Outputs the contents of the file to `stdin`.

`writefile()`: Operates in two modes, either writes a new file with records read from `stdin`, or appends a record to the existing file.

`getrecord()`: Reads a record from `stdin`.

`getname()`: Reads a name from `stdin`.

`writerecord()`: Writes a record to the file.

`readrecord()`: Reads a record from the file.

`findrecord()`: Find the record in the file with a name that matches input.

`duplicatefile()`: Reproduces the file replacing a single updated record. This function is used to update a record when the new record will be a different length from the record being replaced.

Figure 12-5 shows the call hierarchy for the functions in the application.

The three functions called by `main()` implement the basic functionality of the program. The functions to the right of these three provide functionality that helps to simplify the three primary functions.

It will simplify the code if we define a structure that we can use to pass a name and age between functions:

```
struct Record
{
  char name[MAXLEN];
  int age;
};
```

We could easily write objects of type `Record` to the file, but this would mean the whole `name` array of `MAXLEN` elements would be written each time, so the file would contain a lot of spurious bytes for names shorter than `MAXLEN` characters. However, the structure will provide a very convenient way of passing a name and the associated age value to a function. There will be several functions in the new example, so let's first look at the code for each function before we put the example together. You can assemble the code for the functions into a single source file as you read through the following sections.

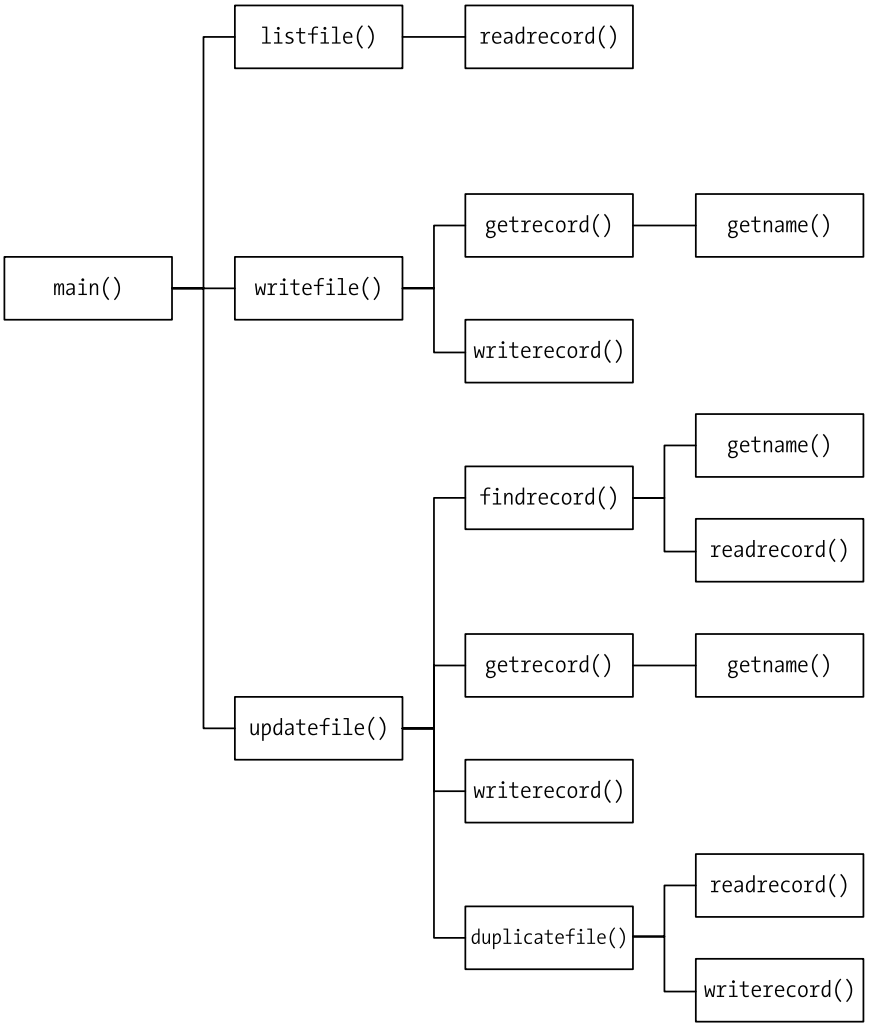


**Figure 12-5.** *The hierarchy of function calls in Program 12.8*

## Reading a Record from the Keyboard

We can write a function that will read a name string and age value from `stdin` and store them in a `Record` object. The prototype of the function will be the following:

```
struct Record *getrecord(struct Record *precord);
```

The function requires an argument that is a pointer to an existing `Record` structure object and it returns the address of the same object. By returning the address of the `Record` object you make it possible to use a call to this function as an argument to another function that expects an argument of type `Record *`.

Here's how the implementation of the function looks:

```
/* Read the name and age for a record from the keyboard */
struct Record *getrecord(struct Record *precord)
{
  /* Verify the argument is good */
  if(!precord)
  {
    printf("No Record object to store input.");
    return NULL;
  }

  printf("\nEnter a name less than %d characters:", MAXLEN);
  getname(precord->name);                    /* readf the name    */

  printf("Enter the age of %s: ", precord->name);
  scanf(" %d", &precord->age);               /* Read the age      */
  return precord;
}
```

This is a straightforward operation where the name and age that are read from `stdin` are stored in the appropriate members of the `Record` object that is pointed to by `precord`. The name is read by the auxiliary function `getname()` that you can implement like this:

```
/* Read a name from the keyboard */
void getname(char *pname)
{
  fflush(stdin);
  fgets(pname, MAXLEN, stdin);        /* Read the name     */
  int len = strlen(pname);
  if(pname[len-1] == '\n')         /* if there's a newline */
    pname[len-1] = '\0';           /* overwrite it         */
}
```

The only slight complication in `getname()` is the need to deal with the `'\n'` that may be stored by the `fgets()` function. If the input exceeds `MAXLEN` characters then the `'\n'` will still be in the input buffer and not stored in the array pointed to by `pname`. You'll need to read a name at more that one location in the program so packaging the operation in the `getname()` function is convenient.

## Writing a Record to a File

You can now define a function that will write the members of a record object to a file identified by a pointer of type `FILE *`. The prototype would look like this:

```
void writerecord(struct Record *precord, FILE *pFile);
```

The first parameter is a pointer to a `Record` structure that has the name and age that are to be written to the file as members. The second argument is the file pointer.

The implementation looks like this:

```
/* Write a new record to the file at the current position */
void writerecord(struct Record *precord, FILE *pFile)
{
```

```
  /* Verify the arguments are good */
  if(!precord)
  {
    printf("No Record object to write to the file.");
    return;
  }
  if(!pFile)
  {
    printf("No stream pointer for the output file.");
    return;
  }

  /* Write the name & age to file */
  size_t length = strlen(precord->name);                 /* Get name length   */
  fwrite(&length, sizeof(length), 1, pFile);             /* Write name length */
  fwrite(precord->name, sizeof(char), length, pFile);    /* then the name     */
  fwrite(&precord->age, sizeof(precord->age), 1, pFile); /* then the age      */
}
```

The function checks that the file pointer exists and the file will be written at the current position. It is therefore the responsibility of the calling function to ensure that the file has been opened in the correct mode and the file position has been set appropriately. The function first writes the length of the string to the file, followed by the string itself, excluding the terminating `'\0'`. This is to enable the code that will read the file to determine first how many characters are in the name string. Finally the age value is written to the file.

## Reading a Record from a File

Here's the prototype of a function to read a single record from a file:

```
struct Record *readrecord(struct Record *precord, FILE *pFile);
```

The file to be read is identified by the second parameter, a file pointer. Purely as a convenience, the return value is the address that is passed as the first argument.

The implementation of the `readrecord()` function looks like this:

```
/* Reads a record from the file at the current position */
struct Record * readrecord(struct Record *precord, FILE *pFile)
{
  /* Verify the arguments are good */
  if(!precord)
  {
    printf("No Record object to store data from the file.");
    return NULL;
  }
  if(!pFile)
  {
    printf("No stream pointer for the input file.");
    return NULL;
  }

  size_t length = 0;                                  /* Name length      */
  fread(&length, sizeof(length), 1, pFile);           /* Read the length  */
  if(feof(pFile))                                     /* If it's end file */
    return NULL;                                      /* return NULL      */
```

```
  /* Verify the name can be accommodated */
  if(length+1>MAXLEN)
  {
    fprintf(stderr, "\nName too long. Ending program.");
    exit(1);
  }

  fread(precord->name, sizeof(char), length, pFile);    /* Read the name     */
  precord->name[length] = '\0';                         /* Append terminator */
  fread(&precord->age, sizeof(precord->age), 1, pFile); /* Read the age      */

  return precord;
}
```

Like the `writerecord()` function, the `readrecord()` function assumes the file has been opened with the correct mode specified and by default attempts to read a record from the current position. Each record starts with a length value that is read first. Of course, the file position could be at the end of the file, so you check for `EOF` by calling `feof()` with the file pointer as the argument after the read operation. If it is the end-of-file, the `feof()` function returns a nonzero integer value, so in this case you return `NULL` to signal the calling program that `EOF` has been reached.

The function then checks for the possibility that the length of the name exceeds the length of the name array. If it does, the program ends after outputting a message to the standard error stream.

If all is well, the name and age are read from the file and stored in the members of the record object. A `'\0'` has to be appended to the `name` string to avoid disastrous consequences when working with the string subsequently.

## Writing a File

Here's the prototype of a function that will write an arbitrary number of records to a file, where the records are entered from the keyboard:

```
void writefile(char *filename, char *mode);
```

The first parameter is the name of the file to be written, so this implies that the function will take care of opening the file. The second parameter is the file open mode to be used. By passing `"wb+"` as the mode, the `writefile()` function will write to a file discarding any existing contents or create a new file with the specified name if it does not already exist. If the mode is specified as `"ab+"`, records will be appended to an existing file, and a new file will be created if there isn't one already.

Here's the implementation of the function:

```
/* Write to a file */
void writefile(char *filename, char *mode)
{
  char answer = 'y';

  FILE *pFile = fopen(filename, mode);    /* Open the file                */
  if(pFile == NULL)                       /* Verify file is open          */
  {
    fprintf(stderr, "\n File open failed.");
    exit(1);
  }
```

```
  do
  {
    struct Record record;                /* Stores a record name & age    */

    writerecord(getrecord(&record), pFile); /* Get record & write the file */

    printf("Do you want to enter another(y or n)?  " );
    scanf("\n%c", &answer);
    fflush(stdin);                       /* Remove whitespace             */
  } while(tolower(answer) == 'y');

  fclose(pFile);                         /* Close the file                */
}
```

After opening the file with the mode passed as the second argument, the function writes the file in the `do-while` loop. The read from `stdin` and the write to the file are done in the single statement that calls `writerecord()` with a call to `getdata()` as the first argument. The pointer to a `Record` object that `getdata()` returns is passed directly to the `writerecord()` function. The operation ends when the user enters `'n'` or `'N'` to indicate that no more data is to be entered. The file is closed before returning from the function.

## Listing the File Contents

The prototype of a function that will list the records in a file on the standard output stream looks like this:

```
void listfile(char *filename);
```

The parameter is the name of the file, so the function will take care of opening the file initially and then closing it when the operation is complete.

Here's the implementation:

```
/* List the contents of the binary file */
void listfile(char *filename)
{
  /* Create the format string for names up to MAXLEN long */
  /* format array length allows up to 5 digits for MAXLEN */
  char format[15];                       /* Format string              */
  sprintf(format, "\n%%-%ds Age:%%4d", MAXLEN);

  FILE *pFile = fopen(filename, "rb");   /* Open file to read          */
  if(pFile == NULL)                      /* Check file is open         */
  {
    printf("Unable to open %s. Verify it exists.\n", filename);
    return;
  }

  struct Record record;                  /* Stores a record            */
  printf("\nThe contents of %s are:", filename);

  while(readrecord(&record, pFile) != NULL)  /* As long as we have records */
    printf(format, record.name, record.age); /* Output the record        */
```

```
  printf("\n");                              /* Move to next line          */

  fclose(pFile);                             /* Close the file             */
}
```

The function generates a format string that will adjust the name field width to be `MAXLEN` characters. The `sprintf()` function writes the format string to the format array.

The file is opened in binary read mode so the initial position will be at the beginning of the file. If the file is opened successfully, records are read from the file in the `while` loop by calling the `readrecord()` function that we defined earlier. The call to `readrecord()` is done in the loop condition so when `NULL` is returned signaling end-of-file has been detected, the loop ends. Within the loop you write the members of the `Record` object that was read by `readrecord()` to `stdout` using the string in the `format` array that was created initially. When all the records have been read, the file is closed by calling `fclose()` with the file pointer as the argument.

## Updating the Existing File Contents

Updating existing records in the file adds a complication because of the variable length of the names in the file. You can't just arbitrarily overwrite an existing record because the chances are it won't fit in the space occupied by the record to be replaced. If the length of the new record is the same as the original, you can overwrite it. If they are different, the only solution is to write a new file. Here's the prototype of the function to update the file:

```
void updatefile(char *filename);
```

The only parameter is the file name, so the function will handle finding out which record is to be changed, as well as opening and closing the file. Here's the code:

```
/* Modify existing records in the file */
void updatefile(char *filename)
{  char answer = 'y';

  FILE *pFile = fopen(filename, "rb+");      /* Open the file for update   */
  if(pFile == NULL)                          /* Check file is open         */
  {
    fprintf(stderr, "\n File open for updating records failed.");
    return;
  }
  struct Record record;                      /* Stores a record            */
  int index = findrecord(&record, pFile);    /* Find the record for a name */
  if(index<0)                                /* If the record isn't there  */
  {
    printf("\nRecord not found.");           /* ouput a message            */
    return;                                  /* and we are done.           */
  }

  printf("\n%s is aged %d,", record.name, record.age);
  struct Record newrecord;                   /* Stores replacement record  */
  printf("\nYou can now enter the new name and age for %s.", record.name);
  getrecord(&newrecord);                     /* Get the new record         */
```

```
  /* Check if we can update in place */
  if((strlen(record.name) == strlen(newrecord.name)))
  { /* Name lengths are the same so we can */
    /* Move to start of old record         */
    fseek(pFile,
          -(long)(sizeof(size_t)+strlen(record.name)+sizeof(record.age)),
          SEEK_CUR);
    writerecord(&newrecord, pFile);          /* Write the new record       */
    fflush(pFile);                           /* Force the write            */
  }
  else
    duplicatefile(&newrecord, index, filename, pFile);

  printf("File update complete.\n");
}
```

There's quite a lot of code in this function but it consists of a sequence of fairly simple steps:

1. Open the file for update.
2. Find the index (first record at index 0) for the record to be updated.
3. Get the data for the record to replace the old record.
4. Check if the record can be updated in place. This is possible when the lengths of the names are the same. If so move the current position back by the length of the old record and write the new record to the old file.
5. If the names are different lengths duplicate the file with the new record replacing the old in the duplicate file.

After opening the file for update, the function reads the name corresponding to the record that is to be changed. The `findrecord()` function, which I'll get to in a moment, reads the name for the record to be updated, then returns the index value for that record, if it exists, with the first record at index 0. The `findrecord()` function will return –1 if the record is not found.

If the old and new names are the same length, move the file position back by the length of the old record by calling `fseek()`. Then write the new record to the file and flush the output buffer. Calling `fflush()` for the file forces the new record to be transferred from the file.

If the old and new records are different lengths, call `duplicatefile()` to copy the file with the new record replacing the old in the copy. You can implement the function like this:

```
/* Duplicate the existing file replacing the record to be update */
/* The record to be replaced is index records from the start     */
void duplicatefile(struct Record *pnewrecord, int index,
                                              char *filename, FILE *pFile)
{
    /* Create and open a new file */
  char tempname[L_tmpnam];
  if(tmpnam(tempname) == NULL)
  {
    printf("\nTemporary file name creation failed.");
    return;
  }
  char tempfile[strlen(dirpath)+strlen(tempname)+1];
  strcpy(tempfile, dirpath);                 /* Copy original file path  */
  strcat(tempfile, tempname);                /* Append temporary name    */
  FILE *ptempfile = fopen(tempfile, "wb+");
```

```
  /* Copy first index records from old file to new file */
  rewind(pFile);                          /* Old file back to start    */
  struct Record record;                   /* Store for a record        */
  for(int i = 0 ; i<index ; i++)
    writerecord(readrecord(&record, pFile), ptempfile);


  writerecord(pnewrecord, ptempfile);     /* Write the new record      */
  readrecord(&record,pFile);              /* Skip the old record       */

  /* Copy the rest of the old file to the new file */
  while(readrecord(&record,pFile))
    writerecord(&record, ptempfile);

  /* close the files */
  if(fclose(pFile)==EOF)
    printf("\n Failed to close %s", filename);
  if(fclose(ptempfile)==EOF)
    printf("\n Failed to close %s", tempfile);

  if(!remove(filename))                    /* Delete the old file       */
  {
    printf("\nRemoving the old file  failed. Check file in %s", dirpath);
    return;
  }

  /* Rename the new file same as original */
  if(!rename(tempfile, filename))
    printf("\nRenaming the file copy failed. Check file in %s", dirpath);
}
```

This function carries the update through the following steps:

1. Create a new file with a unique name in the same directory as the old file. The `dirpath` variable will be a global that contains the path to the original file.
2. Copy all records preceding the record to be changed from the old file to the new file.
3. Write the new record to the new file and skip over the record to be updated in the old file.
4. Write all the remaining records from the old file to the new file.
5. Close both files.
6. Delete the old file and rename the new file with the name of the old file.

Once the new file is created using the name generated by `tmpnam()`, records are copied from the original file to the new file, with the exception that the record to be updated is replaced with the new record in the new file. The copying of the first `index` records is done in the `for` loop where the pointer that is returned by `readrecord()` reading the old file is passed as the argument to `writerecord()` for the new file. The copying of the records that follow the updated record is done in the `while` loop. Here you have to continue copying records until the end-of-file is reached in the old file. Finally, after closing both files, delete the old file to free up its name and then rename the new file to the old. If you want to do this more safely, you can rename the old file in some way rather than deleting it, perhaps by appending `"_old"` to the existing file name. You can then rename the new file as you do here. This would leave a backup file in the directory that would be useful if the update goes awry.

The implementation of the `findrecord()` function that is called by `updatefile()` to find the index for the record that matches the name that is entered looks like this:

```
/* Find a record                          */
/* Returns the index number of the record */
/* or -1 if the record is not found.      */
int findrecord(struct Record *precord, FILE *pFile)
{
  char name[MAXLEN];
  printf("\nEnter the name for the record you wish to find: ");
  getname(name);

  rewind(pFile);                       /* Make sure we are at the start */
  int index = 0;                       /* Index of current record       */

  while(true)
  {
    readrecord(precord, pFile);
    if(feof(pFile))                     /* If end-of-file was reached    */
      return -1;                        /* record not found              */
    if(!strcmp(name, precord->name))
      break;
    ++index;
  }
  return index;                         /* Return record index           */
}
```

This function reads a name for the record that is to be changed, then reads records looking for a name that matches the name that was entered. If end-of-file is reached without finding the name, –1 is returned to signal to the calling program that the record is not in the file. If a name match is found, the function returns the index value of the matching record.

You can now assemble the complete working example.

## TRY IT OUT: READING, WRITING, AND UPDATING A BINARY FILE

I won't repeat all the functions I have just described. You can add the following to the beginning of the source file containing the code for the functions other than `main()`:

```
/* Program 12.7 Writing, reading and updating a binary file */
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

const int MAXLEN = 30;                          /* Size of name buffer     */
const char *dirpath = "C:\\temp\\";             /* Directory path for file */
const char *file = "mydata.bin";                /* File name               */

/* Structure encapsulating a name and age */
struct Record
{
```

```
  char name[MAXLEN];
  int age;
};

void listfile(char *filename);                    /* List the file contents   */
void updatefile(char *filename);                  /* Update the file contents */
struct Record *getrecord(struct Record *precord); /* Read a record from stdin */
void getname(char *pname);                        /* Read a name from stdin   */
void writefile(char *filename, char *mode);       /* Write records to a file  */
void writerecord(struct Record *precord, FILE *pFile);
struct Record * readrecord(struct Record *precord, FILE *pFile);
int findrecord(struct Record *precord, FILE *pFile);
void duplicatefile(struct Record *pnewrecord,
                                  int index, char *filename, FILE *pFile);
int main(void)
{
  char filename[strlen(dirpath)+strlen(file)+1];  /* Stores file path        */
  strcpy(filename, dirpath);                      /* Copy directory path     */
  strcat(filename, file);                         /*  and append file name   */

  /* Choose activity option */
  char answer = 'q';
  while(true)
  {
    printf("\nChoose from the following options:"
           "\nTo list the file contents enter  L"
           "\nTo create a new file enter       C"
           "\nTo add new records enter         A"
           "\nTo update existing records enter U"
           "\nTo delete the file enter         D"
           "\nTo end the program enter         Q\n : ");
    scanf("\n%c", &answer);

    switch(tolower(answer))
    {
      case 'l':                                  /* List file contents      */
        listfile(filename);
        break;
        case 'c':                                /* Create new file         */
        writefile(filename,"wb+");
        printf("\nFile creation complete.");
        break;
      case 'a':                                  /* Append records          */
        writefile(filename, "ab+");
        printf("\nFile append complete.");
        break;
      case 'u':                                  /* Update existing records */
        updatefile(filename);
        break;
      case 'd':
        printf("Are you sure you want to delete %s (y or n)? ", filename);
        scanf("\n%c", &answer);
        if(tolower(answer) == 'y')
          remove(filename);
```

```
        break;
      case 'q':                                /* Quit the program        */
        printf("\nEnding the program.", filename);
        return 0;
      default:
        printf("Invalid selection. Try again.");
        break;
    }
  }
  return 0;
}
```

Here's some sample output from a session using the main options offered by the program:

```
Choose from the following options:
To list the file contents enter  L
To create a new file enter       C
To add new records enter         A
To update existing records enter U
To delete the file enter         D
To end the program enter         Q
 : c

Enter a name less than 30 characters:Bill Bloggs
Enter the age of Bill Bloggs: 22
Do you want to enter another(y or n)?  y

Enter a name less than 30 characters:Kitty Malone
Enter the age of Kitty Malone: 23
Do you want to enter another(y or n)?  n

File creation complete.
Choose from the following options:
To list the file contents enter  L
To create a new file enter       C
To add new records enter         A
To update existing records enter U
To delete the file enter         D
To end the program enter         Q
 : l

The contents of C:\temp\mydata.bin are:
Bill Bloggs                    Age:  22
Kitty Malone                   Age:  23

Choose from the following options:
To list the file contents enter  L
To create a new file enter       C
To add new records enter         A
To update existing records enter U
To delete the file enter         D
To end the program enter         Q
 : a

Enter a name less than 30 characters:Jack Flash
Enter the age of Jack Flash: 30
Do you want to enter another(y or n)?  n
```

```
File append complete.
Choose from the following options:
To list the file contents enter  L
To create a new file enter       C
To add new records enter         A
To update existing records enter U
To delete the file enter         D
To end the program enter         Q
 : l

The contents of C:\temp\mydata.bin are:
Bill Bloggs                    Age:  22
Kitty Malone                   Age:  23
Jack Flash                     Age:  30

Choose from the following options:
To list the file contents enter  L
To create a new file enter       C
To add new records enter         A
To update existing records enter U
To delete the file enter         D
To end the program enter         Q
 : u

Enter the name for the record you wish to find: Kitty Malone

Kitty Malone is aged 23,
You can now enter the new name and age for Kitty Malone.
Enter a name less than 30 characters:Kitty Moline
Enter the age of Kitty Moline: 24
File update complete.

Choose from the following options:
To list the file contents enter  L
To create a new file enter       C
To add new records enter         A
To update existing records enter U
To delete the file enter         D
To end the program enter         Q
 : l

The contents of C:\temp\mydata.bin are:
Bill Bloggs                    Age:  22
Kitty Moline                   Age:  24
Jack Flash                     Age:  30

Choose from the following options:
To list the file contents enter  L
To create a new file enter       C
To add new records enter         A
To update existing records enter U
To delete the file enter         D
To end the program enter         Q
 : q
```

**How It Works**

There's a lot of code in `main` but it's very simple. The global strings `dirpath` and `file` identify the directory and file name respectively for the file that contains the data. These are concatenated in `main()` with the result stored in `filename`.

The indefinite `while` loop offers a series of choices of action and the choice entered is determined in the switch statement. Depending on the character entered, one of the functions you developed for the program is called. Execution continues until the option `'Q'` or `'q'` is entered to end the program.

# File Open Modes Summary

You probably will need a little practice before the file open mode strings come immediately to mind, so Table 12-2 contains a summary that you can refer back to when necessary.

**Table 12-2.** *File Modes*

| Mode | Description |
|---|---|
| `"w"` | Open or create a text file for **w**rite operations. |
| `"a"` | Open a text file for **a**ppend operations, adding to the end of the file. |
| `"r"` | Open a text file for **r**ead operations. |
| `"wb"` | Open or create a **b**inary file for **w**rite operations. |
| `"ab"` | Open a **b**inary file for **a**ppend operations. |
| `"rb"` | Open a **b**inary file for **r**ead operations. |
| `"w+"` | Open or create a text file for update operations. An existing file will be truncated to zero length. |
| `"a+"` | Open or create a text file for update operations, adding to the end of the file. |
| `"r+"` | Open a text file for update operations (read and write anywhere). |
| `"w+b"` or `"wb+"` | Open or create a binary file for update operations. An existing file will be truncated to zero length. |
| `"a+b"` or `"ab+"` | Open a binary file for update operations, adding to the end of the file. |
| `"r+b"` or `"rb+"` | Open a binary file for update operations (read and write anywhere). |

# Designing a Program

Now that you've come to the end of this chapter, you can put what you've learned into practice with a final program. This program will be shorter than the previous example, but nonetheless it's an interesting program that you may find useful.

# The Problem

The problem you're going to solve is to write a file-viewer program. This will display any file in hexadecimal representation and as characters.

# The Analysis

The program will open the file as binary read-only and then display the information in two columns, the first being the hexadecimal representation of the bytes in the file, and the second being the bytes represented as characters. The file name will be supplied as a command line argument or, if that isn't supplied, the program will ask for the file name.

The stages are as follows:

1. If the file name isn't supplied, get it from the user.
2. Open the file.
3. Read and display the contents of the file.

# The Solution

This section outlines the steps you'll take to solve the problem.

## Step 1

You can easily check to see if the file name appears at the command line by specifying that the function `main()` has parameters. Up until now, we have ignored the possibility of parameters being passed to `main()`, but here you can use it as the means of identifying the file that's to be displayed. You'll recall that when `main()` is called, two parameters are passed to it. The first parameter is an integer indicating the number of words in the command line, and the second is an array of pointers to strings. The first string contains the name that you use to start the program at the command line, and the remaining strings represent the arguments that follow at the command line. Of course, this mechanism allows an arbitrary number of values to be entered at the command line and passed to `main()`.

If the value of the first argument to `main()` is 1, there's only the program name on the command line, so in this case you'll have to prompt for the file name to be entered:

```
/* Program 12.8 Viewing the contents of a file */
#include <stdio.h>

const int MAXLEN = 256;                    /* Maximum file path length      */

int main(int argc, char *argv[])
{
  char filename[MAXLEN];                   /* Stores the file path          */

  if(argc == 1)                            /* No file name on command line? */
  {
    printf("Please enter a filename: ");   /* Prompt for input              */
    fgets(filename, MAXLEN, stdin);        /* Get the file name entered     */
```

```
    /* Remove the newline if it's there */
    int len = strlen(filename);
    if(filename[len-1] == '\n')
      filename[len-1] = '\0';
  }
  return 0;
}
```

This allows for a maximum file path length of 256 characters.

## Step 2

If the first argument to `main()` isn't 1, then you have at least one more argument, which you assume is the file name. You therefore copy the string pointed to by `argv[1]` to the variable `openfile`. Assuming that you have a valid file name, you can open the file and start reading it:

```
/* Program 12.8 Viewing the contents of a file */
#include <stdio.h>

const int MAXLEN = 256;                    /* Maximum file path length      */

int main(int argc, char *argv[])
{
  char filename[MAXLEN];                   /* Stores the file path          */
  FILE *pfile;                             /* File pointer                  */

  if(argc == 1)                            /* No file name on command line? */
  {
    printf("Please enter a filename: ");   /* Prompt for input              */
    fgets(filename, MAXLEN, stdin);        /* Get the file name entered     */

    /* Remove the newline if it's there */
    int len = strlen(filename);
    if(filename[len-1] == '\n')
      filename[len-1] = '\0';
  }
  else
    strcpy(filename, argv[1]);             /* Get 2nd command line string   */

  /* File can be opened OK? */
  if(!(pfile = fopen(filename, "rb")))
  {
    printf("Sorry, can't open %s", filename);
    return -1;
  }
  fclose(pfile);                           /* Close the file                */
  return 0;
}
```

You put the call to the `fclose()` function to close the file at the end of the program so that you don't forget about it later. Also, you use a return value of –1 for the program to indicate when an error has occurred.

## Step 3

You can now output the file contents. You do this by reading the file one byte at a time and saving this data in a buffer. Once the buffer is full or the end of file has been reached, you output the buffer in the format you want. When you output the data as characters, you must first check that the character is printable, otherwise strange things may start happening on the screen. You use the function `isprint()`, declared in `ctype.h`, for this. If the character isn't printable, you'll print a period instead.

Here's the complete code for the program:

```
/* Program 12.8 Viewing the contents of a file */
#include <stdio.h>
#include <ctype.h>
#include <string.h>

const int MAXLEN = 256;                  /* Maximum file path length      */
const int DISPLAY = 80;                  /* Length of display line        */
const int PAGE_LENGTH = 20;              /* Lines per page                */

int main(int argc, char *argv[])
{
  char filename[MAXLEN];                 /* Stores the file path          */
  FILE *pfile;                           /* File pointer                  */
  unsigned char buffer[DISPLAY/4 - 1];   /* File input buffer             */
  int count = 0;                         /* Count of characters in buffer */
  int lines = 0;                         /* Number of lines displayed     */

  if(argc == 1)                          /* No file name on command line? */
  {
    printf("Please enter a filename: "); /* Prompt for input              */
    fgets(filename, MAXLEN, stdin);      /* Get the file name entered     */

    /* Remove the newline if it's there */
    int len = strlen(filename);
    if(filename[len-1] == '\n')
      filename[len-1] = '\0';
  }
  else
    strcpy(filename, argv[1]);           /* Get 2nd command line string   */

    /* File can be opened OK?        */
    if(!(pfile = fopen(filename, "rb")))
    {
      printf("Sorry, can't open %s", filename);
      return -1;
    }
  while(!feof(pfile))                    /* Continue until end of file    */
  {
    if(count < sizeof buffer)            /* If the buffer is not full     */
      buffer[count++] = (unsigned char)fgetc(pfile);    /* Read a character */
    else
    { /* Output the buffer contents, first as hexadecimal */
      for(count = 0; count < sizeof buffer; count++)
        printf("%02X ", buffer[count]);
      printf("| ");                       /* Output separator              */
```

```
      /* Now display buffer contents as characters */
      for(count = 0; count < sizeof buffer; count++)
        printf("%c", isprint(buffer[count]) ? buffer[count]:'.');
      printf("\n");                          /* End the line                */
      count = 0;                             /* Reset count                 */

      if(!(++lines%PAGE_LENGTH))             /* End of page?                */
       if(getchar()=='E')                    /* Wait for Enter              */
         return 0;                           /* E pressed                   */
    }
  }

  /* Display the last line, first as hexadecimal */
  for(int i = 0; i < sizeof buffer; i++)
    if(i < count)
      printf("%02X ", buffer[i]);            /* Output hexadecimal          */
    else
      printf("   ");                         /* Output spaces               */
  printf("| ");                              /* Output separator            */

  /* Display last line as characters */
  for(int i = 0; i < count; i++)
    /* Output character    */
    printf("%c",isprint(buffer[i]) ? buffer[i]:'.');

  /* End the line */
  printf("\n");
  fclose(pfile);                             /* Close the file              */
  return 0;
}
```

The symbol `DISPLAY` specifies the width of a line on the screen for output, and the symbol `PAGE_LENGTH` specifies the number of lines per page. You arrange to display a page, and then wait for Enter to be pressed before displaying the next page, thus avoiding the whole file whizzing by before you can read it.

You declare the buffer to hold input from the file as

```
unsigned char buffer[DISPLAY/4 - 1];     /* File input buffer           */
```

The expression for the array dimension arises from the fact that you'll need four characters on the screen to display each character from the file, plus one separator. Each character will be displayed as two hexadecimal digits plus a space, and as a single character, making four characters in all.

You continue reading as long as the `while` loop condition is true:

```
while(!feof(pfile))                      /* Continue until end of file  */
```

The library function, `feof()`, returns `true` if `EOF` is read from the file specified by the argument; otherwise, it returns `false`.

You fill the `buffer` array with characters from the file in the `if` statement:

```
    if(count < sizeof buffer)            /* If the buffer is not full     */
      buffer[count++] = (unsigned char)fgetc(pfile);   /* Read a character  */
```

When `count` exceeds the capacity of `buffer`, the `else` clause will be executed to output the contents of the buffer:

```
else
{ /* Output the buffer contents, first as hexadecimal */
  for(count = 0; count < sizeof buffer; count++)
    printf("%02X ", buffer[count]);
  printf("| ");                          /* Output separator              */

  /* Now display buffer contents as characters */
  for(count = 0; count < sizeof buffer; count++)
    printf("%c", isprint(buffer[count]) ? buffer[count]:'.');
  printf("\n");                          /* End the line                  */
  count = 0;                             /* Reset count                   */

  if(!(++lines%PAGE_LENGTH))             /* End of page?                  */
   if(getchar()=='E')                    /* Wait for Enter                */
     return 0;                           /* E pressed                     */
}
```

The first `for` loop outputs the contents of `buffer` as hexadecimal characters. You then output a separator character and execute the next `for` loop to output the same data as characters. The conditional operator in the second argument to `printf()` ensures that nonprinting characters are output as a period.

The `if` statement increments the line count, `lines`, and for every `PAGE_LENGTH` number of lines, wait for a character to be entered. If you press Enter, the next pageful will be displayed, but if you press **E** and then Enter, the program will end. This provides you with an opportunity to escape from continuing to output the contents of a file that's larger than you thought.

The final couple of `for` loops are similar to those you've just seen. The only difference is that spaces are output for array elements that don't contain file characters. An example of the output is as follows. It shows part of the source file for the self-same program and you can deduce from the output that the file path was entered as a command line argument.

```
2F 2A 20 50 72 6F 67 72 61 6D 20 31 32 2E 38 20 56 69 65 | /* Program 12.8 Vie
77 69 6E 67 20 74 68 65 20 63 6F 6E 74 65 6E 74 73 20 6F | wing the contents o
66 20 61 20 66 69 6C 65 20 2A 2F 0D 0A 23 69 6E 63 6C 75 | f a file */..#inclu
64 65 20 3C 73 74 64 69 6F 2E 68 3E 0D 0A 23 69 6E 63 6C | de <stdio.h>..#incl
75 64 65 20 3C 63 74 79 70 65 2E 68 3E 0D 0A 23 69 6E 63 | ude <ctype.h>..#inc
6C 75 64 65 20 3C 73 74 72 69 6E 67 2E 68 3E 0D 0A 0D 0A | lude <string.h>....
63 6F 6E 73 74 20 69 6E 74 20 4D 41 58 4C 45 4E 20 3D 20 | const int MAXLEN =
32 35 36 3B 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 | 256;
20 20 20 20 20 2F 2A 20 4D 61 78 69 6D 75 6D 20 66 69 6C | /* Maximum fil
65 20 70 61 74 68 20 6C 65 6E 67 74 68 20 20 20 20 20 20 | e path length
2A 2F 0D 0A 63 6F 6E 73 74 20 69 6E 74 20 44 49 53 50 4C | */..const int DISPL
41 59 20 3D 20 38 30 3B 20 20 20 20 20 20 20 20 20 20 20 | AY = 80;
20 20 20 20 20 20 20 20 20 2F 2A 20 4C 65 6E 67 74 68 20 | /* Length
6F 66 20 64 69 73 70 6C 61 79 20 6C 69 6E 65 20 20 20 20 | of display line
20 20 20 20 2A 2F 0D 0A 63 6F 6E 73 74 20 69 6E 74 20 50 | */..const int P
41 47 45 5F 4C 45 4E 47 54 48 20 3D 20 32 30 3B 20 20 20 | AGE_LENGTH = 20;
20 20 20 20 20 20 20 20 20 20 20 20 20 2F 2A 20 4C 69 6E | /* Lin
65 73 20 70 65 72 20 70 61 67 65 20 20 20 20 20 20 20 20 | es per page
20 20 20 20 20 20 20 20 2A 2F 0D 0A 0D 0A 69 6E 74 20 6D | */....int m
61 69 6E 28 69 6E 74 20 61 72 67 63 2C 20 63 68 61 72 20 | ain(int argc, char
```

A lot more output follows, ending with this:

```
66 65 72 5B 69 5D 3A 27 2E 27 29 3B 0D 0A 20 20 2F 2A 20 | fer[i]:'.');..  /*
45 6E 64 20 74 68 65 20 6C 69 6E 65 20 20 20 20 20 20 20 | End the line
20 20 20 2A 2F 0D 0A 20 20 70 72 69 6E 74 66 28 22 5C 6E |    */..  printf("\n
22 29 3B 0D 0A 20 20 66 63 6C 6F 73 65 28 70 66 69 6C 65 | ");..  fclose(pfile
29 3B 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 | );
20 20 20 20 20 20 20 20 20 20 2F 2A 20 43 6C 6F 73 65 20 |           /* Close
74 68 65 20 66 69 6C 65 20 20 20 20 20 20 20 20 20 20 20 | the file
20 20 20 20 20 2A 2F 0D 0A 20 20 72 65 74 75 72 6E 20 30 |      */..  return 0
3B 0D 0A 7D 0D 0A 0D 0A FF                               | ;..}.....
```

# Summary

Within this chapter I've covered all of the basic tools necessary to provide you with the ability to program the complete spectrum of file functions. The degree to which these have been demonstrated in examples has been, of necessity, relatively limited. There are many ways of applying these tools to provide more sophisticated ways of managing and retrieving information in a file. For example, it's possible to write index information into the file, either as a specific index at a known place in the file, often the beginning, or as position pointers within the blocks of data, rather like the pointers in a linked list. You should experiment with file operations until you feel confident that you understand the mechanisms involved.

Although the functions I discussed in this chapter cover most of the abilities you're likely to need, you'll find that the input/output library provided with your compiler offers quite a few additional functions that give you even more options for handling your file operations.

# Exercises

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck, you can download the solutions from the Source Code/Download area of the Apress web site (`http://www.apress.com`), but that really should be a last resort.

**Exercise 12-1**. Write a program that will write an arbitrary number of strings to a file. The strings should be entered from the keyboard and the program shouldn't delete the file, as it will be used in the next exercise.

**Exercise 12-2.** Write a program that will read the file that was created by the previous exercise, and retrieve the strings one at a time in reverse sequence and write them to a new file in the sequence in which they were retrieved. For example, the program will retrieve the last string and write that to the new file, then retrieve the second to last and retrieve that from the file, and so on, for each string in the original file.

**Exercise 12-3.** Write a program that will read names and telephone numbers from the keyboard and write them to a new file if a file doesn't already exist and add them if the file does exist. The program should optionally list all the entries.

**Exercise 12-4.** Extend the program from the previous exercise to implement retrieval of all the numbers corresponding to a given second name. The program should allow further enquiries, adding new name/number entries and deleting existing entries.