17

17 Examples using structs

This chapter contains a few simple examples illustrating the use of structs in programs.

Using structs with existing libraries

In future programs, you will use structs mainly when communicating with existing C libraries. For example, you will get to write code that uses the graphics primitives on your system. The "applications programmer interface" (API) for the graphics will be defined by a set of C functions (in some cases based on an earlier interface defined by a set of Pascal functions). If you a programming on Unix, you will probably be using the Xlib graphics package, on an Apple system you would be using Quickdraw, and on an Intel system you would use a Windows API. The functions in the API will make extensive use of simple structs. Thus, the Xlib library functions use instances of the following structs:

```
typedef struct {
    short x, y;
} XPoint;

typedef struct {
    short x, y;
    unsigned short width, height;
} XRectangle;
```

and

```
typedef struct {
    short x, y;
    unsigned short width, height;
    short angle1, angle2;
} XArc;
```

So, if you are using the Xlib library, your programs will also define and use variables that are instances of these struct types.

Your own structs?

For new application-specific code you will tend to use variables of class types more often than variables of struct types.

The examples in this chapter are intended merely to illustrate how to declare struct types, use instances of these types, access fields and so forth. The example programs in section 17.3, introduce the idea of a "file of records". These programs have a single struct in memory and a whole set of others in a disk file. The struct in memory gets loaded with data from a requested record on disk. Record files are extremely common in simple business data processing applications, and represent a first step towards the more elaborate "databases" used in industry. You will learn more about "files of records" and "databases" in later subjects. While you can continue to use C/C++ when working with database systems, you are more likely to use specialized languages like COBOL and SQL.

17.1 REORDERING THE CLASS LIST AGAIN

The example in section 13.2 illustrated a "sorting" algorithm that was applied to reorder pupil records. The original used two arrays, `pupils` and `marks`:

```
typedef char Name[40];

Name pupils[] = {
    "Armstrong, Alice S.",
    …
    "Zarra, Daniela"
};

int marks[] = {
    57,
    …
    78
};
```

But the data in the two arrays "belong together" – Miss Armstrong's mark is 57 and she shouldn't be separated from it.

This is a typical case where the introduction of a simple struct leads to code that reflects the problem structure more accurately (and is also slightly clearer to read). The struct has to package together a name and a mark:

```
struct PupilRec {
    Name    fName;
    int     fMark;
};
```

(Naming conventions again: use 'f' as the first character of the name of a data member in all structs and classes that are defined for a program.)

With this struct declared, an initialized array of `PupilRecs` can be defined:

```
PupilRec JuniorYear[] = {
    {"Armstrong, Alice S.",     57 } ,
    {"Azur, Hassan M.",         64 } ,
    {"Bates, Peter",            33 } ,
    …
    {"Zarra, Daniela",          81 }
};
```

The function `MakeOrderedCopy()` can be rewritten to use these `PupilRec` structs:

Arrays of structs as "input" and "output" parameters

```
void MakeOrderedCopy(const PupilRec orig[],int num,
    PupilRec reord[])
{
    int mark_count[101];

    for(int i = 0; i< 101; i++)
           mark_count[i] = 0;

    // Count number of occurrences of each mark
    for(i = 0; i < num; i++) {
           int mark = orig[i].fMark;
           mark_count[mark]++;
    }
    // Make that count of number of pupils with marks less than
    // or equal to given mark
    for(i=1; i<101; i++)
           mark_count[i] += mark_count[i-1];

    for(i=num - 1; i >= 0; i--) {
           int mark = orig[i].fMark;
           int position = mark_count[mark];
           position--; // correct to zero based array
           // copy data
           reord[position] = orig[i];
           mark_count[mark]--;
    }

}
```

Accessing data member of one of structs in the array

Assignment of structs

The function takes two arrays of `PupilRec` structs. The `orig` parameter is the array with the records in alphabetic order; it is an "input parameter" so it is specified as `const`. The `reord` parameter is the array that is filled with the reordered copy of the data.

An expression like:

```
orig[i].fMark
```

illustrates how to access a data member of a chosen element from an array of structures.

Note the structure assignment:

```
reord[position] = orig[i];
```

The compiler "knows" the size of structs so allows struct assignment while assignment of arrays is not allowed.

17.2 POINTS AND RECTANGLES

The following structs and functions illustrate the functionality that you will find in the graphics libraries for your system. Many of the basic graphics functions will use points and rectangles; for example, a line drawing function might take a "start point" and an "end point" as its parameters, while a `DrawOval()` function might require a rectangle to frame the oval.

The graphics packages typically use short integers to represent coordinate values. A "point" will need two short integer data fields:

```
struct Point {
    short fx;
    short fy;
};
```

A struct to represent a rectangle can be defined in a number of alternative ways, including:

```
struct Rectangle {
    short ftop;
    short fleft;
    short fwidth;
    short fheight;
};
```

and

```
struct Rectangle {
    Point fcorner;
    short fwidth;
    short fheight;
};
```

Provided that `struct Point` has been declared before `struct Rectangle`, it is perfectly reasonable for `struct Rectangle` to have data members that are instances of type `Point`. The second declaration will be used for the rest of the examples in this section.

The graphics libraries define numerous functions for manipulating points and rectangles. The libraries would typically include variations of the following functions:

```
int Point_In_Rect(const Point& pt, const Rectangle& r);
    Returns "true" (1) if point pt lies with Rectangle r.

int Equal_Points(const Point& p1, const Point& p2);
    Returns "true" if points p1 and p2 are the same.

void Add_Point(const Point& p1, const Point& p2, Point& res);
    Changes the "output parameter" res to be the 'vector sum'
           of points p1 and p2.

Point MidPoint(const Point& p1, const Point& p2);
    Returns the mid point of the given points.

int ZeroRect(const Rectangle& r)
    Return "true" if r's width and height are both zero.

Rectangle UnionRect(const Rectangle& r1, const Rectangle& r2);
    Returns the smallest circumscribing rectangle of the given
           rectangles r1 and r2.

Rectangle IntersectRect(const Rectangle& r1,
    const Rectangle& r2);
    Returns a rectangle that represents the intersection of the
           given rectangles, or a zero rectangle if they don't
           intersect.

Rectangle Points2Rect(const Point& p1, const Point& p2);
    Returns a rectangle that includes both points p1 and p2
           within its bounds or at least on its perimeter
```

These functions are slightly inconsistent in their prototypes, some return structs as results while others have output parameters. This is deliberate. The examples are meant to illustrate the different coding styles. Unfortunately, it is also a reflection of most of the graphics libraries! They tend to lack consistency. If you were trying to design a graphics library you would do better to decide on a style; functions like `Add_Point()` and `MidPoint()` should either all have struct return types or all have output parameters. In this case, there are reasons to favour functions that return structs. Because points and rectangles are both small, it is acceptable for the functions to return structs as results. Code using functions that return structs tends to be a little more readable than code using functions with struct output parameters.

This example is like the "curses" and "menu selection" examples in Chapter 12. We need a header file that describes the facilities of a "points and rectangles" package, and an implementation file with the code of the standard functions. Just so as to illustrate the approach, some of the simpler functions will be defined as "inline" and their definitions will go in the header file.

The header file would as usual have its contents bracketed by conditional compilation directives:

```
#ifndef __MYCOORDS__
#define __MYCOORDS__

the interesting bits

#endif
```

As explained in section 12.1, these conditional compilation directives protect against the possibility of erroneous multiple inclusion.

The structs would be declared first, then the function prototypes would be listed. Any "inline" definitions would come toward the end of the file:

Header file "mycoords.h"

```
#ifndef __MYCOORDS__
#define __MYCOORDS__

struct Point {
    short fx;
    short fy;
};

struct Rectangle {
    …
};

int Point_In_Rect(const Point& pt, const Rectangle& r);

…

inline int Equal_Points(const Point& p1, const Point& p2)
{
    return ((p1.fx == p2.fx) && (p1.fy == p2.fy));
}

inline int ZeroRect(const Rectangle& r)
{
    return ((r.fwidth == 0) && (r.fheight == 0));
}

#endif
```

inline functions defined in the header file

"Inline" functions have to be defined in the header. After all, if the compiler is to replace calls to the functions by the actual function code it must know what that code is. When compiling a program in another file that uses these functions, the compiler only knows the information in the #included header.

The definitions of the other functions would be in a separate mycoords.cp file:

```
#include "mycoords.h"
```

```
int Point_In_Rect(const Point& pt, const Rectangle& r)
{
//  Returns "true" if point pt lies with Rectangle r.
    if((pt.fx < r.fcorner.fx) || (pt.fy < r.fcorner.fy))
           return 0;

    int dx = pt.fx - r.fcorner.fx;
    int dy = pt.fy - r.fcorner.fy;

    if((dx > r.fwidth) || (dy > r.fheight))
           return 0;

    return 1;
}
```

Note multiply qualified names of data members

A rectangle has a `Point` data member which itself has int `fx, fy` data members. The name of the data member that represents the x-coordinate of the rectangle's corner is built up from the name of the rectangle variable, e.g. `r1`, qualified by the name of its point data member `fcorner`, qualified by the name of the field, `fx`.

Function `Add_Point()` returns its result via the res argument:

```
void Add_Point(const Point& p1, const Point& p2, Point& res)
{
//  Changes the "output parameter" res to be the 'vector sum'
//         of points p1 and p2.
    res.fx = p1.fx + p2.fx;
    res.fy = p1.fy + p2.fy;
}
```

while `MidPoint()` returns its value via the stack (it might as well use `Add_Point()` in its implementation).

```
Point MidPoint(const Point& p1, const Point& p2)
{
//  Returns the mid point of the given points.
    Point m;
    Add_Point(p1, p2, m);
    m.fx /= 2;
    m.fy /= 2;

    return m;
}
```

Function `Points2Rect()` is typical of the three `Rectangle` functions:

```
Rectangle Points2Rect(const Point& p1, const Point& p2)
{
//  Returns a rectangle that includes both points p1 and p2
//         within its bounds or at least on its perimeter
```

```
    Rectangle r;
    int left, right, top, bottom;

    if(p1.fx < p2.fx) { left = p1.fx; right = p2.fx; }
    else { left = p2.fx; right = p1.fx; }

    if(p1.fy < p2.fy) { top = p1.fy; bottom = p2.fy; }
    else { top = p2.fy; bottom = p1.fy; }

    r.fcorner.fx = left;
    r.fcorner.fy = top;

    r.fwidth = right - left;
    r.fheight = bottom - top;

    return r;
}
```

If you were developing a small library of functions for manipulation of points and rectangles, you would need a test program like:

```
#include <iostream.h>
#include "mycoords.h"

int main()
{
    Point p1;
    p1.fx = 1;
    p1.fy = 7;

    Rectangle r1;
    r1.fcorner.fx = -3;
    r1.fcorner.fy = -1;
    r1.fwidth = 17;
    r1.fheight = 25;

    if(Point_In_Rect(p1, r1))
           cout << "Point p1 in rect" << endl;
    else cout << "p1 seems to be lost" << endl;

    Point p2;
    p2.fx = 11;
    p2.fy = 9;
    Point p3;
    Add_Point(p1, p2, p3);

    cout << "I added the points, getting p3 at ";
    cout << p3.fx << ", " << p3.fy << endl;

    Point p4;
    p4.fx = 12;
```

```
    p4.fy = 16;

    if(Equal_Points(p3, p4))
           cout << "which is where I expected to be" << endl;
    else cout << "which I find surprising!" << endl;

…
…

    Rectangle r4 = UnionRect(r2, r3);
    cout << "I made rectangle r4 to have a corner at ";
    cout << r4.fcorner.fx << ", " << r4.fcorner.fy << endl;
    cout << "and width of " << r4.fwidth <<
                  ", height " << r4.fheight << endl;

    Rectangle r5 = IntersectRect(r4, r1);
    cout << "I made rectangle r5 to have a corner at ";
    cout << r5.fcorner.fx << ", " << r5.fcorner.fy << endl;
    cout << "and width of " << r5.fwidth <<
                  ", height " << r5.fheight << endl;

    return 0;
}
```

The test program would have calls to all the defined functions and would be organized to check the results against expected values as shown.

You may get compilation errors relating to `struct Point`. Some IDEs automatically include the system header files that define the graphics functions, and these may already define some other `struct Point`. Either find how to suppress this automatic inclusion of headers, or change the name of the structure to `Pt`.

17.3 FILE OF RECORDS

Rather than writing their own program, anyone who really wants to keep files of business records would be better off using the "database" component of one of the standard "integrated office packages". But someone has to write the "integrated office package" of the future, maybe you. So you had better learn how to manipulate simple files of records.

The idea of a file of records was briefly previewed in section 9.6 and is again illustrated in Figure 17.1.

A file record will be exactly the same size as a memory based struct. Complete structs can be written to, and read from, files. The i/o transfer involves simply copying bytes (there is no translation between internal binary forms of data and textual strings). The operating system is responsible for fitting the records into the blocks on disk and keeping track of the end of the file.

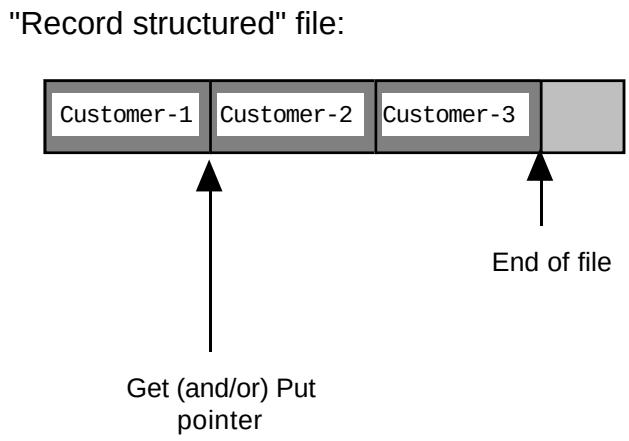

Figure 17.1 A file of "customer records"

Sequential access

Files of record can be processed sequentially. The file can be opened and records read one by one until the end of file marker is reached. This is appropriate when you want to all records processed. For example, at the end of the month you might want to run through the complete file identifying customers who owed money so that letters requesting payment could be generated and dispatched.

Random access

Files of records may also be processed using "random access". Random access does not mean that any data found at random will suffice (this incorrect explanation was proposed by a student in an examination). The "randomness" lies in the sequence in which records get taken from file and used. For example, if you had a file with 100 customer records, you might access them in the "random" order 25, 18, 49, 64, 3, ….

File systems allow "random access" because you can move the "get" (read) or "put" (write) pointer that the operating system associates with a file before you do the next read or write operation. You can read any chosen record from the file by moving the get pointer to the start of that record. This is easy provided you know the record number (imagine that the file is like an array of records, you need the index number into this array). You simply have to multiply the record number by the size of the record, this gives the byte position where the "get pointer" has to be located.

You have to be able to identify the record that you want. You could do something like assign a unique identifying number (in the range 0…?) to each customer and require that this is specified in all correspondence, or you could make use of an auxiliary table (array) of names and numbers.

You would want to use "random access" if you were doing something like taking new orders as customers came to, or phoned the office. When a customer calls, you want to be able to access their record immediately, you don't want to read all the preceding records in the file. (Of course reading the entire file wouldn't matter if you have only 100 records; but possibly you have ambitions and wish to grow to millions.)

New iostream and fstream features

Working with record files and random access requires the use of some additional facilities from the iostream and fstream libraries.

fstream

First, the file that holds the data records will be an "input-output" file. If you need to do something like make up an order for a customer, you need to read (input) the customer's record, change it, then save the changed record by writing it back to file (output). Previously we have used `ifstream` objects (for inputs from file) and `ofstream` objects (for outputs to file), now we need an `fstream` object (for bidirectional i/o). We will need an `fstream` variable:

```
fstream    gDataFile;
```

which will have to be connected to a file by an `open()` request:

```
gDataFile.open(gFileName, ios::in | ios::out );
```

The open request would specify `ios::in` (input) and `ios::out` (output); in some environments, the call to `open()` might also have to specify `ios::binary`. As this file is used for output, the operating system should create the file if it does not already exist.

Positioning the get/put pointers

Next, we need to use the functions that move the "get" and "put" pointers. (Conceptually, these are separate pointers that can reference different positions in the file; in most implementations, they are locked together and will always refer to the same position.) The get pointer associated with an input stream can be moved using the `seekg()` function; similarly, the put pointer can be moved using the `seekp()` function. These functions take as arguments a byte offset and a reference point. The reference point is defined using an enumerated type defined in the iostream library; it can be `ios::beg` (start of file), `ios::cur` (current position), or `ios::end` (end of the file). So, for example, the call:

```
gDataFile.seekg(600, ios::beg);
```

would position the get pointer 600 bytes after the beginning of the file; while the call:

```
gDataFile.seekp(0, ios::end);
```

would position the put pointer at the end of the file.

Finding your current position in a file

The libraries also have functions that can be used to ask the positions of these pointers; these are the `tellp()` and `tellg()` functions. You can find the size of a file, and hence the number of records in a record file, using the following code:

```
gDataFile.seekg(0, ios::end);
long pos = gDataFile.tellg();
gNumRecs = pos / sizeof(Customer);
```

The call to `seekg()` moves the get pointer to the end of the file; the call to `tellg()` returns the byte position where the pointer is located, and so gives the length of the file (i.e. the number of bytes in the file). The number of records in the file can be obtained by dividing the file length by the record size.

Read and write transfers

Data bytes can be copied between memory and file using the read and write functions:

```
read(void *data, int size);
write(const void *data, size_t size);
```

(The prototypes for these functions may be slightly different in other versions of the iostream library. The type `size_t` is simply a typedef equivalent for unsigned int.) These functions need to be told the location in memory where the data are to be placed (or copied from) and the number of bytes that must be transferred.

Memory address "Pointers"

The first argument is a `void*`. In the case of `write()`, the data bytes are copied from memory to the disk so the memory data are unchanged, so the argument is a const `void*`. These `void*` parameters are the first example that we've seen of *pointers.* (Actually, the string library uses pointers as arguments to some functions, but we disguised them by changing the function interfaces so that they seemed to specify arrays).

A pointer variable is a variable that contains the memory address of some other data element. Pointers are defined as derived types based on built in or programmer defined struct (and class) types:

```
int         *iptr;
double      *dptr;
Rectangle   *rptr;
```

and, as a slightly special case:

```
void        *ptr_to_somedata;
```

These definitions make `iptr` a variable that can hold the address of some integer data item, `dptr` a variable that holds the address of a double, and `rptr` a variable that holds the address where a `Rectangle` struct is located.

The variable, `ptr_to_somedata`, is a `void*`. This means that it can hold the address of a data item of any kind. (Here void is not being used to mean empty, it is more that it is "unknown" or at least "unspecified").

As any C programmer who has read this far will have noted, we have been carefully avoiding pointers. Pointers are all right when you get to know them, but they can be cruel to beginners. From now on, almost all your compile time and run-time errors are going to relate to the use of pointers.

But in these calls to the `read()` and `write()` functions, there are no real problems. All that these functions require is the memory address of the data that are involved in

the transfer. In this example, that is going to mean the address of some variable of a struct type `Customer`.

The & "address-of" operator

In C and C++, you can get the address of any variable by using the `&` "address-of" operator. Try running the following program (addresses are by convention displayed in hexadecimal rather than as decimal numbers):

```
float pi = 3.142;

int array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int main()
{
    int data = 101;
    void   *ptr;

    ptr = &data;
    cout << "data is at " << hex << ptr << endl;

    ptr = &pi;
    cout << "pi is at " << hex << ptr << endl;

    ptr = &(array[2]);
    cout << "array[2] is at " << hex << ptr << endl;

    ptr = &(array[3]);
    cout << "array[3] is at " << hex << ptr << endl;
    return 0;
}
```

You should be able to relate the addresses that you get to the models that you now have for how programs and data are arranged in memory. (If you find it hard to interpret the hexadecimal numbers, switch back to decimal outputs.)

The `&` operator is used to get the addresses that must be passed to the `read()` and `write()` functions. If we have a variable:

```
Customer c;
```

we can get it loaded with data from a disk file using the call:

```
gDataFile.read(&c, sizeof(Customer));
```

or we can save its contents to disk by the call:

```
gDataFile.write(&c, sizeof(Customer));
```

(Some compilers and versions of the iostream library may require "`(char*)`" before the `&` in these calls. The `&` operator and related matters are discussed in Chapter 20.)

Specification

Write a program that will help a salesperson keep track of customer records.
The program is to:

1 Maintain a file with records of customers. These records are to include the customer name, address, postcode, phone number, amount owed, and details of any items on order.

2 Support the following options:
- list all customers
- list all customers who currently owe money
- show the record of a named customer
- create a record for a new customer
- fill in an order for a customer (or clear the record for goods delivered and paid for)
- quit.

3 Have a list of all products stocked, along with their costs, and should provide a simple way for the salesperson to select a product by name when making up a customer order.

The program is not to load the entire contents of the data file into memory. Records are to be fetched from the disk file when needed. The file will contain at most a few hundred records so sophisticated structures are not required.

Design

First Iteration

The overall structure of the program will be something like:

```
Open the file and do any related initializations
Interact with the salesperson, processing commands until
    the Quit command is entered
Close the file
```

These steps obviously get expanded into three main functions.

The "Close File" function will probably be trivial, maybe nothing more than actually closing the file. Apart from actually opening the file (terminating execution if it won't open) the "Open File" function will probably have to do additional work, like determining how many records exist. During design of the main processing loop, we may identify data that should be obtained as the file is opened. So, function "open file" should be left for later consideration.

The main processing loop in the routine that works with the salesperson will be along the following lines:

```
"Run the shop"
    quitting = false
    while (not quitting)
           prompt salesperson for a command
           use command number entered to select
•                 list all customers
•                 list debtors
    …
•                 deal with quit command, i.e. quitting = true
```

Obviously, each of the options like "list all customers" becomes a separate function.

Several of these functions will have to "get" a record from the file or "put" a record to file, so we can expect that they will share "get record", "put record" and possibly some other lower level functions.

Second Iteration

The UT functions developed in Chapter 12 (selection from menu, and keyword lookup) might be exploited. Obviously, the menu selection routine could handle the task of getting a command entered by the salesperson. In fact, if that existing routine is to be used there will be no further design work necessary for "Run the shop". The sketched code is easy to implement.

The "list …" functions

So, the next major task is identifying the way the functions will handle the tasks like listing customers and getting orders. The routines for listing all customers and listing debtors are going to be very similar:

```
"list all customers"
    if there are no customers
           just print some slogan and return

    for i = 0 , i < number of customers,. i++
           get customer record i
           print details of customer


"list debtors"
    initialize a counter to zero

    for i = 0 , i < number of customers,. i++
           get customer record i
           if customer owes money
                  print details of customer
                  increment counter

    if counter is zero report "no debtors"
```

You might be tempted to fold these functions into one, using an extra boolean argument to distinguish whether we are interested in all records or just those with debts. However, if the program were later made more elaborate you would probably find greater differences in the work done in these two cases (e.g. you might want the "list

debtors" operation to generate form letters suggesting that payment would be appreciated). Since the functions can be expected to become increasingly different, you might as well code them as two functions from the start.

The sketched pseudo-code uses for loops to work through the file and relies on a (global?) variable that holds the number of records. As noted in the introduction to this section, it is easy to work out the number of records in a file like this; this would get done when the file is opened.

The two functions could work with a local variable of a struct type "Customer". This would get passed as a reference parameter to a "get record" function, and as a "const reference" to a "print details" function.

Auxiliary "get record" and "print details" functions

The two additional auxiliary functions, "get record" and "print details", also need to be sketched:

```
"get record"
    convert record number to byte position
    read from file into record
    if i/o error
           print error message and exit

"print details"
    output name, address, postcode, phone number

    if amount owed
           print the amount
    else ? (either blank, or "nothing owing")

    if any items on order
           print formatted list of items
```

The specification didn't really tie down how information was to be displayed; we can chose the formats when doing the implementation.

Functions for other processing options

The other required functions are those to add a new customer record to the file, show the record of a customer identified by name, and place an order for a customer (the specification is imprecise, but presumably this is again going to be a customer identified by name).

Adding a customer should be easy. The "add customer" routine would prompt the salesperson for the name, address, postcode, and phone number of the new customer. These data would be filled into a struct, then the struct would be written to the end of the existing file. The routine would have to update the global counter that records the number of records so that the listing routines would include the new record.

```
"add customer"
    ? check any maximum limits on the number of customers

    prompt for data like name, filling in a struct,

    set amount owed, and items on order to zero
```

```
    write the struct to the file

    update number of customers
```

The file itself would not limit the number of records (well, not until it got to contain so many millions of records that it couldn't fit on a disk). But limits could (and in this case do) arise from other implementation decisions.

Writing the struct to file would be handled by a "put record" function that matches the "get record":

```
"put record"
    convert record number to byte position
    copy data from memory struct to file
    if i/o error
           print error message and exit
```

Getting the record for a named customer

The other two functions, "show customer" and "record order", both apparently need to load the record for a customer identified by name. The program could work reading each record from the file until it found the correct one. This would be somewhat costly even for little files of a few hundred records. It would be easier if the program kept a copy of the customer names in memory. This should be practical, the names would require much less storage than the complete records.

It would be possible for the "open file" routine to read all existing records, copying the customer names into an array of names. This only gets done when the program starts up so later operations are not slowed. When new records are added to the file, the customer name gets added to this array.

The memory array could hold small structures combining a name and a record number; these could be kept sorted by name so allowing binary search. This would necessitate a sort operation after the names were read from file in the "open file" routine, and an "insert" function that would first find the correct place for a new name and then move other existing names to higher array locations so as to make room for it.

Alternatively, the array could just contain the names; the array indexes would correspond to the record numbers. Since the names wouldn't be in any particular order, the array would have to be searched linearly to find a customer. Although crude, this approach is quite acceptable in this specific context. These searches will occur when the salesperson enters a "show customer" or "place order" command. The salesperson will be expecting to spend many seconds reading the displayed details, or minutes adding new orders. So a tiny delay before the data appear isn't going to matter. A program can check many hundreds of names in less than a tenth of a second; so while a linear search of the table of names might be "slow" in computer terms, it is not slow in human terms and the times of human interaction are going to dominate the workings of this program.

Further, we've got that `UT_PickKeyWord()` function. It searches an array of "words", either finding the match or listing the different possible matches. This would

actually be quite useful here. If names can be equated with the UT_Words we can use the pick key word function to identify a name from its first few characters. Such an "intelligent" interactive response would help the salesperson who would only have to enter the first few characters of a name before either getting the required record or a list of the name and similar names.

So, decisions: Customer names will be UT_Words (this limits them to less than 15 characters which could be a problem), things like addresses might as well be UT_Texts. There will be a global array containing all the names. Functions that require the salesperson to select a customer will use the keyword search routine (automatically getting its ability to handle abbreviations.

These decisions lead to the following design sketches for the two remaining processing options:

```
"show customer"
    use keyword picker function to prompt for data (customer name)
           and find its match in names array
           (returns index of match)

    use Get Record to get the record corresponding to index

    Print Details of record got from file
```

and

```
"record order"
    use keyword picker function to prompt for data (customer name)
           and find its match in names array
           (returns index of match)

    use get record to get the record corresponding to index

    reset customer record to nothing ordered, nothing owed

    loop
           get next order item
           update amount owed
    until either no more order items or maximum number ordered

    put updated record back in file
```

The loop getting items in function "record order" is once again a candidate for promotion to being a separate function. If it had to do all the work, function "record order" would become too complex. Its role should be one of setting up the context in which some other function can get the order items. So, its sketch should be revised to:

```
"record order"
    use keyword picker function to prompt for data (customer name)
           and find its match in names array
```

```
            (returns index of match)

        use get record to get the record corresponding to index

        get items

        put updated record back in file
```

This new "get items" function now has to be planned. The specification stated that this routine should either make up an order for a customer or clear the record for goods delivered and paid for. This might not prove ideal in practice because it means that you can't deliver partial orders, and you can't accept supplementary orders; but it will do to start with. It means that the "get items" routine should start by clearing any existing record of amount owed and items ordered. Then the routine should ask the user whether any item is to be ordered. While the reply is "yes", the routine should get details of the item, add it to the record and its cost to the amount owed, and then again ask whether another item is to be ordered. The Customer struct can have an array to hold information about ordered items. Since this array will have a fixed size, the loop asking for items would need to terminate if the array gets filled up.

The specification requires the program to have a table of goods that can be ordered and a convenient mechanism allowing the salesperson to enter the name of a chosen article. The key word picking function can again be pressed into service. There will need to be a global array with the names of the goods articles, and an associated array with their costs.

An initial sketch for "get items" is:

```
"get items"
    change amount owing data member of record to zero

    ask (YesNo() function) whether another item to be ordered

    while item to be ordered and space left
            use keyword picker function to prompt for
                    data (item name) and find its match
                    in goods array
                    (returns index of match)

            copy name of item into record
            update amount owed by cost of item

            update count of items ordered

            ask (YesNo()) whether another item needed
```

This has sketch has identified another function, `YesNo()`, that gets a yes/no input from the user.

Open File revisited

Earlier consideration of the open file function was deferred. Now, we have a better idea of what it must do. It should open the file (or terminate the program if the file won't open). It should then determine the number of records. Finally, it should scan through all records copying the customer names into an array in memory.

The array for names has a fixed size. So there will be a limit on the number of customers. This will have to be catered for in the "add customer" function.

Third iteration through the design

What is a Customer? It is about time to make some decisions regarding the data.

A Customer had better be a struct that has data members for:

1 customer name (already decided to use a UT_Word, i.e. up to about 15 characters);

2 customer address (a UT_Text, i.e. up to 60 characters);

3 postcode and phone, these could also be UT_Words;

4 amount owing (a double);

5 a count of items on order;

6 an array with the names of the items on order, need to fix a size for this.

Other fields might need to be added later. The following structs declarations should suffice:

```
#ifndef __MYCUSTOMER__
#define __MYCUSTOMER__

#include "UT.h"

struct Date {
    int fDay, fMonth, fYear;
};

const int kMAXITEMS = 5;

struct Customer {
    UT_Word       fName;
    UT_Text       fAddress;
    UT_Word       fPostcode;
    UT_Word       fDialcode;
    UT_Word       fOrders[kMAXITEMS];
    int           fNumOrder;
    double        fAmountOwing;
    Date          fLastOrder;
};

#endif
```

An extra `Date` struct has been declared and a `Date` data member has been included in the `Customer` struct. This code doesn't use dates; one of the exercises at the end of the chapter involves implementing a code to handle dates.

As a `Customer` uses `UT_Word` and `UT_Text` this header file has to include the UT.h header that contains the typedef defining these character array types.

The main implementation file is going to contain a number of global arrays:

- `gStock[]`, an array of `UT_Words` with names of items stocked by shop;
- `gItemCosts[]`, an array of doubles with the costs of items;
- `gCommands[]`, an array with the phrases that describe the commands that the salesperson can select;
- `gNames[]`, an array to hold customer names.

Other global (or filescope) variables will be needed for the file name, the fstream object that gets attached to the file, and for a number of integer counters (number of records, number of commands, number of items in stock list).

The functions have already been considered in detail and don't require further iterations of design. Their prototypes can now be defined:

Function prototypes

```
void GetRecord(Customer& rec, int cNum);

void PutRecord(Customer& rec, int cNum);

void PrintDetails(const Customer& c);

void ShowCustomer(void);

void ListAll(void);

void ListDebtors(void);

void AddCustomer(void);

int YesNo(void);

void GetItems(Customer& c);

void RecordOrder(void);

void RunTheShop(void);

void OpenTheDataFile(void);

void CloseTheDataFile(void);

int main();
```

Other linked files

The code written specifically for this problem will have to be linked with the UT code from Chapter 12 so as to get the key word and menu selection functions.

Header files needed

Quite a large number of systems header files will be needed. The programs uses both iostream and fstream libraries for files. The exit() function will get used to terminate the program if an i/o error occurs with the file accesses, so stdlib is also needed. Strings representing names will have to be copied using strcpy() from the string library. The "yes/no" function will have to check characters so may need the ctype header that defines standard functions like tolower(). Error checking might use assert.

Implementation

The implementation file will start with the #includes:

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <ctype.h>

#include "UT.h"
#include "mycustomer.h"
```

(If you think about it carefully, you will see that we end up #including UT.h twice; that sort of thing happens very easily and it is why we have those #ifdef … #endif brackets on all header files).

The declarations of globals come next. Several are initialized:

```
const char gFileName[] = "CustRec.XXX";
const int kMAXCUSTOMERS = 200;

fstream    gDataFile;
int        gNumRecs;
UT_Word    gNames[kMAXCUSTOMERS];

UT_Word gStock[] = {
    "Floppy disks",
    …
    "Marker pens",
    "Laser pointer"
};

double gItemCosts[] = {
    18.50,  /* disks $18.50 per box */
    …
    6.50,   /* pens */
```

```
    180.0  /* laser pointer */
};

int gNStock = sizeof(gStock) / sizeof(UT_Word);

UT_Text gCommands[] = {
    "Quit",
    "List all customers",
    "List all debtors",
    "Add Customer",
    "Show Customer",
    "Record Order"
};

int gNCommands = sizeof(gCommands) / sizeof(UT_Text);
```

In many ways it would be better to introduce a new struct that packages together an item name and its cost. Then we could have an array of these "costed item" structs which would reduce the chance of incorrect costs being associated with items. However, that would preclude the use of the existing keyword functions that require a simple array of `UT_Word`s.

The function definitions come next:

```
void GetRecord(Customer& rec, int cNum)
{
    /* cNum is customer number (0-based) */
    /* convert into offset into file */
    long where = cNum * sizeof(Customer);
    gDataFile.seekg(where, ios::beg);
    gDataFile.read(&rec, sizeof(Customer));

    if(!gDataFile.good()) {
           cout << "Sorry, can't read the customer file"
                  << endl;
           exit(1);
           }
}

void PutRecord(Customer& rec, int cNum)
{
    long where = cNum * sizeof(Customer);
    gDataFile.seekp(where, ios::beg);
    gDataFile.write(&rec, sizeof(Customer)); //maybe
(char*)&rec

    if(!gDataFile.good()) {
           cout << "Sorry, can't write to the customer file"
                  << endl;
           exit(1);
           }
```

```
}
```

Terminating the program after an i/o error may seem a bit severe, but really there isn't much else that we can do in those circumstances.

```
void PrintDetails(const Customer& c)
{
    cout << "----" << endl;
    cout << "Customer Name : " << c.fName << endl;
    cout << "Address       : " << c.fAddress << ", "
                 << c.fPostcode << endl;
    cout << "Phone         : " << c.fDialcode << endl;
    if(c.fAmountOwing > 0.0)
           cout << "Owes          : $" << c.fAmountOwing
                        << endl;
    else cout << "Owes nothing" << endl;
    if(c.fNumOrder == 1) cout << "On order: " << c.fOrders[0]
                        << endl;
    else
    if(c.fNumOrder >0) {
           cout << "On order" << endl;
           for(int j = 0; j < (c.fNumOrder-1); j++)
                  cout << c.fOrders[j] << ", ";
           cout << "and ";
           cout << c.fOrders[c.fNumOrder-1] << endl;
           }
    cout << "---" << endl;
}
```

The `PrintDetails()` function gets a little elaborate, but it is trying to provide a nice listing of items with commas and the word "and" in the right places.

```
void ShowCustomer(void)
{
    UT_Text aPrompt = "Customer Name : ";
    int who = UT_PickKeyWord(aPrompt, gNames, gNumRecs);
    if(who < 0)
           return;
    Customer c;
    GetRecord(c, who);
    PrintDetails(c);
}
```

There is one problem in using the "pick keyword" function. If the salesperson enters something that doesn't match the start of any of the names, the error message is "there is no keyword …". Possibly the "pick keyword" function should have been designed to take an extra parameter that would be used for the error message.

```
void ListAll(void)
```

```
{
    if(gNumRecs == 0) {
           cout << "You have no customers" << endl;
           return;
           }

    for(int i = 0; i < gNumRecs; i++) {
           Customer c;
           GetRecord(c, i);
           PrintDetails(c);
           }
}

void ListDebtors(void)
{
    int count = 0;
    for(int i = 0; i < gNumRecs; i++) {
           Customer c;
           GetRecord(c, i);
           if(c.fAmountOwing > 0.0) {
                        PrintDetails(c);
                        count++;
                        }
           }
    if(count == 0) cout << "Nothing owed" << endl;
}
```

Function `AddCustomer()` checks whether the program's array of names is full and prevents extra names being added. The input statements use `getline()`. This is because addresses are going to be things like "234 High Street" which contain spaces. If we tried to read an address with something like `cin >> c.fAddress`, the address field would get "234", leaving "High …" etc to confuse the input to post code. The routine isn't robust; you can cause lots of troubles by entering names that are too long to fit in the specified data member.

```
void AddCustomer(void)
{
    if(gNumRecs == kMAXCUSTOMERS) {
           cout << "Sorry, you will have to edit program "
                        "before it can handle" << endl;
           cout << "   more customers." << endl;
           return;
           }

    Customer c;
    // N.B. This input routine is "unsafe"
    // there are no checks successful reads etc

    cout << "Name       : ";
    cin.getline(c.fName, UT_WRDLENGTH-1, '\n');
```

```
    cout << "Address    : ";
    cin.getline(c.fAddress, UT_TXTLENGTH-1, '\n');

    cout << "Post code  : ";
    cin.getline(c.fPostcode, UT_WRDLENGTH-1, '\n');

    cout << "Phone      : ";
    cin.getline(c.fDialcode, UT_WRDLENGTH-1, '\n');

    c.fNumOrder = 0;
    c.fAmountOwing = 0.0;

    PutRecord(c, gNumRecs);
    strcpy(gNames[gNumRecs], c.fName);

    gNumRecs++;
}

int YesNo(void)
{
    char ch;
    cout << "Order an item? (Y or N)";
    cin >> ch;
    ch = tolower(ch);
    return (ch == 'y');
}

void GetItems(Customer& c)
{
    c.fAmountOwing = 0.0;

    int count = 0;
    while(YesNo() && (count <kMAXITEMS)) {
           UT_Text aPrompt = "Identify Type of Goods";
           int which =
                  UT_PickKeyWord(aPrompt, gStock, gNStock);

           strcpy(c.fOrders[count], gStock[which]);
           c.fAmountOwing += gItemCosts[which];
           count++;
           }
    c.fNumOrder = count;
}
```

Look a bug in `GetItems()`! Can you spot it? It isn't serious, the program won't crash. But it makes the user interaction clumsy.

If you can't spot the bug, run the code and try to order more than the limit of five items. You should then observe a certain clumsiness.

The bug can be fixed by a trivial change to the code.

```
void RecordOrder(void)
{
    UT_Text aPrompt = "Enter Customer Name";
    int who = UT_PickKeyWord(aPrompt, gNames, gNumRecs);

    Customer c;
    GetRecord(c, who);

    GetItems(c);

    PutRecord(c, who);

}

void RunTheShop(void)
{
    UT_Text aPrompt = "Command";
    int quitting = 0;
    while(!quitting) {
           int command =
                  UT_MenuSelect(aPrompt,gCommands,
                         gNCommands, 0);

           // Need to consume any trailing spaces or newlines
           // after the command number
           char ch;
           cin.get(ch);
           while(ch != '\n')
                  cin.get(ch);

           switch(command) {
case 0:           /* Quit */
                  quitting = 1;
                  break;
case 1:           /* List all customers */
                  ListAll();
                  break;
case 2:           /* List all debtors */
                  ListDebtors();
                  break;
case 3:           /* Add Customer */
                  AddCustomer();
                  break;
case 4:           /* Show Customer */
                  ShowCustomer();
                  break;
case 5:           /* Record Order */
                  RecordOrder();
                  break;
           }
    }
```

```
}
```

The `RunTheShop()` function has one complication. The salesperson has to enter a number when picking the required menu option. The digits get read but any trailing spaces and newlines will still be waiting in the input stream. These could confuse things if the next function called needed to read a string, a character, or a complete line. So the input stream is cleaned up by reading characters until get the '\n' at the end of the input line.

```
void OpenTheDataFile(void)
{
    // Open the file, allow creation if not already there
    gDataFile.open(gFileName, ios::in | ios::out );
           // may need also ios::binary
    if(!gDataFile.good()) {
           cout << "? Couldn't open the file.  Sorry." << endl;
           exit(1);
           }

    gDataFile.seekg(0, ios::end);
    long pos = gDataFile.tellg();
    gNumRecs = pos / sizeof(Customer);

    assert(gNumRecs <= kMAXCUSTOMERS);
    for(int i = 0; i < gNumRecs; i++) {
           Customer c;
           GetRecord(c, i);
           strcpy(gNames[i], c.fName);
           }
}
```

Logically, there is no way that the file could hold more than the maximum number of records (the file has to be created by this program and the "add customer" function won't let it happen).

Don't believe it. Murphy's law applies. The program can go wrong if the file is too large, so it will go wrong. (Something will happen like a user concatenating two data files to make one large one.) Since the program will overwrite arrays if the file is too large, it better not even run. Hence the `assert()` checking the number of records.

```
void CloseTheDataFile(void)
{
    gDataFile.close();
}

int main()
{
    OpenTheDataFile();
    RunTheShop();
```

```
    CloseTheDataFile();
    return 0;
}
```

On the whole, the program runs OK:

```
Command
Enter option number in range 1 to 6, or ? for help
6
Enter Customer Name
Ga
Possible matching keywords are:
Gates, B.
Garribaldi, J
Gamble,P
Gam
i.e. Gamble,P
Order an item? (Y or N)y
Identify Type of Goods
Las
i.e. Laser pointer
Order an item? (Y or N)Y
Identify Type of Goods
Tone
i.e. Toner
Order an item? (Y or N)n
Command
Enter option number in range 1 to 6, or ? for helpEnter option
number in range 1 to 6, or ? for help
3
----
Customer Name : Gates, B.
Address       : The Palace, Seattle, 923138
Phone         : 765 456 222
Owes          : $186.5
On order
Laser pointer, and Marker pens
---
----
Customer Name : Gamble,P
Address       : 134 High St, 89143
Phone         : 1433
Owes          : $220
On order
Laser pointer, and Toner
---
```

EXERCISES

1 Fix the "bug" in `GetItems() from 17.3`.

2 Change the way that the Customer records program handles the names to the alternative approach described in the text (sorted array of name-index number structures).

3 Implement code to support the Date data structure and arrange that Customer orders include a Date.