File Routines: The First Step

WEEK
3

On Days 14 and 15, you added menus and action bars to your application. This helped the user to navigate effectively through your application. To make your application truly useful, you need to be able to work with data. Until today, your application has only been able to write data to a file. Today, you will expand on this. Today you will:

- ☐ Look at the different ways that you will want to access a file.
- ☐ Learn different file formats.
- ☐ Learn about indexing a file.
- ☐ Look at the functions that will enable you to work with the Medium Code file.

Storing Data in a Disk File

As you begin accepting data in your application, you'll want to be able to save it so that you can access it later. To do this, you use disk files. By storing information on a disk drive, you'll be able to access it at a later time.

Note: Many beginning C books cover disk files. In addition to covering such concepts as naming them, they also cover the basics of opening, reading, writing, and seeking within them. Much of this will be covered in today's material; however, many of the basics will be skimmed over. If you find the amount of information isn't adequate, then you should reference a beginning C book such as *Teach Yourself C in 21 Days.*

A Review of the Terminology Used with Files

Before examining the concepts involved in using data in a disk file, a few terms should be reviewed to ensure that you understand them. Many of these terms have been used throughout this book. The terms that you should be familiar with are I/O, fields, records, data files, and databases. Figure 17.1 helps to show the relationship of these terms.

I/O is an abbreviation for Input/Output. Many programmers use the term *I/O* to stand for any action that results in information flowing into, or out of, a computer program. Input is information flowing in; output is information flowing out. Information will flow into the *Record of Records!* application in two different ways. One is by being entered into the entry and edit screens. The other is by retrieving a record from the data file. Information is output in two ways also. It is output to the disk file when it is saved. It is also output to the reports that will be presented on Day 19.

A *field* is an individual piece of data. Because data is simply information, a field is simply a piece of information. In the *Record of Records!* application, there are a multitude of fields. Examples of some of the fields are medium code, group name, song title, and type of music.

A *record* is a group of related fields. In the *Record of Records!* application, there are several different records used. The smallest record used is the medium code record, which contains two fields, a medium code, and a medium description.

A *data file* is a group of related records that are stored on a secondary medium such as a disk drive. When you save records, they create files. In the *Record of Records!* application, you have three different data files. They are the Mediums, Groups, and Albums.

A *database* is a group of related data files. Often, the files in a database can be related to each other. For example, in the *Record of Records!* application, the Mediums file can be related to the Albums file via the medium code field. This Groups file can be related to the Albums file via the group name field. In both the examples, an identical field appears in both files. It is these fields that cause the files to be related.

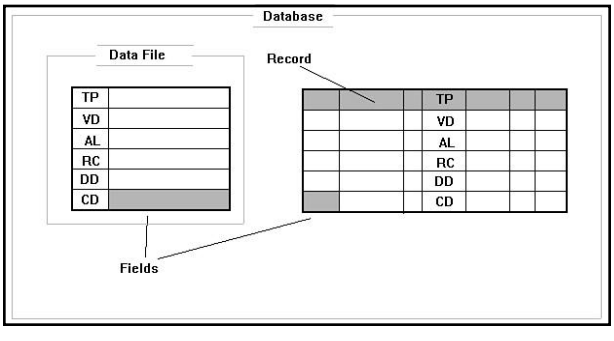

Figure 17.1. *The relationship between the file terms.*

File Formats

As stated before, files are stored on disks. In working with these disk files, there are two things that you must be aware of. The first is the mode of the file. The second is the format.

File Modes

There are two different modes for disk files. These are binary mode and text mode. When you initially open a file for use, you specify which of these two modes you will be using. Because the two modes operate differently, you should understand them.

If you are using text mode, then several translations will occur automatically as you work with the data. Because of this, the text mode is sometimes referred to as translated mode. Following are the translations:

- Carriage-return-line-feeds (CR-LF) are translated into a single line feed (LF) on input.
- A line-feed character (LF) is translated to a carriage-return-line-feed (CR-LF) on output.
- A Control+Z is translated as the end of the file. On input, the Control+Z character is removed (if possible).

A file opened in binary mode doesn't do the translations of a text mode file. In fact, a binary file doesn't do any translations. This means that data is stored exactly as it is.

When you open a file, you specify which mode you want to use, binary or text. In addition, you must specify the type of access that you will need with the file. There are several different accesses that can be specified. The modes that can be used are read (r), write (w), and append (a). If a file is opened to be read, then it cannot be written to or appended to. A file can be opened to both read and write by including a plus sign after the read (r) or write (w). If you open as w+ (write plus), then it will allow reading of and writing to a file. If the file already exists, it will be truncated to a zero length.

Note: Using the standard C file functions is covered in most beginning C books. File functions will be used and explained in some detail later. If you find that you need to know more, then you should consult either your compiler manuals or a beginning C book.

File Formats

In addition to different modes, files can also have different formats. The format of a file is how it is stored on the disk drive. Many C programmers learn to work with flat files; however, this is only one of many different formats of files. Today, you will be presented with three different formats; flat files, comma delimited files, and custom file formats.

Flat Files

With a flat file, information is written straight to the disk one record at a time. The text isn't provided with any special format. If you use an operating system command to list the file (such as the DOS TYPE command), you will see the raw data in the file.

In the *Record of Records!* application, each of the three entry and edit screens wrote the structures to the disk drive. These structures were creating a flat file on your disk. Each time you pressed the function key to add a record, a new record was added to the end of the flat file associated with that screen. You can list this information out. In doing so, you'll find that it is almost always readable. You'll also notice that each record takes the same amount of space. Following is an example of what a flat file might look like:

```
DDD12/25/98DIME DINNER DRINKS        000.10
SAK02/14/92SMILING APPLE KINGS       002.95
BEJ10/23/93BIG EAGLE JIGS            010.99
JMS03/13/93JAR MACARONI SALAD        004.95
```

Note: Numbers in a flat file may be stored as characters or as numbers. If they are stored as numbers, then they will appear differently than what was previously shown. For example, if you write a numeric value that was stored in an integer to a disk file, it would take the same space as an integer. If you wrote the number 65, it would be stored as "A"—the character representation (ASCII) of the number.

Comma Delimited Files

Comma delimited files are most often seen with older spreadsheet applications. Many commercial database applications will read in and write out comma delimited files. A comma delimited file writes a record to the file field-by-field. Each field is separated by a comma—hence the name. In addition, text fields are enclosed in quotes. Numeric fields are typically written as characters. This means that if you type a comma

delimited file, you'll know exactly what is in the file. Following is an example of what a portion of a listed comma delimited file may look like:

```
"DEANNA",35,"01/01/92","Body Building",
"CONNIE",21,"12/25/93","Square Dancing",
"JOE",54,"01/14/91","Knitting",
```

Custom File Formats

There are a multitude of custom file formats. Many of these formats have standards surrounding them. Among these standards are C-tree, Btrieve, dBase, and Vsam. These are just a few of the many different file formats.

There are several reasons to have a custom file format. The main reason is to protect your data from other programs. An additional reason for the creation of custom files is based on usage. Depending on how your application works, you may need to create special features in your data.

One common feature that is incorporated into many—but not all—file systems is indexing. Files can either contain built-in indexes or separate files can be created that index other files. The files that you add to the *Record of Records!* application will be a combination of flat files and index files.

Working with an Indexed File

Using an index allows quick access to a large data file. An index file is a separate file from the data file. Because an index contains a smaller amount of information, more of it can be read and manipulated quicker than the data file itself.

An index file generally contains the key field and a pointer. The key field is the field that will be used to sort the file. The pointer will contain the offset into the data file for the corresponding data record. Figure 17.2 illustrates a basic index and data file.

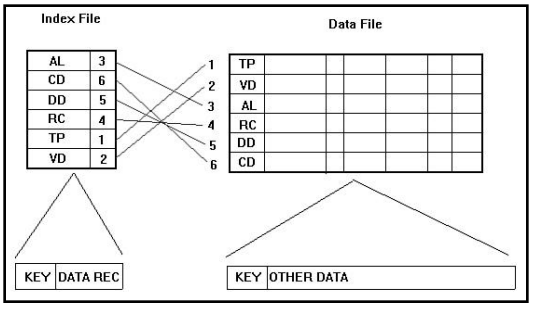

Figure 17.2. *An index file and a data file.*

Not all index files are the same. Index files can be created in two ways. They can be sorted or they can be linked. A sorted index file requires that every time you write a record to the index file, it is written in the correct order on the disk. This means that several records may need to be moved around each time a new index record is added. The index shown in Figure 17.2 was sorted.

Tip: If the index file will be small enough that you can read it into memory in its entirety, then using a sorted index can be time saving.

A linked index requires more work; however, it doesn't require that you do as much rewriting and moving of data records on the disk drive. A linked index works like a linked list. Using pointers, you keep track of the next index record. This is in addition to the key field and the pointer to the data record. If you will only be accessing your records in a single order, such as alphabetically, then you only need one additional pointer. If you will be navigating forward and backward through your data, then you'll need two pointers. Figure 17.3 illustrates a linked index that has the capability to navigate forward and backward.

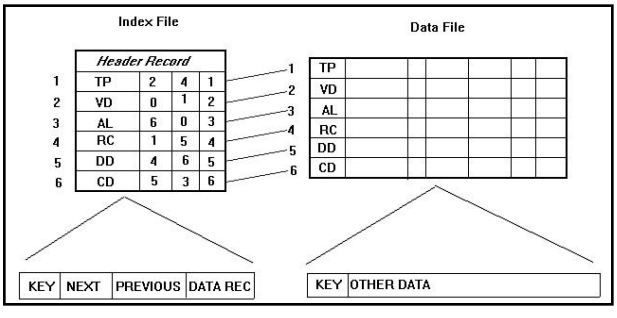

Figure 17.3. *A linked index.*

You should have noticed by now that the data file is stored independent of the indexed file. As new data file records are created, you simply add them to the end of the data file. The index file is then manipulated depending on whether you are using a sorted or linked index. In a linked index, you can also simply add the new index record to the end and then adjust the pointers accordingly. In the *Record of Records!* application, you will use a linked index containing the key field, a pointer to the corresponding data record, a pointer to the next record, and a pointer to the previous record.

Using a File

With the *Record of Records!* application, you'll need to be able to perform several file I/O functions. These I/O functions will use a data file and a linked index. The functions that you need are the following:

- ☐ Opening the file.
- ☐ Closing the file.
- ☐ Adding a record.
- ☐ Updating or changing a record.
- ☐ Deleting a record.
- ☐ Finding the next record.
- ☐ Finding the previous record.
- ☐ Finding a specific record.

Using the indexed file and the data file, you'll be able to do all of these tasks. Before detailing each of these, you first need to be aware of the structures that will be needed.

The medium file will have both an index file and a data file. When working with the *Record of Records!* application, you'll read and write directly from structures. The index file will be kept in one structure and the data will be kept in another structure. The data file's structure will be the same structure that you have been using all along:

```
typedef struct
{
   char code[2+1];
   char desc[35+1];

} MEDIUM_REC;
```

The index file will be a new structure that needs to be added to the RECORDS.H header file that contains all the structures. The index structure for the medium code screen should be:

```
typedef struct
{
   char code[2+1];
   unsigned short prev;
   unsigned short next;
   unsigned short data;

} MEDIUM_INDEX;
```

As you can see, this structure contains a field, `code`, for the key field. In addition, it contains three variables that will be used to hold record numbers. The record numbers will be for the previous index record, the next index record, and the address of the data in the data file. You will see each of these record numbers used as the file is manipulated.

There will be one additional piece of information needed. This is a header record for the index file. This header record was shown in Figure 17.3. This record will contain the address of the first sorted records in the index file so that you will know where to begin reading the index records. The header will also contain the number of records in the file. For the *Record of Records!* application, the header information will be read and written without the use of a structure. You will see this later.

Preparation for the New I/O Functions

Before jumping into the I/O functions, you should make some modifications to your TYAC.H header file. This header file should be modified to include several defined constants that will be used with the I/O functions. In the next sections, you will create several functions that will be added to your TYAC.LIB library. These functions will be generic I/O functions that will be used by the specific screens to work with the indexed data files.

Several defined constants will be used in the file I/O routines. These defined constants will be used to describe errors that may occur as you are doing file routines. You should add the following to your TYAC.H header file.

```
/* ------------------------------- *
 *     FILE I/O Errors/Flags       *
 * ------------------------------- */

#define NO_ERROR          00
#define OPEN_ERROR        01
#define CREATE_ERROR      02
#define USER_CREATE_ERR   03
#define SEEK_ERROR        04
#define WRITE_ERROR       05
#define READ_ERROR        06
#define CLOSE_ERROR       07
#define NOT_FOUND         08

#define PREVIOUS_REC      01
#define NEXT_REC          00

/* ------------------------------- *
 *   Numeric Conversion Flags      *
 * ------------------------------- */
```

```
#define PACK_ZERO        01    /* For packing numerals */
#define NO_PACK          00    /* w/ zeros at the front */
```

As you can see, most of these constants define errors. The PREVIOUS_REC and NEXT_REC will be used for moving through the database. The PACK_ZERO and NO_PACK constants are conversion flags that will be detailed later.

Before jumping into the specific changes for the *Record of Record!* application screens, the following are prototypes for functions that will be created and described.

```
int open_files(char *, char *);
int close_files(void);
int update_header(void);
int get_rec(int, FILE *, int, int, char * );
int put_rec(int, FILE *, int, int, char * );
```

You should add these prototypes to the TYAC.H header file. The following sections will present each of these functions.

Opening the FILES: *open_files()*

Before being able to work with files, you need to first open them. The `open_files()` function presented in Listing 17.1 opens both an index and a data file.

Type

Listing 17.1. OPENFILE.C. Opens an index and a data file.

```
1:  /* ----------------------------------------------------------
2:   * Program: OPENFILE.C
3:   * Authors: Bradley L. Jones
4:   *          Gregory L. Guntle
5:   *
6:   * Purpose: Opens a file - Sets two global variables
7:   *          nbr of rec in file and the starting record
8:   *          in alpha.
9:   *
10:  * Enter with:
11:  *             idx = Name of index file
12:  *             db  = Name of data file
13:  *
14:  * Returns:
15:  *          0 = No Error
16:  *          >0 = Error - see DEFINES in TYAC.H for FILE I/O
17:
18:  *=========================================================*/
19:
20:  #include <stdio.h>
```

```
21:  #include “tyac.h”
22:
23:  /*  Global Variables */
24:  extern FILE *idx_fp;
25:  extern FILE *db_fp;
26:  extern int nbr_records;
27:  extern int start_rec;
28:
29:
30:        /******************
31:         *   OPEN_FILES   *
32:         ******************/
33:  int open_files(char *idx, char *db)
34:  {
35:    int rv = NO_ERROR;
36:    int cr_flg = FALSE;  /* Assume file exist */
37:
38:    /* Open index file first */
39:    if( (idx_fp = fopen(idx, “r+b” )) == NULL )
40:    {
41:      /* Doesn’t exist - create it */
42:      if( (idx_fp = fopen(idx, “w+b”)) == NULL )
43:        rv = CREATE_ERROR;
44:      else
45:        cr_flg = TRUE;   /*Indicates no hdr exist yet */
46:    }
47:
48:    /* Open Database File - as long as no previous errors */
49:    if ( rv == NO_ERROR )
50:    {
51:      /* Open existing file */
52:      if( (db_fp = fopen(db, “r+b” )) == NULL )
53:      {
54:        /* Create new DB file */
55:        if( (db_fp = fopen(db, “w+b”)) == NULL )
56:           rv = CREATE_ERROR;
57:      }
58:    }
59:
60:    /* Only continue if no errors above */
61:    if ( rv==NO_ERROR)
62:    {
63:      /* Get number of records */
64:      if ( !cr_flg ) /* File exist - get hdr record */
65:      {
66:        rv = get_rec(0, idx_fp, sizeof(int), 0,
                    (char*)  &nbr_records);
67:
68:        /* Get starting record # */
69:        if (rv == 0)
```

continues

Listing 17.1. continued

```
70:         {
71:           rv = get_rec(0, idx_fp, sizeof(int), sizeof(int),
72:                        (char *) &start_rec);
73:         }
74:       }
75:       else
76:       {
77:         nbr_records = 0;      /* New file - no records yet */
78:         start_rec = 1;
79:       }
80:     }
81:
82:     return(rv);
83:   }
```

Analysis The `open_files()` function makes several assumptions. First, it assumes that you will have several external variables defined (lines 23 to 27). These external variables must be declared or else `open_files()` function will fail. Each of these variables is important. The `idx_fp` is a file pointer that will be used with for the index file. The `db_fp` is a file pointer that will be used for the data file. `nbr_records` will be filled with the number of records in the file pointed to be the `db_fp`. The `start_rec` variable will be used to contain the record number for the first sorted record, or the starting record. The starting record's number and the number of records will initially be obtained from the header record when the file is opened.

Note: Later, you will see that these variables can be set up at the beginning of your applications. All the I/O functions will use these variables.

You call this function with the names of the index file and the data file that you want opened. The function then attempts to open the index file first in line 39. The file is opened using the `fopen()` function. By passing it the value of `"r+b"`, you are attempting to open an existing file for reading and writing in binary mode. If the file does not exist, an error will be returned. Line 42 handles this by then attempting to create the file. If the file can't be created, then `rv` is set to show an error return value of `CREATE_ERROR`. Earlier, you defined `CREATE_ERROR` in your TYAC.H header file. If the file didn't exist and was successfully created, then a flag, `cr_flg`, is set to `TRUE`. This will be used later to indicate that the header information needs to be set up.

Line 49 checks to see if there was an error with the index file. If there wasn't, then the same processes are repeated for opening the data file in lines 52 to 57.

If there hasn't been an error up to this point, then the header information can be set up. Line 61 begins the process of setting up the header file information. Line 64 checks the `cr_flg`, to see if a new file was created. If a new file wasn't created, then line 66 calls the `get_rec()` function to read an integer from the index file. This integer is placed in the global variable `nbr_records`. If the read was successful, then a second read of the index is done to retrieve the address of the starting record, `start_rec`.

Note: The `get_rec()` function is a new function that will be covered in the next section.

If the index file was created, then lines 77 and 78 set the two global variables, `nbr_records` and `start_rec`. The number of records is set to zero because there are no records. The `start_rec` is set to 1 because there are no records in the file. With this, the header information is set up and the files are ready to use.

Getting a Data Record: *get_rec()*

The `get_rec()` function enables you to get a record. Listing 17.2 presents this function.

Listing 17.2. GETREC.C. Getting a record—the `get_rec()` function.

```
1:   /* ----------------------------------------------------------
2:    * Program: GETREC.C
3:    * Authors: Bradley L. Jones
4:    *          Gregory L. Guntle
5:    *
6:    * Purpose: Reads a record from a file.
7:    *
8:    * Enter with: rec    = record # to retrieve
9:    *             FILE * = FILE ptr to file
10:   *             rec_size= size of record
11:   *             offset = any offset to adjust to (hdr?)
12:   *             buff = place to put information being read
13:   *
14:   * Returns:  0 = No Error
15:   *          >0 = Error - see DEFINES in TYAC.H for FILE I/O
```

continues

Listing 17.2. continued

```
16:   *============================================================*/
17:
18:  #include <stdio.h>
19:  #include "tyac.h"
20:
21:  int get_rec(int rec, FILE *fp, int rec_size, int offset, char
                 *buff)
22:  {
23:     int rv = NO_ERROR;
24:
25:     if ( rec == 0)  /* Getting Index Header ? */
26:       rec++;        /* Adjust to fit into formula */
27:
28:     /* Seek to position */
29:     if ( fseek(fp, (rec-1)*rec_size+offset,
30:          SEEK_SET) == 0 )
31:     {
32:       /* Read information */
33:       if ( fread(buff, rec_size, 1, fp) != 1 )
34:         rv = READ_ERROR;
35:     }
36:     else
37:       rv = SEEK_ERROR;
38:
39:     return(rv);
40:  }
```

Analysis The `get_rec()` function requires several parameters. The first, `rec`, is the number of the record to retrieve. Each record is numbered sequentially from the beginning to the end of the file. The first record is one, the second is two, and so on. The second parameter, `fp`, is the file pointer. This will be the opened file that the record is retrieved from. The third parameter, `rec_size`, is the size of the information or record to be read. The `offset`, or forth parameter, is the size of the header record in the file if there is one. This offset is needed to adjust for the room taken by the header record. The last parameter, `buff`, is the area to store the information being read.

The `get_rec()` function calculates where the record is in the file. This is done by using the record number that tells how many records into the file the record is. The record number is multiplied by the size of each record. Actually, one less than the record number is used so that the value gives the distance into the file that the record is. A final adjustment, by adding the size of the header information, provides the exact location.

This calculation can be seen in line 29 of the `get_rec()` function. Reviewing the function, you can see how this works. Line 25 checks to see if the record number being retrieved is zero. If a header record or other miscellaneous data is being retrieved, then the record number won't be applicable, so zero will be passed. Because the formula for calculating the position subtracts one, the record number has one added. This allows the final result to be zero.

Line 29 uses the formula for calculating where the record will be located. Using the `fseek()` command, the position in the file is set to the location that was calculated. If the placement of the position is successful, then line 33 reads the information into the buffer, `buff`. In the cases of either the seeking or the reading failing, defined error constants are placed in the return value variable, `rv`. With that, the function ends.

Writing a Data Record: *put_rec()*

While you can retrieve records with a `get_rec()` function, you can put records back with a `put_rec()` function. The `put_rec()` function enables you to write a record at a specific location in the file. This function will look almost identical to the `get_rec()` function with the exception of the error constants that are used and the use of `fwrite()` instead of `fread()`. Listing 17.3 presents the `put_rec()` function.

Type

Listing 17.3. PUTREC.C. Writing a record—the `put_rec()` function.

```
1:  /* ----------------------------------------------------------
2:   * Program: PUTREC.C
3:   * Authors: Bradley L. Jones
4:   *          Gregory L. Guntle
5:   *
6:   * Purpose: Writes a record to a file.
7:   *
8:   * Enter with: rec    = record # to write
9:   *             FILE * = FILE ptr to file
10:  *             rec_size= size of record
11:  *             offset = any offset to adjust to (hdr?)
12:  *             buff = place which holds the data to write
13:  *
14:  * Returns:  0 = No Error
15:  *          >0 = Error - see DEFINES in TYAC.H for FILE I/O
16:  *==========================================================*/
17:
18:  #include <stdio.h>
19:  #include "tyac.h"
20:
```

continues

Listing 17.3. continued

```
21:  int put_rec(int rec, FILE *fp, int rec_size, int offset, char
                 *buff)
22:  {
23:     int rv = NO_ERROR;
24:
25:     if ( rec == 0)  /* Writing Index Header ? */
26:       rec++;        /* Adjust to fit into formula */
27:
28:     /* Seek to position */
29:     if ( fseek(fp, (rec-1)*rec_size+offset,
30:          SEEK_SET) == 0 )
31:     {
32:       /* Write the information */
33:       if ( fwrite(buff, rec_size, 1, fp) == 1 )
34:          fflush(fp);
35:       else
36:          rv = WRITE_ERROR;
37:     }
38:     else
39:       rv = SEEK_ERROR;
40:
41:     return(rv);
42:  }
```

Analysis As stated already, this listing is virtually identical to the `get_rec()` function described earlier. Line 29 seeks the position that the record is to be written to. If successful, line 33 writes the information that is in the buffer, `buff`, to the location. If information was already at that particular location, then it will be overwritten. If not, then new information will be added to the file.

Updating the Header Record: *update_header()*

When adding new records to the database, you'll need to update the header information as well. As stated earlier, the index files will have header information. This header information will be the number of records in the file and the record number of the first sorted record. Listing 17.4 includes a function that updates this header information.

Type **Listing 17.4. UPDHDR.C. The `update_header()` function.**

```
1:   /* ----------------------------------------------------------
2:    * Program: UPDHDR.C
```

```
3:    * Authors: Bradley L. Jones
4:    *          Gregory L. Guntle
5:    *
6:    * Purpose: Updates the header info of the Index.
7:    *
8:    * Enter with: All variables are global
9:    *
10:   * Returns:  0 = No Error
11:   *          >0 = Error - see DEFINES in TYAC.H for FILE I/O
12:   *============================================================*/
13:
14:  #include <stdio.h>
15:  #include "tyac.h"
16:
17:  /*  Global Variables */
18:  extern FILE *idx_fp;
19:  extern FILE *db_fp;
20:  extern int nbr_records;
21:  extern int start_rec;
22:
23:
24:  int update_header()
25:  {
26:    int rv = NO_ERROR;
27:
28:    /* Update number of records */
29:    rv = put_rec(0, idx_fp, sizeof(int), 0,
30:                 &nbr_records);
31:    if (rv == 0)
32:    {
33:       rv = put_rec(0, idx_fp, sizeof(int), sizeof(int),
34:                    &start_rec);
35:    }
36:
37:    return(rv);
38:  }
```

Analysis You should notice that this function doesn't take any parameters. This is because the function expects global variables to be available. The global variables are the same as those presented in the `open_files()` listing (Listing 17.1). They are the index file pointer (`idx_fp`), the data file pointer (`db_fp`), the number of records (`nbr_records`), and the starting record number (`start_rec`). External declarations in lines 18 to 21 help to inform that this function requires these global variables.

Line 29 uses the `put_rec()` function to write the number of records. From the earlier discussion, you should be able to understand the parameters being passed. The first parameter for record number is set to zero, thus stating that there isn't a record

number. The second parameter, `idx_fp`, shows that the index file is being updated. The third parameter shows that the size of the data being written is an integer. The fourth parameter gives the offset to be added. This is zero because the number of records is first in the file. The last parameter is the address of the `nbr_records`. This is typecast to a character pointer to avoid a warning when you compile.

If the number of records is written successfully, then the starting record is updated in line 32. This uses a second call to `put_rec()` with only a slight difference. Instead of the fourth parameter being zero, it is `sizeof(int)`. This allows the `start_rec` value to be written at an offset of one integer into the file.

Closing the Files: *close_ files()*

Just as you opened a file before using it, when you are done with a file, you need to close it. In Listing 17.5, the `close_files()` function performs the closing functions for the global file pointers that were opened.

Listing 17.5. CLOSFILE.C. Closing a file—the `close_files()` function.

```
1:  /* ----------------------------------------------------------
2:   * Program: CLOSFILE.C
3:   * Authors: Bradley L. Jones
4:   *          Gregory L. Guntle
5:   *
6:   * Purpose: Closes both the index and DBF file.
7:   *          Also, updates the hdr information.
8:   *
9:   * Enter with:
10:  *
11:  * Returns:  0 = No Error
12:  *          >0 = Error - see DEFINES in TYAC.H for FILE I/O
13:  *==========================================================*/
14:
15: #include <stdio.h>
16: #include "tyac.h"
17:
18: /*  Global Variables */
19: extern FILE *idx_fp;
20: extern FILE *db_fp;
21: extern int nbr_records;
22: extern int start_rec;
23:
24:
```

```
25:  int close_files()
26:  {
27:     int rv = NO_ERROR;    /* Assume no errors will happen */
28:
29:     /* Close data file first */
30:     if ( fclose(db_fp) == 0 )
31:     {
32:       /* Update hdr record in INDEX file */
33:       rv = update_header();
34:
35:       if (rv == 0)
36:       {
37:
38:         fflush(idx_fp);
39:       }
40:
41:       if ( fclose(idx_fp) != 0 )
42:          rv = CLOSE_ERROR;
43:     }
44:     else
45:        rv = WRITE_ERROR;
46:
47:     return(rv);
48:  }
```


Analysis This function again includes the external declarations in lines 19 to 22. In line 30, the function uses the `fclose()` function to close the data file. If successful, the index will be closed. Before closing the index file, the header information should be updated. This is done in line 33 using the `update_header()` function that was described earlier. If the update of the header was successful, then line 38 calls `fflush()` to ensure that the information is actually written to the file and not held in a buffer. Line 41 then closes the index file. Any errors are returned in line 50 to the calling function.

Note: Several C functions were used without explanation. These were `fopen()`, `fclose()`, `fread()`, `fwrite()`, and `fflush()`. These are standard functions. If you need more information on these functions, they are in most beginning C books. In addition, they should be covered in your compiler's function reference materials.

Working with the Medium Screen

With the functions presented, you are ready to begin modifying the *Record of Records!* application so that it will be able to work with files. You will start with the medium screen. Before beginning, however, you should add the previous five functions to your TYAC.LIB library. You should have already added the prototypes to the header earlier today.

Before adding the processes of manipulating medium records, you will need to make a few modifications to the RECOFREC.C source file. The global variables needed by many of the functions have to be declared. You should add the following global declarations to the beginning of the RECOFREC.C source file.

```
FILE *idx_fp;            /* Index File Ptr */
FILE *db_fp;             /* Data file */
int nbr_records;         /* Total # of recs */
int start_rec;           /* Starting rec for alpha */
int disp_rec;
```

The first four of these variables will be used in the ways described before. The `disp_rec` variable will be used to hold the record number of any record that is currently being displayed.

The RECOFREC.H header file should receive a few modifications also. To this header file, you should add the prototypes of a few new functions that will be created later today. By adding them now, you will be ready to dive into the medium code's file I/O functions. The following function prototypes should be in the RECOFREC.H header file medium code prototypes.

```
/*-----------------------------*
 * Prototypes for medium screen  *
 *-----------------------------*/

int  do_medium_actionbar(void);
void display_medium_help(void);

void display_context_help( char *, int );
void display_cntxt_help_msg( char *, char *, char * );

void clear_medium_fields(void);

int  verify_mdata(void);
int  add_mdata(void);
int  add_medium_rec(void);
int  del_med_rec(void);
int  proc_med_rec(int);
```

Note: Earlier today you were asked to modify the RECORDS.H header file. Make sure that you added the medium code's index file structure:

```
typedef struct
{
   char code[2+1];
   unsigned short prev;
   unsigned short next;
   unsigned short data;
} MEDIUM_INDEX;
```

In addition to the RECORDS.H header file, the RECOFREC.C file, and the RECOFREC.H header file, the MEDIUM.C file will also need several changes. Following is an updated MEDIUM.C file.

Type

Listing 17.6. MEDIUM.C. The medium screen's main file.

```
1:  /*=========================================================
2:   * Filename: medium.c
3:   *
4:   * Author:   Bradley L. Jones
5:   *           Gregory L. Guntle
6:   *
7:   * Purpose:  Allow entry and edit of medium codes.
8:   *
9:   * Note:     This listing is linked with RECofREC.c
10:  *           (There isn't a main() in this listing!)
11:  *=========================================================*/
12:
13: #include <stdlib.h>
14: #include <stdio.h>
15: #include <string.h>
16: #include <conio.h>        /* for getch() */
17: #include "tyac.h"
18: #include "records.h"
19:
20: /*--------------------*
21:  *     prototypes     *
22:  *--------------------*/
23:
24: #include "recofrec.h"
25:
26: void draw_medium_screen( void );
27: void draw_medium_prompts( void );
```

continues

Listing 17.6. continued

```
28:  void display_medium_fields( void );
29:
30:  int  clear_medium_fields(void);
31:  int  get_medium_data( int row );
32:  int  get_medium_input_data( void );
33:  void display_medium_help(void);
34:
35:  int  add_mdata(void);
36:
37:  /* Global variables */
38:  extern FILE *idx_fp;      /* Main File ptr to data file */
39:  extern FILE *db_fp;       /* Data file */
40:  extern nbr_records;       /* Total # of rec for mediums */
41:  extern int start_rec;
42:  extern int disp_rec;
43:
44:
45:  /*-------------------*
46:   * Defined constants *
47:   *-------------------*/
48:
49:  #define MEDIUM_IDX   "MEDIUMS.IDX"
50:  #define MEDIUM_DBF   "MEDIUMS.DBF"
51:  #define HELP_DBF     "MEDIUM.HLP"
52:
53:  /*------------------------*
54:   * structure declarations *
55:   *------------------------*/
56:
57:  MEDIUM_REC medium;
58:  MEDIUM_REC *p_medium = &medium;
59:  MEDIUM_REC med_prev;
60:
61:  /*=======================================================*
62:   *   do_medium_screen()                                  *
63:   *=======================================================*/
64:
65:  int do_medium_screen(void)
66:  {
67:    int rv;
68:
69:    /* Open both Index and DBF file */
70:    if ( (rv = open_files(MEDIUM_IDX, MEDIUM_DBF)) == 0 )
71:    {
72:      /* Setup for adding new records at beginning */
73:      memset(&med_prev, '\0', sizeof(med_prev));
74:      disp_rec = 0;            /* Initialize displaying rec # */
75:      clear_medium_fields();
76:      draw_medium_screen();
```

```
77:      get_medium_input_data();
78:      rv = close_files();       /* Close IDX and DBF file */
79:    }
80:    else
81:    {
82:      display_msg_box("Error opening MEDIUM files...",
83:                      ct.err_fcol, ct.err_bcol );
84:    }
85:    return(rv);
86:  }
87:
88:  /*------------------------*
89:   *   draw_medium_screen()  *
90:   *------------------------*/
91:
92:  void draw_medium_screen( void )
93:  {
94:
95:
96:    draw_borders("     MEDIUM     ");  /* draw screen bckgrnd */
97:
98:    write_string( " File   Edit   Search   Help",
99:          ct.abar_fcol, ct.abar_bcol, 1, 2);
100:   write_string(
101:     "<F1=Help> <F3=Exit> <F4=Save> <F7=Next> <F8=Prev>"
        "<F10=Actions>",
102:         ct.abar_fcol, ct.abar_bcol, 24, 3);
103:
104:   draw_medium_prompts();
105:   display_medium_fields();
106: }
107:
108: /*------------------------*
109:  *   draw_medium_prompts() *
110:  *------------------------*/
111:
112: void draw_medium_prompts( void )
113: {
114:   write_string("Medium Code:",
115:         ct.fld_prmpt_fcol, ct.fld_prmpt_bcol, 4, 3 );
116:   write_string("Medium Description:",
117:         ct.fld_prmpt_fcol, ct.fld_prmpt_bcol, 6, 3 );
118: }
119:
120: /*------------------------*
121:  *   draw_medium_fields()  *
122:  *------------------------*/
123:
124: void display_medium_fields( void )
125: {
```

continues

Listing 17.6. continued

```
126:    write_string("__", ct.fld_fcol, ct.fld_bcol, 4, 17 );
127:    write_string("_____________________________________________",
128:          ct.fld_fcol, ct.fld_bcol, 6, 24 );
129:
130:    /* display data, if exists */
131:
132:    write_string(medium.code, ct.fld_fcol, ct.fld_bcol, 4, 17 );
133:    write_string(medium.desc, ct.fld_fcol, ct.fld_bcol, 6, 24 );
134: }
135:
136: /*-----------------------------------------------------*
137:  *   get_medium_input_data()                            *
138:  *-----------------------------------------------------*/
139: int get_medium_input_data( void )
140: {
141:     int   position,
142:           rv,
143:           loop = TRUE;
144:
145:     /*  Set up exit keys.  */
146:      static char fexit_keys[ 19 ] = { F1, F2, F3, F4, F5, F6,
147:                      F7, F8, F10,
148:                      ESC_KEY, PAGE_DN, PAGE_UP, CR_KEY,
149:                      TAB_KEY, ENTER_KEY, SHIFT_TAB,
150:                      DN_ARROW, UP_ARROW, NULL };
151:
152:     static char *exit_keys = fexit_keys;
153:     getline( SET_EXIT_KEYS, 0, 18, 0, 0, 0, exit_keys );
154:
155:     /*** setup colors and default keys ***/
156:     getline( SET_DEFAULTS, 0, 0, 0, 0, 0, 0 );
157:     getline( SET_NORMAL, 0, ct.fld_fcol, ct.fld_bcol,
158:                 ct.fld_high_fcol, ct.fld_high_bcol, 0 );
159:     getline( SET_UNDERLINE, 0, ct.fld_fcol, ct.fld_bcol,
160:                 ct.fld_high_fcol, ct.fld_high_bcol, 0 );
161:     getline( SET_INS, 0, ct.abar_fcol, ct.abar_bcol, 24, 76, 0 );
162:
163:
164:     position = 0;
165:
166:     while( loop == TRUE )      /**  get data for top fields  **/
167:     {
168:        switch( (rv = get_medium_data( position )) )
169:        {
170:           case CR_KEY    :
171:           case TAB_KEY   :
172:           case ENTER_KEY :
173:           case DN_ARROW  : /* go down a field */
174:                  ( position == 1 ) ? ( position = 0 ) : position++;
```

```
175:                            break;
176:
177:            case SHIFT_TAB :
178:            case UP_ARROW  : /* go up a field */
179:                   ( position == 0 ) ? ( position = 1 ) : position—;
180:                            break;
181:
182:            case ESC_KEY :
183:            case F3      :   /* exit back to main menu */
184:                            if( (yes_no_box( “Do you want to exit?”,
185:                                    ct.db_fcol, ct.db_bcol )) == ‘Y’
)
186:                            {
187:                                loop = FALSE;
188:                            }
189:                            break;
190:
191:            case F4:  /* add data */
192:                      rv = add_mdata();
193:                      if ( rv == NO_ERROR )
194:                      {
195:                        /* Reset display counter */
196:                        display_msg_box(“Added record!”,
197:                                  ct.db_fcol, ct.db_bcol);
198:                        disp_rec = 0;
199:                        clear_medium_fields();
200:                        draw_medium_screen();
201:                        position = 0;
202:                      }
203:                      else  /* Only do next part if File I/O  */
204:                      if ( rv > NO_ERROR )
205:                      {
206:                        display_msg_box(“Fatal Error writing data...”,
207:                             ct.err_fcol, ct.err_bcol );
208:                        exit(1);
209:                      }
210:                      break;
211:
212:            case F5:  /* Change Record */
213:
214:                      rv = add_mdata();   /* updates record */
215:
216:                      if ( rv == NO_ERROR )
217:                      {
218:                        /* Reset display counter */
219:
220:                        display_msg_box(“Record Updated!”,
221:                                  ct.db_fcol, ct.db_bcol);
222:                      }
223:                      else
224:                      if ( rv > NO_ERROR )
```

continues

Listing 17.6. continued

```
225:                    {
226:                      display_msg_box("Fatal Error writing
                                           data...",
227:                           ct.err_fcol, ct.err_bcol );
228:
229:                      exit(1);
230:                    }
231:                    break;
232:
233:          case F6:  /* Delete data */
234:                    /* Make sure rec is on screen */
235:                    if( (yes_no_box( "Delete record ?",
236:                          ct.db_fcol, ct.db_bcol )) == 'Y' )
237:                    {
238:                      rv = del_med_rec();
239:                      if (rv == NO_ERROR)
240:                      {
241:                        disp_rec = 0;
242:                        clear_medium_fields();
243:                        draw_medium_screen();
244:                      }
245:                      else
246:                      {
247:                        display_msg_box("Fatal Error deleting
                                             data...",
248:                           ct.err_fcol, ct.err_bcol );
249:                        exit(1);
250:                      }
251:                    }
252:                    break;
253:
254:          case F7:  /* Next record */
255:                      rv = proc_med_rec(NEXT_REC);
256:                      if ( rv == NO_ERROR )
257:                      {
258:                         draw_medium_screen();
259:                      }
260:                      else
261:                      {
262:                        display_msg_box("Fatal Error processing
                                             data...",
263:                             ct.err_fcol, ct.err_bcol );
264:                        exit(1);
265:                      }
266:                      break;
267:
268:          case F8:  /* Prev record */
269:                      rv = proc_med_rec(PREVIOUS_REC);
```

```
270:                        if ( rv == NO_ERROR )
271:                        {
272:                           draw_medium_screen();
273:                        }
274:                        else
275:                        {
276:                          display_msg_box("Fatal Error
                                                processing data...",
277:                                ct.err_fcol, ct.err_bcol );
278:                          exit(1);
279:                        }
280:                        break;
281:
282:           case F10:         /* action bar */
283:                             rv = do_medium_actionbar();
284:
285:                             if( rv == F3 )
286:                             {
287:                                if( (yes_no_box( "Do you want to exit?",
288:                                      ct.db_fcol, ct.db_bcol )) == 'Y' )
289:                                {
290:                                    loop = FALSE;
291:                                }
292:                             }
293:
294:                             position = 0;
295:                             break;
296:
297:           case PAGE_DN :    /* go to last data entry field */
298:                             position = 1;
299:                             break;
300:
301:           case PAGE_UP :    /* go to first data entry field */
302:                             position = 0;
303:                             break;
304:
305:           case F1:          /* context sensitive help */
306:                             display_context_help( HELP_DBF, position );
307:                             break;
308:
309:           case F2:          /* extended help */
310:                             display_medium_help();
311:                             break;
312:
313:           default:          /* error */
314:                             display_msg_box( " Error ",
315:                                    ct.err_fcol, ct.err_bcol );
316:                             break;
317:
318:        }     /* end of switch */
```

continues

Listing 17.6. continued

```
319:     }        /* end of while  */
320:
321:     return( rv );
322: }
323:
324: /*-----------------------------------------------------*
325:  *   get_medium_data()                                 *
326:  *-----------------------------------------------------*/
327:
328: int get_medium_data( int row )
329: {
330:     int rv;
331:
332:     switch( row )
333:     {
334:        case 0 :
335:           rv = getline( GET_ALPHA, 0,  4, 17, 0,  2, medium.code);
336:           break;
337:        case 1 :
338:           rv = getline( GET_ALPHA, 0,  6, 24, 0, 35, medium.desc);
339:           break;
340:     }
341:     return( rv );
342: }
343:
344: /*-----------------------------------------------------*
345:  *   clear_medium_fields()                            *
346:  *-----------------------------------------------------*/
347:
348: int clear_medium_fields(void)
349: {
350:   getline( CLEAR_FIELD, 0,  3, 0, 0, 0, medium.code );
351:   getline( CLEAR_FIELD, 0, 36, 0, 0, 0, medium.desc );
352:
353:   return(0);
354: }
355:
356: /*-----------------------------------------------------*
357:  *   add_mdata()                                      *
358:  *                                                    *
359:  *   Returns  0 - No Errors                           *
360:  *           >0 - File I/O Error                      *
361:  *           <0 - Missing info before can store       *
362:  *-----------------------------------------------------*/
363:
364: int add_mdata()
365: {
366:    int rv = NO_ERROR;
367:
```

```
368:    /* Verify data fields */
369:    rv = verify_mdata();
370:    if ( rv == NO_ERROR )
371:    {
372:      /* Check to see if matches old rec */
373:      /* If match - then update db record only */
374:      if ( stricmp(med_prev.code, medium.code) == 0)
375:         rv = put_rec(disp_rec, db_fp,
376:              sizeof(medium), 0, (char *)&medium);
377:      else
378:        /* Keys no longer match - need to
379:           add this key as a new one */
380:        rv = add_medium_rec();
381:    }
382:
383:    return(rv);
384: }
385:
386: /*-------------------------------------------------------*
387:  *   Verify data fields                                  *
388:  *-------------------------------------------------------*/
389: int verify_mdata()
390: {
391:    int rv = NO_ERROR;
392:
393:    if( strlen( medium.code ) == 0 )
394:    {
395:       display_msg_box("Must enter a medium code",
396:              ct.err_fcol, ct.err_bcol);
397:       rv = -1;
398:    }
399:    else
400:    if( strlen( medium.desc ) == 0 )
401:    {
402:       display_msg_box("Must enter a description",
403:              ct.err_fcol, ct.err_bcol);
404:       rv = -1;
405:    }
406:
407:    return(rv);
408: }
409:
410: /*-------------------------------------------------------*
411:  *   display_medium_help()                               *
412:  *-------------------------------------------------------*/
413:
414: void display_medium_help(void)
415: {
416:   int  ctr;
417:   char *scrnbuffer;
```

continues

Listing 17.6. continued

```
418:
419:   char helptext[19][45] = {
420:      "              Medium Codes",
421:      "-------------------------------------------",
422:      "",
423:      "The medium code screen allows you to track",
424:      "the different types of storage mediums that",
425:      "your music collection may be stored on. A",
426:      "two character code will be used in the",
427:      "Musical Items Screen to verify that the",
428:      "medium type is valid. The code entered will",
429:      "need to match a code entered on this screen.",
430:      "Additionally, the codes will be used in",
431:      "reporting on your musical items.",
432:      "",
433:      "An example of a musical code might be:",
434:      "",
435:      "       CD - Compact Disk",
436:      "       CS - Cassette",
437:      "       VD - Video",
438:      "-------------------------------------------" };
439:
440:   scrnbuffer = save_screen_area(2, 23, 28, 78 );
441:   cursor_off();
442:
443:   grid( 3, 23, 28, 77, ct.shdw_fcol, ct.bg_bcol, 3 );
444:   box(2, 22, 29, 78, SINGLE_BOX, FILL_BOX,
445:                       ct.help_fcol, ct.help_bcol);
446:
447:   for( ctr = 0; ctr < 19; ctr++ )
448:   {
449:     write_string( helptext[ctr],
450:                   ct.help_fcol, ct.help_bcol, 3+ctr, 32 );
451:   }
452:
453:   getch();
454:   cursor_on();
455:   restore_screen_area(scrnbuffer);
456: }
457:
458: /*-------------------------------------------------- *
459:  * Function: display_context_help()                 *
460:  * Purpose:  Display help that is relative to a specific *
461:  *           field on a specific screen.            *
462:  * Notes:    The first parameter needs to be one of the  *
463:  *           following:                             *
464:  *             medium - medium screen               *
465:  *             album  - musical items screen        *
466:  *             group  - groups screen               *
```

```
467:  *           It is assumed that each screen has its own  *
468:  *           file                                        *
469:  *-------------------------------------------------------- */
470: void display_context_help( char *file, int position )
471: {
472:    FILE *fp;
473:    int   ctr;
474:    char *rv = NULL;
475:
476:    char *buffer1 = NULL;
477:    char *buffer2 = NULL;
478:    char *buffer3 = NULL;
479:
480:    /* allocate buffers */
481:    buffer1 = (char *) malloc(65 * sizeof(char));
482:    buffer2 = (char *) malloc(65 * sizeof(char));
483:    buffer3 = (char *) malloc(65 * sizeof(char));
484:
485:    /* make sure all the allocations worked */
486:    if( buffer1 == NULL || buffer2 == NULL || buffer3 == NULL)
487:    {
488:        display_msg_box("Error allocating memory...",
489:                        ct.err_fcol, ct.err_bcol );
490:        if( buffer1 != NULL )
491:           free(buffer1);
492:        if( buffer2 != NULL )
493:           free(buffer2);
494:        if( buffer3 != NULL )
495:           free(buffer3);
496:
497:        return;
498:    }
499:
500:
501:    fp = fopen( file, "r");
502:
503:    if( fp == NULL )  /* make sure the file was opened */
504:    {
505:       display_msg_box("Error opening help file...",
506:                       ct.err_fcol, ct.err_bcol );
507:    }
508:    else
509:    {
510:       /* spin through to appropriate record */
511:       for( ctr = 0; (ctr < (position * 3)) ; ctr++ )
512:       {
513:          rv = fgets( buffer1, 65, fp );
514:          if( rv == NULL)
515:            break;
516:       }
```

continues

Listing 17.6. continued

```
517:
518:       /* ready to read three lines */
519:       if( rv != NULL || position == 0 )
520:       {
521:          rv = fgets( buffer1, 65, fp );
522:          if( rv != NULL )
523:          {
524:             rv = fgets( buffer2, 65, fp );
525:             if( rv != NULL )
526:                 rv = fgets( buffer3, 65, fp );
527:          }
528:
529:          display_cntxt_help_msg( buffer1, buffer2, buffer3 );
530:       }
531:       else  /* hit end of file too soon */
532:       {
533:          display_msg_box( "Error in message file...",
534:                        ct.err_fcol, ct.err_bcol );
535:       }
536:
537:       fclose( fp );
538:    }
539:
540:    free( buffer1 );
541:    free( buffer2 );
542:    free( buffer3 );
543: }
544:
545: /*-----------------------------------------------------*
546:  *   display_context_help()                            *
547:  *-----------------------------------------------------*/
548: void display_cntxt_help_msg( char *string1,
549:                              char *string2,
550:                              char *string3 )
551: {
552:    char *scrn_buffer = NULL;
553:    scrn_buffer = save_screen_area( 10, 16, 10, 70 );
554:    cursor_off();
555:
556:    grid( 11, 16, 10, 69, ct.shdw_fcol, ct.bg_bcol, 3 );
557:    box(10, 15, 11, 70, SINGLE_BOX, FILL_BOX,
558:                    ct.help_fcol, ct.help_bcol);
559:
560:    write_string( string1, ct.help_fcol, ct.help_bcol, 11, 14 );
561:    write_string( string2, ct.help_fcol, ct.help_bcol, 12, 14 );
562:    write_string( string3, ct.help_fcol, ct.help_bcol, 13, 14 );
563:
564:    write_string( "Press any key to continue...",
565:                  ct.help_fcol, ct.help_bcol, 14, 14 );
```

```
566:
567:     getch();
568:     cursor_on();
569:     restore_screen_area(scrn_buffer);
570: }
571:
572:
573:
574: /*=========================================================*
575:  *                      end of listing                     *
576:  *=========================================================*/
```


Note: The *Record of Records!* application should have several source files, not including the TYAC.LIB library. The files that you should be compiling include the following:

RECOFREC.C ABOUT.C
ALBMABAR.C GRPSABAR.C MEDMABAR.C MMNUABAR.C
MEDIUM.C ALBUMS.C GROUPS.C

To these files, you should link the TYAC.LIB library, which should contain all the functions learned in this book, including those from earlier today.

Warning: If you compile the medium screen with these changes, you will get unresolved externals for `del_med_rec()`, `proc_med_rec()`, and `add_medium_rec()`. These new functions will be covered later today.

Analysis While the MEDIUM.C listing is getting long, you should already have much of it from previous days. This should be a nearly complete MEDIUM.C source file. You'll still need a few additional files to complete the file I/O functions and eliminate the unresolved externals that you received from the new MEDIUM.C listing.

The changes in this listing start in line 35. The prototype for `add_mdata()` has been placed here. Following this, several external declarations are present in lines 38 to 42. These are the same external declarations that you have seen before. In lines 49 to 51, you should also notice a change. There is now more than one MEDIUM file. The defined constant `MEDIUM_DBF` contains the name of the data file. The `MEDIUM_IDX` contains the name of the index file. The help file has remained the same.

Expert Tip: It's best to give an index and a data file the same name with the exception of the three character extension. For example, both of the Medium Code's files are called MEDIUMS.

Line 59 has also been added. A second Medium Code structure has been declared. This will be used to hold information about a Medium Code record before changes are made. You will see it used later.

The changes up to this point have been outside of the actual medium code functions. Line 72 starts the internal changes. When you first enter the medium code, you will open the files. Line 70 does this using the `open_files()` function described earlier. The index and data files for the Medium Codes screen are opened. If the open was not successful, then an error message is displayed in line 82. If the open was successful, then the `med_prev` Medium Code structure is cleared out with `memset()`. The currently displayed record, `disp_rec` is set to zero because no records are being displayed. The functions that were present in this function before are then called. When the user is done entering data, then line 78 calls the `close_files()` function from earlier.

In displaying the screen, only a minor change is made. In line 101, the keys' values that are displayed on the bottom of the screen are updated. The F4, F7, and F8 keys are added to the display. You can choose to display whichever keys you believe are the most important. In the `get_medium_input_data()` function, a similar change is made in lines 146 to 153. The setup for the `getline()` function needs to be changed to include the newly-added key functions. The F4, F5, F6, F7, and F8 keys need to be added, bringing the total number of exit keys to 18.

In the following `switch` statement, cases will need to be added or modified for each of these new function keys. The F4 key in lines 191 to 210 is used to add new data to the data file. This function calls the `add_mdata()` function. If the `add_mdata()` function doesn't return an error, then line 196 displays a message box stating that the add was successful. The screen is then cleared for a new record to be added. Line 198

then resets the current display record field, `disp_rec`. The fields and screen are then cleared, followed by the position being reset to zero. If the `add_mdata()` returned a fatal error—a number greater than `NO_ERROR`(0)—then a message is displayed and the program instantly exits.

Tip: You should avoid exiting in the middle of a program, such as shown in line 208. The only time that you should do this is when a fatal error occurs. You should allow the program to continue running whenever possible.

The `add_mdata()` function is in lines 356 to 384. This function does two things. First, it calls a function, `verify_mdata()`, that checks to see that all of the data entered is valid (line 369). It then checks to see if the user is adding a new record, or trying to add a record that was changed. This is done starting in line 374. Using `stricmp()`, the code from the current entry and edit screen, `medium.code`, is compared to the medium code in `med_prev.code`. If the two values match, then line 375 calls `put_rec()` to update the current record, `disp_rec`. In the case of a new record, the `med_prev.code` will be empty so they won't match. The codes also won't match if the user changed the one on the screen. In these cases, line 380 calls the `add_medium_rec()` function to add a new record. The `add_medium_rec()` function will be presented later today. As you should be able to see, the `add_mdata()` function effectively covers both adding and updating records.

Warning: The `add_mdata()` function makes the assumption that the code field is required. This assumes that the `verify_mdata()` will give an error if the `medium.code` field is blank. If this key field is allowed to be blank, then this logic will need to be modified; however, you shouldn't allow the key field to be blank.

Before returning to the new cases in the `get_medium_data()` function, you should first look at the `verify_mdata()` function in lines 386 to 408. This function contains all the edits that need to occur before the record can be added to the data file. By consolidating them in a function such as this, it makes them easy to call from several functions. You'll see that the add functions aren't the only function that verifies the data.

The F5 case in lines 212 to 231 is used to update a record that is currently displayed on the screen. This function is nearly identical to the add function. The difference is that the change function doesn't refresh the screen. The record that was updated remains displayed.

The F6 case in lines 233 to 252 is used to delete or remove a record from the data and index files. Because deleting a record involves permanently removing it from the file, this case starts by prompting the user to ensure that the user really wants to delete. This is done in line 235 using the `yes_no_box()` function. If the user does want to delete, then the `del_med_rec()` function is called to actually do the delete. This function is covered later today. If the `del_med_rec()` function is successful, then the screen is refreshed for a new record (lines 241 to 243); otherwise, a fatal error message is displayed and the program exits.

The F7 function in lines 254 to 266 and the F8 function in lines 268 to 280 are similar. These functions display the next or previous record. They start by calling the `proc_med_rec()` function, which retrieves a record. One of the two defined constants—`NEXT_REC` or `PREVIOUS_REC`—are passed to signal if the next or previous record is retrieved. The `proc_med_rec()` function will be covered later today. If the `proc_med_rec()` function is successful, then the medium screen is redrawn to display the new information. If the function wasn't successful, then a fatal error is displayed and the program exits.

With this, you have seen all the changes made to MEDIUM.C. Three new functions were called that need to be created, `add_medium_rec()`, `del_med_rec()`, and `proc_med_rec()`. These are each covered next.

Adding a Record: The *add_medium_rec()* function

The `add_medium_rec()` function is used to add a new record to the medium code data file. This function is presented in Listing 17.7.

Listing 17.7. ADDMREC.C. Adding a medium code.

```
1:   /*=======================================================
2:    * Filename: ADDMREC.C
3:    *
4:    * Author:   Bradley L. Jones
5:    *           Gregory L. Guntle
6:    *
7:    * Purpose: Adds a record to a MEDIUM DB
8:    *
```

```
9:    *=========================================================*/
10:
11:  #include <stdio.h>
12:  #include <string.h>
13:
14:  #include "records.h"
15:  #include "tyac.h"
16:
17:  /* Global variables */
18:  extern MEDIUM_REC medium;      /* Record structure with data */
19:  extern MEDIUM_REC med_prev;
20:  extern FILE *idx_fp;           /* Main File ptr to data file */
21:  extern FILE *db_fp;            /* Data file */
22:  extern int nbr_records;        /* Total # of recs */
23:  extern int start_rec;          /* Starting rec for alpha */
24:
25:  /*---------------------------------------------------*
26:   *  add_medium_rec();                                *
27:   *  Returns:  0 - No Error                           *
28:   *            >0 Error - see defines at top of file  *
29:   *---------------------------------------------------*/
30:
31:  int add_medium_rec()
32:  {
33:   int result;
34:   MEDIUM_INDEX temp, newrec;
35:   int rv = NO_ERROR;
36:   int found;
37:   int srch_rec;
38:
39:   nbr_records++;
40:   if (nbr_records == 1)    /* Is this first record */
41:   {
42:     temp.prev = 0;
43:     temp.next = 0;
44:     temp.data = 1;         /* First rec in data file */
45:     strcpy(temp.code, medium.code);
46:     /* Write Index record */
47:     rv = put_rec(nbr_records, idx_fp, sizeof(temp),
48:                  sizeof(int)*2, (char *) &temp);
49:     if (rv == NO_ERROR)     /* No Error from prior */
50:     {
51:       /* Store data */
52:       rv = put_rec(nbr_records, db_fp, sizeof(medium),
53:                    0, (char *) &medium);
54:     }
55:
56:     /* Update Alpha starting pointer */
57:     if (rv == NO_ERROR)
58:     {
```

continues

Listing 17.7. continued

```
59:       start_rec = nbr_records; /* Update global starting point */
60:       rv = update_header();
61:     }
62:
63:     fflush(idx_fp);
64:     fflush(db_fp);
65:   }
66:   /* Need to search for appropriate place to hold rec */
67:   else
68:   {
69:     found = FALSE;
70:     srch_rec = start_rec;
71:     while (!found)
72:     {
73:       rv = get_rec(srch_rec, idx_fp, sizeof(temp),
74:                    sizeof(int)*2, (char *)&temp);
75:       /* Proceed only if no errors */
76:       if (rv == NO_ERROR)
77:       {
78:         /* Compare two keys - ignoring CASE of keys */
79:         result = stricmp(medium.code, temp.code);
80:         if (result < 0)   /* New key is < this rec key */
81:         {
82:           /* Found place to put it - store info */
83:           found = TRUE;
84:           /* See if this new rec is < start rec  */
85:           /* If so - need to adjust starting ptr */
86:           if (srch_rec == start_rec)
87:              start_rec = nbr_records;
88:
89:           /* First build new Index rec & store new rec */
90:           newrec.prev = temp.prev;  /* Previous record */
91:           newrec.next = srch_rec;   /* Point to record just read */
92:           newrec.data = nbr_records;  /* Pt to data */
93:           strcpy(newrec.code, medium.code);
94:           rv = put_rec(nbr_records, idx_fp, sizeof(newrec),
95:                        sizeof(int)*2, (char *)&newrec);
96:           if (rv == NO_ERROR)
97:           {
98:             /* Update previous rec */
99:             temp.prev = nbr_records;
100:            rv = put_rec(srch_rec, idx_fp, sizeof(temp),
101:                         sizeof(int)*2, (char *)&temp);
102:
103:            /* Now write data - only if no errors */
104:            if (rv == NO_ERROR)
105:            {
106:               /* Now write data */
107:               rv = put_rec(nbr_records, db_fp, sizeof(medium),
```

```
108:                              0, (char *)&medium);
109:             }
110:
111:             /* Now check on updating Next pointer */
112:             /* Is there a ptr pointing to this new rec ?*/
113:             if (rv == NO_ERROR)
114:             {
115:               if (newrec.prev !=0 )
116:               {
117:                 rv = get_rec(newrec.prev, idx_fp, sizeof(temp),
118:                              sizeof(int)*2, (char *)&temp);
119:                 if ( rv == NO_ERROR)
120:                 {
121:                   temp.next = nbr_records;
122:                   rv = put_rec(newrec.prev, idx_fp, sizeof(temp),
123:                                sizeof(int)*2, (char *)&temp);
124:                 }
125:               }
126:             }
127:
128:           }
129:         }
130:         else    /* new rec >= alpha, adjust ptr */
131:         {
132:           if (temp.next == 0) /* End of chain - add to end */
133:           {
134:             found = TRUE;
135:
136:             /* Build Index record */
137:             /* Point backwards to prev rec */
138:             newrec.prev = srch_rec;
139:             newrec.next = 0;        /* There is no next rec */
140:             newrec.data = nbr_records;
141:             strcpy(newrec.code, medium.code);
142:             rv = put_rec(nbr_records, idx_fp, sizeof(newrec),
143:                          sizeof(int)*2, (char *)&newrec);
144:             if (rv == NO_ERROR)
145:             {
146:               /* Update previous rec */
147:               temp.next = nbr_records;
148:               rv = put_rec(srch_rec, idx_fp, sizeof(temp),
149:                            sizeof(int)*2, (char *)&temp);
150:               if (rv == NO_ERROR)
151:               {
152:                 /* Now write data */
153:                 rv = put_rec(nbr_records, db_fp, sizeof(medium),
154:                              0, (char *)&medium);
155:               }
156:             }
157:           }
```

continues

Listing 17.7. continued

```
158:          else  /* Not at end - get next rec ptr */
159:             srch_rec = temp.next;
160:        }
161:      }
162:      else
163:        found = TRUE;  /* Exit because of error */
164:    }   /* End of While */
165:
166:    /* Update starting alpha ptr in hdr */
167:    if (rv == NO_ERROR)
168:    {
169:      rv = update_header();
170:    }
171:
172:    /* Makes sure file gets updated */
173:    fflush(idx_fp);
174:    fflush(db_fp);
175:
176:  }  /* End else */
177:
178:  return(rv);
179: }
```

Analysis This function has the overall objective of adding a record. While the process of physically adding the record is simple, you must also update the index file. It is the updating of the index file that gives this function its size.

This function starts out in the same way as most of the other I/O functions. External declarations are provided for the file pointers, the Medium Code structures, and the header information (lines 16 to 22). In the `add_medium_rec()` function, several additional variables are declared. These include two new `MEDIUM_INDEX` structures, `temp` and `newrec`. In addition to these, a `found` flag is declared.

Line 38 starts the code off by adding one to the total number of records in the file. Because we are adding a record, this will be accurate. Line 39 then checks to see if this is the first record being added to the file. The first record is a special case. If this is the first record, then the values in the `temp` index structure are all set. The record number of the `prev` and `next` records are zero because neither exists. The data record is one because this will be the first record in the data file. Line 44 copies the medium code into the key field of the `temp` index's data field. With this, the temp index structure is set up and ready to be written.

Line 46 uses the `put_rec()` function to write the index record from the `temp` structure that you just set up. The first parameter, `nbr_records` will be one because this is the

first record. The `idx_fp` is the pointer to the index file. The third parameter is the size of the data you want to write. In this case, it's the size of the index structure, temp. The fourth parameter is the offset for the header record. You should remember that the header record is two integers. The last parameter is the `temp` structure that you just set up.

If the writing of the index structure was successful, then the data record is written in line 51. Its parameters are similar. The fourth parameter is zero because the data file doesn't have a header.

In line 55, you are still working with the addition of the first record in the file. If the addition of the first record was successful, then the `start_rec` is set to `nbr_records`. This could have been set to one because you know that there is only one record in the file. Line 59 ensures that the header record is updated. The addition of the first record is completed with the flushing of the index and data files with the `fflush()` function. This ensures that the records are written to the disk and not buffered.

If the record isn't the first record to be added to the files, then the processing is different. The `else` statement starting in line 66 works with this. If the record isn't the first in the file, then you need to find the area in the indexes where the record is to be placed. Consider adding the code `GG` to Figure 17.3. While the GG would be added to the end, the NEXT, PREVIOUS, and DATA REC values all need to be determined. Figure 17.4 shows the result of adding GG to the files in Figure 17.3.

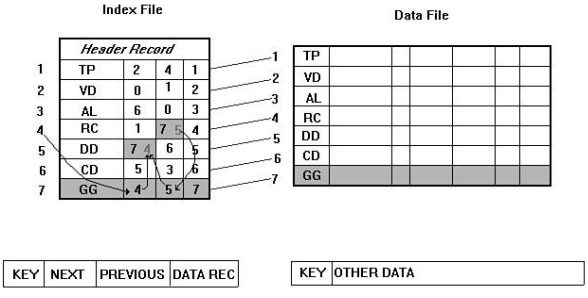

Figure 17.4. *Adding the* `GG` *code.*

As you can see from the figure, the new records are added to the end of both the index and the data file. Before they can be added, the `next` and `prev` record numbers need to be determined. In addition, the `next` and `prev` index records need to be updated with the new record.

The process that will be used to do this is relatively easy to follow. Starting with the first record signified by `start_rec` (line 69), each index record will be read into a temporary index structure, `temp`, (line 72) using a `while` loop, which starts in line 70. If the read is successful, then the code from the read index record is compared to the code from the record that is being added. The `if` statement in line 79 checks to see if the code from the screen is less than the code that was was just read into `temp`. If not, then another index record may need to be read unless the end of the index file has been reached.

The `else` statement to the `if` in line 79 checks the end of file condition. If the end has been reached, then the index structure for the new record to be added can be set up in lines 137 to 139. The `prev` record is set to the current `srch_rec` because it is the last record in the index file. The `next` record is set to zero because there isn't a next record. The data record is the last record in the data file. You should note that the last record in the data file is the same as the number of records, `nbr_records`. After copying the code to the new index structure, the record is written to the file in line 141. If the write is successful, then the previous record needs its next record number updated so that it points to the new last number. This is done in lines 146 to 148. If this update is successful, then the index file is up-to-date. All that's left is to write the new data record. This is done in line 152.

If the code was less than the search record read in line and if it was not the last sorted record in the index file, then the `while` loop starts again. The `while` loop continues until the new code is either the last code in the sort order, or until it is greater than a code that is already in the file.

If the new code is found to be greater than a code already in the file, then the if in line 79 takes effect. This starts by marking the found flag as `TRUE` in line 82 because a position has been found. In line 85, a test is done to see if the new code is the first record in sorted order. If so, then the `start_rec` field is set to the record number of the new code. In any other cases, the starting record number remains what it was.

Lines 89 to 91 set up the new record's index structure, `newrec`. This is similar to what was explained earlier in adding the index record as the last sorted record. The difference is that the previous record is set to the previous record from the last record read. The next record is then set to the last record read. The data record is set to the last record in the data file, which, as stated earlier, is the same as the number of records. If this sounds confusing, then take it one step at a time and use Figure 17.4 as a reference.

Tip: If you are confused, read the previous paragraph slowly and step through the process while looking at Figure 17.4.

Once these values are set, then the new index record is written to the end of the index file (line 93). If there were no problems, then the last search record that was read has its `prev` record number changed to point to the new record (line 98). This record is then updated with a call to `put_rec()`. Line 105 then updates the data record if the previous updates were successful.

This writes or updates all of the records except one. The new record isn't yet a part of the next record pointers. The record that comes before the new record needs to be updated so that its next pointer points to the new number. Lines 114 to 124 do just this. The new records previous record number is used to find the record that needs to be updated. Once found, the record's next record number is set to the last record number and is then updated in line 121.

If the updating is successful, then line 168 updates the header. Lines 172 and 173 force the writes to occur. The function then ends with the return of any error values in line 177.

Deleting a Record: The *del_med_rec()* function

The `del_med_rec()` function is used to remove a record from the medium data file. Listing 17.8 presents the DELMREC.C source file, which contains this function.

Listing 17.8. DELMREC.C. Deleting a medium code.

```
1:   /*=========================================================
2:    * Filename: DELMREC.C
3:    *
4:    * Author:   Bradley L. Jones
5:    *           Gregory L. Guntle
6:    *
7:    * Purpose: Deletes a record from the DB
8:    *          It adjusts the pointers and takes the last DB
9:    *          record and overwrites the record being deleted
10:   *
11:   *=========================================================*/
12:
```

continues

Listing 17.8. continued

```
#include <stdio.h>

#include “records.h”
#include “tyac.h”

/* Global variables */
extern MEDIUM_REC medium;     /* Record structure with data */
extern MEDIUM_REC med_prev;
extern FILE *idx_fp;          /* Main File ptr to data file */
extern FILE *db_fp;           /* Data file */
extern int nbr_records;       /* Total # of recs */
extern int start_rec;         /* Starting rec for alpha */
extern int disp_rec;

int del_med_rec()
{
  int rv = NO_ERROR;
  MEDIUM_INDEX temp, delrec;
  MEDIUM_REC hold_data;
  int chg;
  int srch_rec;

  /* Are we trying to delete a blank record */
  if (disp_rec != 0)  /* No - then proceed */
  {
     /* Get the index info for this rec */
      rv = get_rec(disp_rec, idx_fp, sizeof(delrec),
                   sizeof(int)*2, (char *)&delrec);
     if (rv == NO_ERROR)
     {
       /* Are there any pointers in this rec
          If both pointers are 0 - then this must
          be the only record in the DB           */
       if (delrec.prev == 0 && delrec.next == 0)
       {
         nbr_records = 0;
         disp_rec = 0;
         start_rec = 1;
       }
       else
       {
         chg = FALSE;
         srch_rec = 1;   /* Start at first */

         /* Are we deleting the starting alpha record ? */
         if (disp_rec == start_rec)
         {
           start_rec = delrec.next;     /* Reset pointer */
         }
```

```
62:
63:           /* Go until all the ptrs have been adjusted */
64:           while ( (srch_rec <= nbr_records) && (rv == NO_ERROR) )
65:           {
66:             /* Get record */
67:             rv = get_rec(srch_rec, idx_fp, sizeof(temp),
68:                    sizeof(int)*2, (char *)&temp);
69:             if (rv == NO_ERROR)
70:             {
71:               /* Does this rec prev pointer need to
72:                   be adjusted */
73:               if (temp.prev == disp_rec)
74:               {
75:                  chg = TRUE;
76:                  temp.prev = delrec.prev;
77:               }
78:
79:               /* Since moving last record up - need
80:                  to adjust pointers to last one as well */
81:               if (temp.prev == nbr_records)
82:               {
83:                  chg = TRUE;
84:                  temp.prev = disp_rec;
85:               }
86:
87:               if (temp.next == disp_rec)
88:               {
89:                   chg = TRUE;
90:                   temp.next = delrec.next;
91:               }
92:
93:               /* Since moving last record up - need
94:                  to adjust pointers to last one as well */
95:               if (temp.next == nbr_records)
96:               {
97:                  chg = TRUE;
98:                  temp.next = disp_rec;
99:               }
100:
101:              /* Data Ptr - match last nbr */
102:              if (temp.data == nbr_records)
103:              {
104:                 chg = TRUE;
105:                 temp.data = delrec.data;
106:              }
107:
108:              /* Only update rec if change has been made */
109:              if (chg)
110:              {
111:                /* Is this the rec to overlay ? */
```

continues

Listing 17.8. continued

```
112:                if (srch_rec == nbr_records)
113:                {
114:                   /* Write new index into deleted spot */
115:                   rv = put_rec(disp_rec, idx_fp, sizeof(temp),
116:                        sizeof(int)*2, (char *)&temp);
117:                }
118:                else
119:                {
120:                   /* Rewrite index back to file in same position */
121:                   rv = put_rec(srch_rec, idx_fp, sizeof(temp),
122:                        sizeof(int)*2, (char *)&temp);
123:                }
124:                chg = FALSE;
125:             }
126:
127:             if (rv == NO_ERROR)
128:             {
129:                 /* Go to next record */
130:                 srch_rec++;
131:                 }
132:           }
133:         }
134:
135:         /* DATA    */
136:         /* Delete the data rec - take last rec in data
137:            file and overwrite the rec to be deleted
138:            then search for index pointer that points to
139:            the last rec and update its data pointer.     */
140:         if (rv == 0)
141:         {
142:           /* Only need to adjust if not last rec */
143:           if (nbr_records != disp_rec)
144:           {
145:             /* Get last rec */
146:             rv = get_rec(nbr_records, db_fp, sizeof(medium),
147:                  0, (char *)&hold_data);
148:             if (rv == 0)
149:             {
150:                /* Overwrite data to delete w/Last one */
151:                rv = put_rec(delrec.data, db_fp, sizeof(medium),
152:                   0, (char *)&hold_data);
153:             }
154:           }
155:         }
156:
157:         nbr_records—;   /* Adjust global number of recs */
158:       }
159:       if (rv == NO_ERROR)
160:       {
```

```
161:          rv = update_header();
162:        }
163:
164:        /* Makes sure file gets updated */
165:        fflush(idx_fp);
166:        fflush(db_fp);
167:      }
168:   }
169:   else
170:     boop();  /* Let them know - can't delete blank rec */
171:
172:   return(rv);
173: }
```


Analysis The deleting of records operates differently from the adding of a record. When deleting a record, you need to adjust the index record numbers just as you did when you added a record; however, additional complexity exists. You could simply adjust the index pointers. This would prevent you from being able to access the deleted record; however, this would put holes throughout your files. To prevent these holes, when a record is deleted, the last record in the file is moved forward and placed in the hole that is created. This is exactly what the `del_med_rec()` function does.

Note: If you wanted to be able to undelete a record, then you shouldn't move the last record to the deleted record. By adding a flag to your record, you could signal whether a record had been deleted or not. Additional functions could then read the flag to determine if the record should be used.

The `del_med_rec()` starts with the same external declarations as the `add_medium_rec()` function. In addition, several local variables are declared in lines 29 to 33. The first, `rv`, is the return value in line 29. Two additional `MEDIUM_INDEX` structures are declared, `temp` and `delrec`. The `temp` structure will be used to hold a temporary index structure. The `delrec` index structure will be used to hold the index record for the record that is being deleted. The `chg` field is used as a flag to signal if a change has been made to a given index record. If a change is made to an index record then it will need to be updated in the file. The `srch_rec` variable is used to keep track of the index record that is currently being checked. As you will see later, each record in the index is checked to ensure that it doesn't need to be adjusted.

Line 36 starts the processing by ensuring that the user wasn't adding a new record. If a record hasn't been added, then it can't be deleted. Line 170 beeps the computer with the `boop()` function if the user isn't editing a record. If the user is editing a record, then line 39 retrieves the index record for the currently displayed information. This index record is stored in `delrec`.

Line 46 checks to see if the record being deleted is the only record in the database. You know a record is the only record if both the `prev` and `next` record numbers are zero. If this is the only record, then lines 48 to 50 set the values of `nbr_records`, `disp_rec`, and `start_rec` accordingly. Otherwise, the rest of the function is executed.

If there is more than one record in the file, then processing, starting in line 54, is executed. First, the `chg` flag is defaulted to `FALSE`. The `srch_rec` is initialized to one. Line 58 checks to see if the first record in the file is being deleted. This is done by seeing if the current record, `disp_rec`, is equal to the starting record number, `start_rec`. If the first record is the one being deleted, then the `start_rec` value is adjusted to the delete record's next record. Once this is completed, you are ready to loop through the index file and make all the updates.

The `while` statement in lines 64 to 133 loops through each record in the index file. It continues to loop until all of the records have been processed or until there is an error. Line 67 retrieves the current record, `srch_rec`, into the temporary index structure, `temp`. If there isn't an error, then line 73 checks to see if the current record needs the `prev` record number updated. If the `prev` record number is equal to the record that is being deleted, `disp_rec`, then it needs to be changed. It is changed to the `prev` record from the deleted record in line 76. Additionally, the `chg` flag is set to `TRUE` so that the record will be updated. Because the last record in the file will be moved to fill in the deleted record's position, line 81 is needed. In line 81, the `prev` value is checked to see if it is equal to the last record. If the `prev` record is equal to the last record, then it is set to the position of the record being deleted. This is because the last record will be moved to the deleted position.

Lines 87 to 99 are similar to what was just described. Instead of working with the previous record, these lines work with the next record number pointer.

Line 102 checks the index file's data record number. If this number is pointing to the last data record in the file, then it will also need adjusted. The last data record will also be moved to fill in the deleted record's position. While these lines of code don't change the data record's position, they do go ahead and update the data record flag in the index records. Later in this function, the data record will actually be moved.

Line 109 checks the `chg` flag to see if there was a change to a record. If there was a change, then lines 110 to 123 are executed. If the record that is currently being

processed is the last record in the file (`srch_rec` is equal to `nbr_records`), then you will want to write the record in the deleted records position instead of at the end of the file. This is exactly what line 114 does. You should note that `disp_rec` is the position that is written to instead of the `srch_rec` position. If this isn't the last record in the index file, then line 121 writes the record back to its current location. If this is all successfully accomplished, then line 130 increments the `srch_rec` counter and the next iteration of the while loop occurs.

Once you have looped through all of the index records, the delete will be completely processed. All that will be left to do is delete the actual data record. This is a much simpler process, which is accomplished in lines 140 to 155. Line 143 checks to see if the last record is the one being deleted. If the last record is the record being deleted, then no changes need to be made. If the record being deleted isn't the last record, then the last record is read in line 146. This record is then written in line 151 to the location where the deleted record is. This effectively deletes the record.

In line 157, the number of records is adjusted. This is followed by some housekeeping. In line 161, the header record is updated with the `update_header()` function. Lines 165 and 166 prevent any buffering of data by forcing the writes with the `fflush()` function. With this, the job of deleting a record is complete.

Processing a Record (Next/Previous)

The last function needed before you will be able to compile the medium code without external errors is the `proc_med_rec()` function. This function is presented in Listing 17.9, which contains the PROCMREC.C source file.

Type

Listing 17.9. PROCMREC.C. Processing the next and previous records.

```
1:  /*=========================================================
2:   * Filename: PROCMREC.C
3:   *
4:   * Author:   Bradley L. Jones
5:   *           Gregory L. Guntle
6:   *
7:   * Purpose: Process the requests for getting next/prev
8:   *          records from the MEDIUM DB
9:   *
10:  *=========================================================*/
11:
12: #include <stdio.h>
13: #include <string.h>
```

continues

Listing 17.9. continued

```
#include "records.h"
#include "tyac.h"


/* Global variables required */
extern FILE *idx_fp;
extern FILE *db_fp;
extern nbr_records;       /* Total # of rec for mediums */
extern int start_rec;
extern int disp_rec;

extern MEDIUM_REC medium;
extern MEDIUM_REC med_prev;

int proc_med_rec(int);
int get_med_info(void);

/*-------------------------------------------------------*
 *   proc_med_rec                                        *
 *-------------------------------------------------------*/

int proc_med_rec(int direct)
{
  MEDIUM_INDEX temp;
  int rv = NO_ERROR;

  /* Only do - if there are records in the file */
  if (nbr_records != 0)
  {
    /* Do we just need to display the very first rec */
    if (disp_rec == 0)
    {
       disp_rec = start_rec;
       rv = get_med_info();
    }
    else
    {
       /* Get Index ptrs for record on screen */
       rv = get_rec(disp_rec, idx_fp, sizeof(temp),
                          sizeof(int)*2, (char *)&temp);
       if (rv == NO_ERROR)
      {
        if( direct == NEXT_REC )
        {
          if (temp.next == 0)     /* There is no other rec */
             boop();               /* No more records */
          else
          {
            disp_rec = temp.next;
```

```
63:              rv = get_med_info();
64:            }
65:          }
66:          else  /* Need to go backwards */
67:          {
68:            if (temp.prev == 0)     /* There is no other rec */
69:               boop();               /* No more records */
70:            else
71:            {
72:              disp_rec = temp.prev;
73:              rv = get_med_info();
74:            }
75:          }
76:        }
77:      }
78:    }
79:    else
80:      boop();
81:
82:    return(rv);
83:
84:  }
85:
86:  /* ---------------------------------------------- *
87:   *  get_med_info()                                *
88:   *                                                *
89:   *  Handles getting the Index rec, getting the    *
90:   *  data pointer from the index and then getting  *
91:   *  the appropriate data rec.  It then updates    *
92:   *  the Global MEDIUM record, as well as updating *
93:   *  a copy of the medium record for later use.    *
94:   *                                                *
95:   * ---------------------------------------------- */
96:
97:  int get_med_info()
98:  {
99:    MEDIUM_INDEX temp;
100:   int rv;
101:
102:   /* Get Index record for this Request */
103:   rv = get_rec(disp_rec, idx_fp, sizeof(temp),
104:                sizeof(int)*2, (char *)&temp);
105:   if (rv == NO_ERROR)
106:   {
107:     /* Now get the actual data record to display */
108:     rv = get_rec(temp.data, db_fp, sizeof(medium),
109:                  0, (char *)&medium);
110:     if (rv == NO_ERROR)
111:     {
112:       memcpy(&med_prev, &medium, sizeof(medium));
```

continues

Listing 17.9. continued

```
113:      }
114:    }
115:    return(rv);
116: }
```

Analysis This listing should be easier to follow than the `add_medium_rec()` and `del_med_rec()` listings. This listing enables you to get a next or previous record from the medium data file. Like all the other functions, this function also starts with the list of external declarations before starting into the function in line 35.

Tip: The external declarations could be put into a header file that is included in each of the source files. This cut the amount of code that would need to be written.

This function is called with a value of either NEXT_REC or PREVIOUS_REC, which is stored in the direct parameter. This determines which direction the reading should occur. In line 41, a test is performed to see if there are any records in the file. If there aren't any records, then line 80 beeps the computer. If there are, then the processing moves to line 44. Line 44 determines if a record is currently being displayed. If the `disp_rec` is equal to zero, then there isn't a record being displayed. In this case, the first record is set to be the displayed record and the `get_med_info()` function is called.

The `get_med_info()` function is presented in lines 97 to 116. This function gets the record that is in the `disp_rec` variable. Line 103 gets the index record. If successful, then line 108 gets the data record based on the index's data value. Again, if this read is successful, then this new record is copied into the `med_prev` structure to be used elsewhere. With this completed, the new record is stored in the medium code structure, `medium`. All that is left to do is redisplay the record, which must be done by the calling program.

If `disp_rec` didn't equal zero when this function was called, then you must determine the new record before calling `get_med_info()`. Line 52 starts this process by getting the index record for the currently displayed data record. If the direction is NEXT_REC, then lines 58 to 65 process, otherwise, lines 66 to 74 process. In both cases, the processing is similar. In the case of NEXT_REC being the direction, if the index record's `next` value is zero, the end of the file has been reached. In this case, the `boop()` function

is called to signal the end of the file. If the next value isn't zero, then it is set to the `disp_rec` variable and `get_med_info()` is called to display the record.

Check Point

This completes the functions that are necessary to compile the Medium Screen without any external errors; however, you still have several holes in your entry and edit screen. You haven't added any of the file functions to the action bar. In addition, the action bar contains two additional file functions that haven't been covered, `Clear` and `Find...`. Tomorrow, you'll continue with the Medium Code entry and edit screen. You'll update the action bar with the functions you created today. In addition, you'll add the remaining functions. You'll also work with the albums screen, which has some slightly different processing. You should update the group screen's code as a part of today's exercises.

DO	DON'T
DO complete Exercise 3, which asks you to update the Group's source files.	

Summary

Today was a big day. You began to pull in the most important part of the application—file access. Today, you learned a little background on file I/O and the terms used with it. In addition, you learned about different file modes and different file formats. Using some of what you should have already learned from your C experiences, you were lead in the development of an indexed file. You worked with the Medium Code entry and edit screen to create functions that enabled you to add, update, and delete records from your medium codes file. In addition, you created functions that enabled you to retrieve the next or previous record in alphabetical order from the database. Today's material stops short of completing everything you need to know about indexed files and file I/O in the *Record of Records!* application. Tomorrow will pick up where today left off and complete the topic.

Q&A

Q Why is there a need for text files to translate the CR-LF characters when reading and writing the file?

A The C programming language uses a single character to represent CR-LF. This is the `\n` character that you use all the time. DOS expects two characters for this process, a carriage return and then a line feed. Because DOS and C are expecting different things, there has to be a translation.

Q Are the file I/O functions presented today the best ones to use in an application?

A Not necessarily. Many applications need to be compatible with other custom formats. In these cases, you will need to use routines that write your data in the custom format. Additionally, if your application doesn't need to worry about quick access to records in order, then the indexing may not be necessary.

Q Is the use of global variables good as shown in today's listings?

A You should avoid using global variables whenever possible. At times, it becomes much easier to use global variables than always passing a variable around. Typically, file pointers are declared globally. Because these variable are small, they don't take up much space. You should avoid declaring variables globally whenever possible.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. What is the difference between input and output?
2. What is a field?
3. What is a record?
4. What is a file?

5. What is a database?
6. What is an index file?
7. What is the difference between a file opened in text mode and a file opened in binary mode?
8. A preexisting file is opened with the a mode of “w+b”. What does this mean?
9. What would Figure 13.3 look like if the “AA” code was added?
10. What would Figure 13.3 look like if the “TP” code was deleted?

Exercises

1. Make sure that you updated your TYAC.LIB library file with today’s library functions:

```
put_rec()
get_rec()
open_files()
close_files()
```

2. Create the index structure for the group file and add it to the RECORDS.H header file.
3. **ON YOUR OWN:** Add the file functions such as the ones presented today to the Groups entry and edit screen.