CHAPTER 4

Fundamental File Structure Concepts

CHAPTER OBJECTIVES

- Introduce file structure concepts dealing with
  - Stream files,
  - Reading and writing fields and records,
  - Field and record boundaries,
  - Fixed-length and variable-length fields and records, and
  - Packing and unpacking records and buffers.
- Present an object-oriented approach to file structures
  - Methods of encapsulating object value and behavior in classes,
  - Classes for buffer manipulation,
  - Class hierarchy for buffer and file objects and operations,
  - Inheritance and virtual functions, and
  - Template classes.

CHAPTER OUTLINE

4.1 Field and Record Organization

When we build file structures, we are making it possible to make data *persistent*. That is, one program can create data in memory and store it in a file and another program can read the file and re-create the data in its memory. The basic unit of data is the *field*, which contains a single data value. Fields are organized into aggregates, either as many copies of a single field (an *array*) or as a list of different fields (a *record*). Programming language type definitions allows us to define the structure of records. When a record is stored in memory, we refer to it as an *object* and refer to its fields as *members*. When that object is stored in a file, we call it simply a record.

In this chapter we investigate the many ways that objects can be represented as records in files. We begin by considering how to represent fields and continue with representations of aggregates. The simplest representation is with a file organized as a stream of bytes.

4.1.1 A Stream File

Suppose the objects we wish to store contain name and address information about a collection of people. We will use objects of class `Person`, from Section 1.5, "Using Objects in C++," to store information about individuals. Figure 4.1 (and file `writestr.cpp`) gives a C++ function (`operator <<`) to write the fields of a `Person` to a file as a stream of bytes.

File `writstrm.cpp` in Appendix D includes this output function, together with a function to accept names and addresses from the keyboard and a main program. You should compile and run this program. We use it as the basis for a number of experiments, and you can get a better feel for the differences between the file structures we are discussing if you perform the experiments yourself.

The following names and addresses are used as input to the program:

Mary Ames	Alan Mason
123 Maple	90 Eastgate
Stillwater, OK 74075	Ada, OK 74820

When we list the output file on our terminal screen, here is what we see:

```
-esMary123 MapleStillwaterOK74075MasonAlan90 EastgateAdaOK74820
```

The program writes the information out to the file precisely as specified, as a stream of bytes containing no added information. But in meeting our specifications, the program creates a kind of reverse Humpty-Dumpty problem. Once we put all that information together as a single byte stream, there is no way to get it apart again.

```
=tream & operator << (ostream & outputFile, Person & p)
   insert (write) fields into stream
 outputFile << p.LastName
    << p.FirstName
    << p.Address
    << p.City
    << p.State
    << p.ZipCode;
 return outputFile;
```

gure 4.1 Function to write (<<) a `Person` as a stream of bytes.

We have lost the integrity of the fundamental organizational units of our input data; these fundamental units are not the individual characters but meaningful aggregates of characters, such as "Ames" or "123 Maple." When we are working with files, we call these fundamental aggregates *fields*. A field is *the smallest logically meaningful unit of information in a file.*[1]

A field is a logical notion; it is a *conceptual tool*. A field does not necessarily exist in any physical sense, yet it is important to the file's structure. When we write out our name and address information as a stream of undifferentiated bytes, we lose track of the fields that make the information meaningful. We need to organize the file in some way that lets us keep the information divided into fields.

4.1.2 Field Structures

There are many ways of adding structure to files to maintain the identity of fields. Four of the most common methods follow:

- Force the fields into a predictable length.
- Begin each field with a length indicator.
- Place a *delimiter* at the end of each field to separate it from the next field.
- Use a "keyword = value" expression to identify each field and its contents.

Method 1: Fix the Length of Fields

The fields in our sample file vary in length. If we force the fields into predictable lengths, we can pull them back out of the file simply by counting our way to the end of the field. We can define a `struct` in C or a `class` in C++ to hold these fixed-length fields, as shown in Fig. 4.2. As you can see, the only difference between the C and C++ versions is the use of the keyword `struct` or `class` and the designation of the fields of class `Person` as `public` in C++.

1. Readers should not confuse the terms *field* and *record* with the meanings given to them by some programming languages, including Ada. In Ada, a record is an aggregate data structure that can contain members of different types, where each member is referred to as a field. As we shall see, there is often a direct correspondence between these definitions of the terms and the fields and records that are used in files. However, the terms *field* and *record* as we use them have much more general meanings than they do in Ada.

In C:

```
struct Person{
   char last [11];
   char first [11];
   char address [16];
   char city [16];
   char state [3];
   char zip [10];
};
```

In C++:

```
class Person { public:
   char last [11];
   char first [11];
   char address [16];
   char city [16];
   char state [3];
   char zip [10];
};
```

Figure 4.2 Definition of record to hold person information.

In this example, each field is a character array that can hold a string value of some maximum size. The size of the array is one larger than the longest string it can hold. This is because strings in C and C++ are stored with a terminating 0 byte. The string "Mary" requires five characters to store. The functions in `string.h` assume that each string is stored this way. A fixed-size field in a file does not need to add this extra character. Hence, an object of class `Person` can be stored in 61 bytes: 10+10+15+15+2+9.

Using this kind of fixed-field length structure changes our output so it looks like that shown in Fig. 4.3(a). Simple arithmetic is sufficient to let us recover the data from the original fields.

One obvious disadvantage of this approach is that adding all the padding required to bring the fields up to a fixed length makes the file much larger. Rather than using 4 bytes to store the last name "Ames," we use 10. We can also encounter problems with data that is too long to fit into the allocated amount of space. We could solve this second problem by fixing all the fields at lengths that are large enough to cover all cases, but this would make the first problem of wasted space in the file even worse.

Because of these difficulties, the fixed-field approach to structuring data is often inappropriate for data that inherently contains a large amount of variability in the length of fields, such as names and addresses. But there are kinds of data for which fixed-length fields are highly appropriate. If every field is already fixed in length or if there is very little variation in field lengths, using a file structure consisting of a continuous stream of bytes organized into fixed-length fields is often a very good solution.

```
Ames      Mary    123 Maple       Stillwater    OK74075
Mason     Alan    90 Eastgate     Ada           OK74820
```

(a)

```
04Ames04Mary09123 Maple10Stillwater02OK0574075
05Mason04Alan1190 Eastgate03Ada02OK0574820
```

(b)

```
Ames|Mary|123 Maple|Stillwater|OK|74075|
Mason|Alan|90 Eastgate|Ada|OK|74820|
```

(c)

```
last=Ames|first=Mary|address=123 Maple|city=Stillwater|
state=OK|zip=74075|
```

(d)

Figure 4.3 Four methods for organizing fields within records. (a) Each field is of fixed length. (b) Each field begins with a length indicator. (c) Each field ends with a delimiter |. (d) Each field is identified by a key word.

Method 2: Begin Each Field with a Length Indicator

Another way to make it possible to count to the end of a field is to store the field length just ahead of the field, as illustrated in Fig. 4.3(b). If the fields are not too long (less than 256 bytes), it is possible to store the length in a single byte at the start of each field. We refer to these fields as *length-based.*

Method 3: Separate the Fields with Delimiters

We can also preserve the identity of fields by separating them with delimiters. All we need to do is choose some special character or sequence of characters that will not appear within a field and then *insert* that delimiter into the file after writing each field.

The choice of a delimiter character can be very important as it must be a character that does not get in the way of processing. In many instances *white-space characters* (blank, new line, tab) make excellent delimiters because they provide a clean separation between fields when we list them

on the console. Also, most programming languages include I/O statements which, by default, assume that fields are separated by white space.

Unfortunately, white space would be a poor choice for our file since blanks often occur as legitimate characters within an address field. Therefore, instead of white space we use the vertical bar character as our delimiter, so our file appears as in Fig. 4.3(c). Readers should modify the original stream-of-bytes program, `writstrm.cpp`, so that it places a delimiter after each field. We use this delimited field format in the next few sample programs.

Method 4: Use a "Keyword = Value" Expression to Identify Fields

This option, illustrated in Fig. 4.3(d), has an advantage that the others do not: it is the first structure in which a field provides information about itself. Such *self-describing* structures can be very useful tools for organizing files in many applications. It is easy to tell which fields are contained in a file, even if we don't know ahead of time which fields the file is supposed to contain. It is also a good format for dealing with missing fields. If a field is missing, this format makes it obvious, because the keyword is simply not there.

You may have noticed in Fig. 4.3(d) that this format is used in combination with another format, a delimiter to separate fields. While this may not always be necessary, in this case it is helpful because it shows the division between each value and the keyword for the following field.

Unfortunately, for the address file this format also wastes a lot of space: 50 percent or more of the file's space could be taken up by the keywords. But there are applications in which this format does not demand so much overhead. We discuss some of these applications in Section 5.6: "Portability and Standardization."

4.1.3 Reading a Stream of Fields

Given a modified version of `operator <<` that uses delimiters to separate fields, we can write a function that overloads the extraction operator (operator >>) that reads the stream of bytes back in, breaking the stream into fields and storing it as a `Person` object. Figure 4.4 contains the implementation of the extraction operation. Extensive use is made of the `istream` method `getline`. The arguments to `getline` are a character array to hold the string, a maximum length, and a delimiter. `Getline` reads up to the first occurrence of the delimiter, or the end-of-line,

```
istream & operator >> (istream & stream, Person & p)
{ // read delimited fields from file
   char delim;
   stream.getline(p.LastName, 30,'|');
   if (strlen(p.LastName)==0) return stream;
   stream.getline(p.FirstName,30,'|');
   stream.getline(p.Address,30,'|');
   stream.getline(p.City, 30,'|');
   stream.getline(p.State,15,'|');
   stream.getline(p.ZipCode,10,'|');
   return stream;
}
```

Figure 4.4 Extraction operator for reading delimited fields into a `Person` object.

whichever comes first. A full implementation of the program to read a stream of delimited `Person` objects in C++, `readdel.cpp`, is included in Appendix D.

When this program is run using our delimited-field version of the file containing data for Mary Ames and Alan Mason, the output looks like this:

```
Last Name   'Ames'
First Name  'Mary'
Address     '123 Maple'
City        'Stillwater'
State       'OK'
Zip Code    '74075'
Last Name   'Mason'
First Name  'Alan'
Address     '90 Eastgate'
City        'Ada'
State       'OK'
Zip Code    '74820'
```

Clearly, we now preserve the notion of a field as we store and retrieve this data. But something is still missing. We do not really think of this file as a stream of fields. In fact, the fields are grouped into records. The first six fields form a record associated with Mary Ames. The next six are a record associated with Alan Mason.

4.1.4 Record Structures

A *record* can be defined as *a set of fields that belong together when the file is viewed in terms of a higher level of organization.* Like the notion of a field, a record is another conceptual tool. It is another level of organization that we impose on the data to preserve meaning. Records do not necessarily exist in the file in any physical sense, yet they are an important logical notion included in the file's structure.

Most often, as in the example above, a record in a file represents a structured data object. Writing a record into a file can be thought of as saving the state (or value) of an object that is stored in memory. Reading a record from a file into a memory resident object restores the state of the object. It is our goal in designing file structures to facilitate this transfer of information between memory and files. We will use the term *object* to refer to data residing in memory and the term *record* to refer to data residing in a file.

In C++ we use *class* declarations to describe objects that reside in memory. The members, or attributes, of an object of a particular class correspond to the fields that need to be stored in a file record. The C++ programming examples are focused on adding methods to classes to support using files to preserve the state of objects.

Following are some of the most often used methods for organizing the records of a file:

- Require that the records be a predictable number of bytes in length.
- Require that the records be a predictable number of fields in length.
- Begin each record with a length indicator consisting of a count of the number of bytes that the record contains.
- Use a second file to keep track of the beginning byte address for each record.
- Place a delimiter at the end of each record to separate it from the next record.

Method 1: Make Records a Predictable Number of Bytes (Fixed-Length Records)

A *fixed-length record file* is one in which each record contains the same number of bytes. This method of recognizing records is analogous to the first method we discussed for making fields recognizable. As we will see in

the chapters that follow, fixed-length record structures are among the most commonly used methods for organizing files.

The C structure `Person` (or the C++ class of the same name) that we define in our discussion of fixed-length fields is actually an example of a fixed-length *record* as well as an example of fixed-length fields. We have a fixed number of fields, each with a predetermined length, that combine to make a fixed-length record. This kind of field and record structure is illustrated in Fig. 4.5(a).

It is important to realize, however, that fixing the number of bytes in a record does not imply that the size or number of fields in the record must be fixed. Fixed-length records are frequently used as containers to hold variable numbers of variable-length fields. It is also possible to mix fixed- and variable-length fields within a record. Figure 4.5(b) illustrates how variable-length fields might be placed in a fixed-length record.

Method 2: Make Records a Predictable Number of Fields

Rather than specify that each record in a file contain some fixed number of bytes, we can specify that it will contain a fixed number of fields. This is a good way to organize the records in the name and address file we have been looking at. The program in `writstrm.cpp` asks for six pieces of information for every person, so there are six contiguous fields in the file for each record (Fig. 4.5c). We could modify `readdel` to recognize fields simply by counting the fields *modulo* six, outputting record boundary information to the screen every time the count starts over.

Method 3: Begin Each Record with a Length Indicator

We can communicate the length of records by beginning each record with a field containing an integer that indicates how many bytes there are in the rest of the record (Fig. 4.6a on page 128). This is a commonly used method for handling variable-length records. We will look at it more closely in the next section.

Method 4: Use an Index to Keep Track of Addresses

We can use an *index* to keep a byte offset for each record in the original file. The byte offsets allow us to find the beginning of each successive record and compute the length of each record. We look up the position of a record in the index then seek to the record in the data file. Figure 4.6(b) illustrates this two-file mechanism.

Ames	Mary	123 Maple	Stillwater	OK74075
Mason	Alan	90 Eastgate	Ada	OK74820

(a)

Ames¦Mary¦123 Maple¦Stillwater¦OK¦74075¦ ← Unused space →

Mason¦Alan¦90 Eastgate¦Ada¦OK¦74820¦ ← Unused space →

(b)

Ames¦Mary¦123 Maple¦Stillwater¦OK¦74075¦Mason¦Alan¦90 Eastgate¦Ada¦OK . . .

(c)

igure 4.5 Three ways of making the lengths of records constant and predictable. a Counting bytes: fixed-length records with fixed-length fields. (b) Counting bytes: xed-length records with variable-length fields. (c) Counting fields: six fields per ecord.

Method 5: Place a Delimiter at the End of Each Record

This option, at a record level, is exactly analogous to the solution we used to keep the *fields* distinct in the sample program we developed. As with fields, the delimiter character must not get in the way of processing. Because we often want to read files directly at our console, a common choice of a record delimiter for files that contain readable text is the end-of-line character (carriage return/new-line pair or, on Unix systems, just a new-line character: \n). In Fig 4.6(c) we use a # character as the record delimiter.

4.1.5 A Record Structure That Uses a Length Indicator

None of these approaches to preserving the idea of a *record* in a file is appropriate for all situations. Selection of a method for record organization depends on the nature of the data and on what you need to do with it. We begin by looking at a record structure that uses a record-length field at the beginning of the record. This approach lets us preserve the *variability* in the length of records that is inherent in our initial stream file.

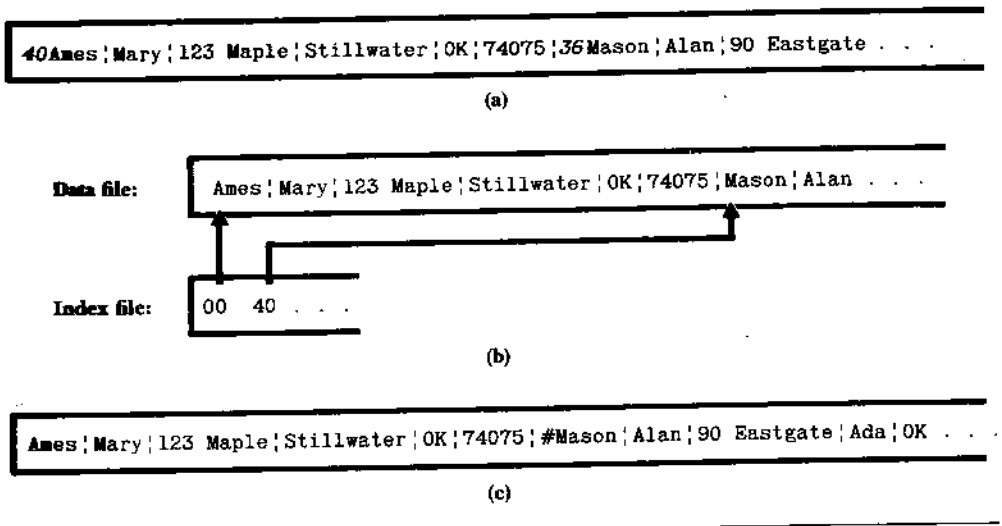

Figure 4.6 Record structures for variable-length records. (a) Beginning each record with a length indicator. (b) Using an index file to keep track of record addresses. (c) Placing the delimiter # at the end of each record.

Writing the Variable-Length Records to the File

Implementing variable-length records is partially a matter of building on the program in `writstrm.cpp` that we created earlier in this chapter, but it also involves addressing some new problems:

- If we want to put a length indicator at the *beginning* of every record (before any other fields), we must know the sum of the lengths of the fields in each record before we can begin writing the record to the file. We need to accumulate the entire contents of a record in a *buffer* before writing it out.
- In what form should we write the record-length field to the file? As a binary integer? As a series of ASCII characters?

The concept of buffering is one we run into again and again as we work with files. In this case, the buffer can simply be a character array into which we place the fields and field delimiters as we collect them. A C++ function `WritePerson`, written using the C string functions, is found in Figure 4.7. This function creates a buffer; fills it with the delimited field values using `strcat`, the string concatenation function; calculates the length of the of the buffer using `strlen`; then writes the buffer length and the buffer to the output stream.

```
const int MaxBufferSize = 200;
int WritePerson (ostream & stream, Person & p)
   char buffer [MaxBufferSize]; // create buffer of fixed size
   strcpy(buffer, p.LastName); strcat(buffer,"|");
   strcat(buffer, p.FirstName); strcat(buffer,"|");
   strcat(buffer, p.Address);  strcat(buffer,"|");
   strcat(buffer, p.City);  strcat(buffer,"|");
   strcat(buffer, p.State);  strcat(buffer,"|");
   strcat(buffer, p.ZipCode);  strcat(buffer,"|");
   short length=strlen(buffer);
   stream.write (&length, sizeof(length)); // write length
   stream.write (&buffer, length);
```

Figure 4.7 Function `WritePerson` writes a variable-length, delimited buffer to a file.

Representing the Record Length

The question of how to represent the record length is a little more difficult. One option would be to write the length in the form of a 2-byte binary integer before each record. This is a natural solution in C, since it does not require us to go to the trouble of converting the record length into character form. Furthermore, we can represent much bigger numbers with an integer than we can with the same number of ASCII bytes (for example, 32 767 versus 99). It is also conceptually interesting, since it illustrates the use of a fixed-length binary field in combination with variable-length character fields.

Another option is to convert the length into a character string using formatted output. With C streams, we use `fprintf`; with C++ stream classes, we use the overloaded insertion operator (<<):

```
fprintf (file, "%d ", length); // with C streams
stream << length << ' '; // with C++ stream classes
```

Each of these lines inserts the length as a decimal string followed by a single blank that functions as a delimiter.

In short, it is easy to store the integers in the file as fixed-length, 2-byte fields containing integers. It is just as easy to make use of the automatic conversion of integers into characters for text files. File structure design is always an exercise in flexibility. Neither of these approaches is correct; good design consists of choosing the approach that is most *appropriate* for a given language and computing environment. In the functions included

```
40 Ames|Mary|123 Maple|Stillwater|OK|74075|36
Mason|Alan|90 Eastgate|Ada|OK|74820
```

Figure 4.8 Records preceded by record-length fields in character form.

in program `readvar.cpp` in Appendix D, we have implemented our record structure using binary field to hold the length. The output from an implementation with a text length field is shown in Fig. 4.8. Each record now has a record length field preceding the data fields. This field is delimited by a blank. For example, the first record (for Mary Ames) contains 40 characters, counting from the first *A* in "Ames" to the final delimiter after "74075," so the characters *4* and *0* are placed before the record, followed by a blank.

Since the implementation of variable-length records presented in Section 4.2 uses binary integers for the record length, we cannot simply print it to a console screen. We need a way to interpret the noncharacter portion of the file. In the next section, we introduce the file dump, a valuable tool for viewing the contents of files. But first, let's look at how to read in any file written with variable-length records.

Reading the Variable-Length Records from the File

Given our file structure of variable-length records preceded by record-length fields, it is easy to write a program that reads through the file, record by record, displaying the fields from each of the records on the screen. The program must read the length of a record, move the characters of the record into a buffer, then break the record into fields. The code to read and break up the record is included in function `ReadVariablePerson` in Fig. 4.9. The function is quite simple because it takes advantage of the extraction operator that was previously defined for reading directly from a file. The implementation of `ReadVariablePerson` may be hard to understand because it uses features of C++ that we haven't yet covered. In particular, class `istrstream` (input string stream) is a type of input stream that uses the same operators as other input streams but has its value stored in a character string instead of in a file. The extraction operation of Figure 4.4 works just as well on a string stream as it does on a file stream. This is a wonderful result of the use of inheritance in C++ classes. We use inheritance extensively in later C++ classes, but that will have to wait for Section 4.3.

```
int ReadVariablePerson (istream & stream, Person & p)
    read a variable sized record from stream and store it in p
  short length;
  stream . read (&length, sizeof(length));
  char * buffer = new char[length+1];// create buffer space
  stream . read (buffer, length);
  buffer [length] = 0; // terminate buffer with null
  istrstream strbuff (buffer); // create a string stream
  strbuff >> p; // use the istream extraction operator
  return 1;
```

Figure 4.9 Function `ReadVariablePerson` that reads a variable-sized `Person` record.

4.1.6 Mixing Numbers and Characters: Use of a File Dump

File dumps give us the ability to look inside a file at the actual bytes that are stored there. Consider, for instance, the record-length information in the text file that we were examining a moment ago. The length of the Ames record, the first one in the file, is 40 characters, including delimiters. The actual bytes stored *in the file* look like the representation in Fig. 4.10(a). In the mixed binary and text implementation, where we choose to represent the length field as a 2-byte integer, the bytes look like the representation in Fig. 4.10(b).

As you can see, the *number* 40 is not the same as the set of characters *4* and *0*. The 1-byte hex value of the *binary integer* 40 is 0x28; the hex values of the *characters* 4 and *0* are 0x34 and 0x30. (We are using the C language convention of identifying hexadecimal numbers through the use of the prefix 0x.) So, when we are storing a number in ASCII form, it is the hex values of the *ASCII characters* that go into the file, not the hex value of the number itself.

Figure 4.10(b) shows the byte representation of the number 40 stored as an integer (this is called storing the number in *binary* form, even though we usually view the output as a hexadecimal number). Now the hexadecimal value stored in the file is that of the number itself. The ASCII characters that happen to be associated with the number's hexadecimal value have no obvious relationship to the number. Here is what the version of the file that uses binary integers for record lengths looks like if we simply print it on a terminal screen:

```
(Ames | Mary | 123 Maple | Stillwater | OK | 74075 | $Mason|Alan|…
```

0x28 is ASCII code for '('
Blank, since '\0' is unprintable.

0x28 is ASCII code for '('
Blank; '\0' is unprintable.

The ASCII representations of characters and numbers in the actual record come out nicely enough, but the binary representations of the length fields are displayed cryptically. Let's take a different look at the file, this time using the Unix dump utility od. Entering the Unix command

```
od -xc filename
```

produces the following:

```
Offset      Values
0000000  \0   (   A   m   e   s   |   M   a   r   y   |   1   2   3
           0028    416d    6573    7c4d    6172    797c    3132    3320
0000020  M   a   p   l   e   |   S   t   i   l   l   w   a   t   e   r
           4d61    706c    657c    5374    696c    6c77    6174    6572
0000040  |   O   K   |   7   4   0   7   5   |  \0   $   M   a   s   o
           7c4f    4b7c    3734    3037    357c    0024    4d61    736f
0000060  n   |   A   l   a   n   |   9   0       E   a   s   t   g   a
           6e7c    416c    616e    7c39    3020    4561    7374    6761
0000100  t   e   |   A   d   a   |   O   K   |   7   4   8   2   0   |
           7465    7c41    6461    7c4f    4b7c    3734    3832    307c
```

As you can see, the display is divided into three different kinds of data. The column on the left labeled `Offset` gives the offset of the first byte of the row that is being displayed. The byte offsets are given in octal form; since each line contains 16 (decimal) bytes, moving from one line to the next adds 020 to the range. Every pair of lines in the printout contains inter-

	Decimal value of number	Hex value stored in bytes	ASCII character form
(a) 40 stored as ASCII chars:	40	34 | 30	4 0
(b) 40 stored as a 2-byte integer:	40	00 | 28	'\0' '('

Figure 4.10 The number 40, stored as ASCII characters and as a short integer.

pretations of the bytes in the file in ASCII and hexadecimal. These representations were requested on the command line with the `-xc` flag (x = hex; c = character).

Let's look at the first row of ASCII values. As you would expect, the data placed in the file in ASCII form appears in this row in a readable way. But there are hexadecimal values for which there is no printable ASCII representation. The only such value appearing in this file is 0 x 00. But there could be many others. For example, the hexadecimal value of the number 500 000 000 is 0x1DCD6500. If you write this value out to a file, an `od` of the file with the option `-xc` looks like this:

```
0000000 \035\315 e \0
          1dcd 6500
```

The only printable byte in this file is the one with the value 0x65 (`e`). `Od` handles all of the others by listing their equivalent octal values in the ASCII representation.

The hex dump of this output from `writrec` shows how this file structure represents an interesting mix of a number of the organizational tools we have encountered. In a single record we have both binary and ASCII data. Each record consists of a fixed-length field (the byte count) and several delimited, variable-length fields. This kind of mixing of different data types and organizational methods is common in real-world file structures.

A Note about Byte Order

If your computer is a PC or a computer from DEC, such as a VAX, your octal dump for this file will probably be different from the one we see here These machines store the individual bytes of numeric values in a reverse order. For example, if this dump were executed on a PC, using the MS-DOS `debug` command, the hex representation of the first 2-byte value in the file would be 0x2800 rather than 0x0028.

This reverse order also applies to long, 4-byte integers on these machines. This is an aspect of files that you need to be aware of if you expect to make sense out of dumps like this one. A more serious consequence of the byte-order differences among machines occurs when we move files from a machine with one type of byte ordering to one with a different byte ordering. We discuss this problem and ways to deal with it in Section 5.6, "Portability and Standardization."

4.2 Using Classes to Manipulate Buffers

Now that we understand how to use buffers to read and write information, we can use C++ classes to encapsulate the pack, unpack, read, and write operations of buffer objects. An object of one of these buffer classes can be used for output as follows: start with an empty buffer object, pack field values into the object one by one, then write the buffer contents to an output stream. For input, initialize a buffer object by reading a record from an input stream, then extract the object's field values, one by one. Buffer objects support only this behavior. A buffer is not intended to allow modification of packed values nor to allow pack and unpack operations to be mixed. As the classes are described, you will see that no direct access is allowed to the data members that hold the contents of the buffer. A considerable amount of extra error checking has been included in these classes.

There are three classes defined in this section: one for delimited fields, one for length-based fields, and one for fixed-length fields. The first two field types use variable-length records for input and output. The fixed-length fields are stored in fixed-length records.

4.2.1 Buffer Class for Delimited Text Fields

The first buffer class, `DelimTextBuffer`, supports variable-length buffers whose fields are represented as delimited text. A part of the class definition is given as Fig. 4.11. The full definition is in file `deltext.h` in Appendix E. The full implementations of the class methods are in `deltext.cpp`. Operations on buffers include constructors, read and write, and field pack and unpack. Data members are used to store the delimiter used in pack and unpack operations, the actual and maximum number of bytes in the buffer, and the byte (or character) array that contains the value of the buffer. We have also included an extension of the class `Person` from Fig. 4.2 to illustrate the use of buffer objects.

The following code segment declares objects of class `Person` and class `DelimTextBuffer`, packs the person into the buffer, and writes the buffer to a file:

```
Person MaryAmes;
DelimTextBuffer buffer;
buffer . Pack (MaryAmes . LastName);
buffer . Pack (MaryAmes . FirstName);
```

```
lass DelimTextBuffer
  public:
     DelimTextBuffer (char Delim = '|', int maxBytes = 1000);
     int Read (istream & file);
     int Write (ostream & file) const;
     int Pack (const char * str, int size = -1);
     int Unpack (char * str);
rivate:
     char Delim; // delimiter character
     char * Buffer; // character array to hold field values
     int BufferSize; // current size of packed fields
     int MaxBytes; // maximum number of characters in the
ffer
     int NextByte; // packing/unpacking position in buffer
```

igure 4.11 Main methods and members of class `DelimTextBuffer`.

```
...
buffer . Pack (MaryAmes . ZipCode);
buffer . Write (stream);
```

This code illustrates how default values are used in C++. The declaration of object `buffer` has no arguments, but the only constructor for `DelimTextBuffer` has two parameters. There is no error here, since the constructor declaration has default values for both parameters. A call that omits the arguments has the defaults substituted. The following two declarations are completely equivalent:

```
elimTextBuffer buffer; // default arguments used
elimTextBuffer buffer ('|', 1000); // arguments given explicitly
```

Similarly, the calls on the `Pack` method have only a single argument, so the second argument (`size`) takes on the default value -1.

The `Pack` method copies the characters of its argument `str` to the buffer and then adds the delimiter character. If the `size` argument is not -1, it specifies the number of characters to be written. If `size` is -1, the C function `strlen` is used to determine the number of characters to write. The `Unpack` function does not need a size, since the field that is being unpacked consists of all of the characters up to the next instance of the delimiter. The implementation of `Pack` and `Unpack` utilize the private member `NextByte` to keep track of the current position in the buffer. The `Unpack` method is implemented as follows:

```
:: DelimTextBuffer :: Unpack (char * str)
  extract the value of the next field of the buffer

  int len = -1; // length of packed string
  int start = NextByte; // first character to be unpacked
  for (int i = start; i < BufferSize; i++)
    if (Buffer[i] == Delim)
      {len = i - start; break;}
  if (len == -1) return FALSE; // delimeter not found
  NextByte += len + 1;
  if (NextByte > BufferSize) return FALSE;
  strncpy (str, &Buffer[start], len);
  str [len] = 0; // zero termination for string
  return TRUE;
```

The `Read` and `Write` methods use the variable-length strategy as described in Section 4.1.6. A binary value is used to represent the length of the record. `Write` inserts the current buffer size, then the characters of the buffer. `Read` clears the current buffer contents, extracts the record size, reads the proper number of bytes into the buffer, and sets the buffer size:

```
:: DelimTextBuffer :: Read (istream & stream)

  Clear ();
  stream . read ((char *)&BufferSize, sizeof(BufferSize));
  if (stream.fail()) return FALSE;
  if (BufferSize > MaxBytes) return FALSE; // buffer overflow
  stream . read (Buffer, BufferSize);
  return stream . good ();
```

4.2.2 Extending Class Person with Buffer Operations

The buffer classes have the capability of packing any number and type of values, but they do not record how these values are combined to make objects. In order to pack and unpack a buffer for a `Person` object, for instance, we have to specify the order in which the members of `Person` are packed and unpacked. Section 4.1 and the code in Appendix D included operations for packing and unpacking the members of `Person` objects in insertion (<<) and extraction (>>) operators. In this section and Appendix E, we add those operations as methods of class `Person`. The

definition of the class has the following method for packing delimited text buffers. The unpack operation is equally simple:

```
int Person::Pack (DelimTextBuffer & Buffer) const
{// pack the fields into a DelimTextBuffer
   int result;
   result = Buffer . Pack (LastName);
   result = result && Buffer . Pack (FirstName);
   result = result && Buffer . Pack (Address);
   result = result && Buffer . Pack (City);
   result = result && Buffer . Pack (State);
   result = result && Buffer . Pack (ZipCode);
   return result;
}
```

4.2.3 Buffer Classes for Length-Based and Fixed-length Fields

Representing records of length-based fields and records of fixed-length fields requires a change in the implementations of the `Pack` and `Unpack` methods of the delimited field class, but the class definitions are almost exactly the same. The main members and methods of class `LengthTextBuffer` are given in Fig. 4.12. The full class definition and method implementation are given in `lentext.h` and `lentext.cpp`

```
lass LengthTextBuffer
  public:
  LengthTextBuffer (int maxBytes = 1000);
  int Read (istream & file);
  int Write (ostream & file) const;
  int Pack (const char * field, int size = -1);
  int Unpack (char * field);
rivate:
  char * Buffer; // character array to hold field values
  int BufferSize; // size of packed fields
  int MaxBytes; // maximum number of characters in the buffer
  int NextByte; // packing/unpacking position in buffer
```

gure 4.12 Main methods and members of class `LengthTextBuffer`.

in Appendix E. The only changes that are apparent from this figure are the name of the class and the elimination of the `delim` parameter on the constructor. The code for the `Pack` and `Unpack` methods is substantially different, but the `Read` and `Write` methods are exactly the same.

Class `FixedTextBuffer`, whose main members and methods are in Fig. 4.13 (full class in `fixtext.h` and `fixtext.cpp`), is different in two ways from the other two classes. First, it uses a fixed collection of fixed-length fields. Every buffer value has the same collection of fields, and the `Pack` method needs no size parameter. The second difference is that it uses fixed-length records. Hence, the `Read` and `Write` methods do not use a length indicator for buffer size. They simply use the fixed size of the buffer to determine how many bytes to read or write.

The method `AddField` is included to support the specification of the fields and their sizes. A buffer for objects of class `Person` is initialized by the new method `InitBuffer` of class `Person`:

```
int Person::InitBuffer (FixedTextBuffer & Buffer)
// initialize a FixedTextBuffer to be used for Person objects
{
   Buffer . Init (6, 61);//6 fields, 61 bytes total
   Buffer . AddField (10); // LastName [11];
   Buffer . AddField (10); // FirstName [11];
   Buffer . AddField (15); // Address [16];
```

```
class FixedTextBuffer
{  public:
   FixedTextBuffer (int maxBytes = 1000);
   int AddField (int fieldSize);
   int Read (istream & file);
   int Write (ostream & file) const;
   int Pack (const char * field);
   int Unpack (char * field);
private:
   char * Buffer; // character array to hold field values
   int BufferSize; // size of packed fields
   int MaxBytes; // maximum number of characters in the buffer
   int NextByte; // packing/unpacking position in buffer
   int * FieldSizes; // array of field sizes
};
```

Figure 4.13 Main methods and members of class `FixedTextBuffer`.

```
Buffer . AddField (15); // City [16];
Buffer . AddField (2);  // State [3];
Buffer . AddField (9); // ZipCode [10];
return 1;
```

4.3 Using Inheritance for Record Buffer Classes

A reading of the cpp files for the three classes above shows a striking similarity: a large percentage of the code is duplicated. In this section, we eliminate almost all of the duplication through the use of the inheritance capabilities of C++.

4.3.1 Inheritance in the C++ Stream Classes

C++ incorporates *inheritance* to allow multiple classes to share members and methods. One or more base classes define members and methods, which are then used by subclasses. The stream classes are defined in such a hierarchy. So far, our discussion has focused on class `fstream`, as though it stands alone. In fact, `fstream` is embedded in a class hierarchy that contains many other classes. The read operations, including the extraction operators are defined in class `istream`. The write operations are defined in class `ostream`. Class `fstream` inherits these operations from its parent class `iostream`, which in turn inherits from `istream` and `ostream`. The following definitions are included in `iostream.h` and `fstream.h`:

```
class istream: virtual public ios { . . .
class ostream: virtual public ios { . . .
class iostream: public istream, public ostream { . . .
class ifstream: public fstreambase, public istream { . . .
class ofstream: public fstreambase, public ostream { . . .
class fstream: public fstreambase, public iostream { . . .
```

We can see that this is a complex collection of classes. There are two base classes, `ios` and `fstreambase`, that provide common declarations and basic stream operations (`ios`) and access to operating system file operations (`fstreambase`). There are uses of *multiple inheritance* in these classes; that is, classes have more than one base class. The keyword

virtual is used to ensure that class `ios` is included only once in the ancestry of any of these classes.

Objects of a class are also objects of their base classes, and generally, include members and methods of the base classes. An object of class `fstream`, for example, is also an object of classes `fstreambase`, `iostream`, `istream`, `ostream`, and `ios` and includes all of the members and methods of those base classes. Hence, the read method and extraction (>>) operations defined in `istream` are also available in `iostream`, `ifstream`, and `fstream`. The open and close operations of class `fstreambase` are also members of class `fstream`.

An important benefit of inheritance is that operations that work on base class objects also work on derived class objects. We had an example of this benefit in the function `ReadVariablePerson` in Section 4.1.5 that used an `istrstream` object `strbuff` to contain a string buffer. The code of that function passed `strbuff` as an argument to the person extraction function that expected an `istream` argument. Since `istrstream` is derived from `istream`, `strbuff` *is* an `istream` object and hence can be manipulated by this `istream` operation.

4.3.2 A Class Hierarchy for Record Buffer Objects

The characteristics of the three buffer classes of Section 4.2 can be combined into a single class hierarchy, as shown in Fig. 4.14. Appendix F has the full implementation of these classes. The members and methods that are common to all of the three buffer classes are included in the base class `IOBuffer`. Other methods are in classes `VariableLengthBuffer` and `FixedLengthBuffer`, which support the read and write operations for different types of records. Finally the classes `LengthFieldBuffer`, `DelimFieldBuffer`, and `FixedFieldBuffer` have the pack and unpack methods for the specific field representations.

The main members and methods of class `IOBuffer` are given in Fig. 4.15. The full class definition is in file `iobuffer.h`, and the implementation of the methods is in file `iobuffer.cpp`. The common members of all of the buffer classes, `BufferSize`, `MaxBytes`, `NextByte`, and `Buffer`, are declared in class `IOBuffer`. These members are in the `protected` Section of `IOBuffer`.

This is our first use of protected access, which falls between private (no access outside the class) and public (no access restrictions). Protected members of a class can be used by methods of the class and by methods of

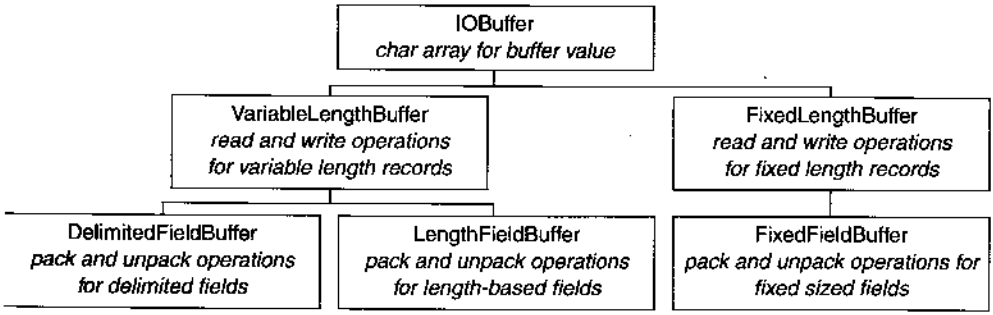

Figure 4.14 Buffer class hierarchy

classes derived from the class. The protected members of `IOBuffer` can be used by methods in all of the classes in this hierarchy. Protected members of `VariableLengthBuffer` can be used in its subclasses but not in classes `IOBuffer` and `FixedLengthBuffer`.

The constructor for class `IOBuffer` has a single parameter that specifies the maximum size of the buffer. Methods are declared for reading, writing, packing, and unpacking. Since the implementation of these methods depends on the exact nature of the record and its fields, `IOBuffer` must leave its implementation to the subclasses.

Class `IOBuffer` defines these methods as *virtual* to allow each subclass to define its own implementation. The = 0 declares a *pure virtual*

```
class IOBuffer
{public:
   IOBuffer (int maxBytes = 1000); // a maximum of maxBytes
   virtual int Read (istream &) = 0; // read a buffer
   virtual int Write (ostream &) const = 0; // write a buffer
   virtual int Pack (const void * field, int size = -1) = 0;
   virtual int Unpack (void * field, int maxbytes = -1) = 0;
 protected:
   char * Buffer; // character array to hold field values
   int BufferSize; // sum of the sizes of packed fields
   int MaxBytes; // maximum number of characters in the buffer
};
```

Figure 4.15 Main members and methods of class `IOBuffer`.

method. This means that the class `IOBuffer` does not include an implementation of the method. A class with pure virtual methods is an *abstract* class. No objects of such a class can be created, but pointers and references to objects of this class can be declared.

The full implementation of read, write, pack, and unpack operations for delimited text records is supported by two more classes. The reading and writing of variable-length records are included in the class `VariableLengthBuffer`, as given in Figure 4.16 and files `varlen.h` and `varlen.cpp`. Packing and unpacking delimited fields is in class `DelimitedFieldBuffer` and in files `delim.h` and `delim.cpp`. The code to implement these operations follows the same structure as in Section 4.2 but incorporates additional error checking. The `Write` method of `VariableLengthBuffer` is implemented as follows:

```
int VariableLengthBuffer :: Write (ostream & stream) const
// read the length and buffer from the stream

  int recaddr = stream . tellp ();
  unsigned short bufferSize = BufferSize;
  stream . write ((char *)&bufferSize, sizeof(bufferSize));
  if (!stream) return -1;
  stream . write (Buffer, BufferSize);
  if (!stream.good ()) return -1;
  return recaddr;
```

The method is implemented to test for all possible errors and to return information to the calling routine via the return value. We test for failure in the write operations using the expressions `!stream` and `!stream.good()`, which are equivalent. These are two different ways to test if the stream has experienced an error. The `Write` method returns the address in the stream where the record was written. The address is determined by calling `stream.tellg()` at the beginning of the function. `Tellg` is a method of `ostream` that returns the current location of the put pointer of the stream. If either of the write operations fails, the value -1 is returned.

An effective strategy for making objects persistent must make it easy for an application to move objects from memory to files and back correctly. One of the crucial aspects is ensuring that the fields are packed and unpacked in the same order. The class `Person` has been extended to include pack and unpack operations. The main purpose of these operations is to specify an ordering on the fields and to encapsulate error testing. The unpack operation is:

```
class VariableLengthBuffer: public IOBuffer
{
  public:
   VariableLengthBuffer (int MaxBytes = 1000);
   int Read (istream &);
   int Write (ostream &) const;
   int SizeOfBuffer () const; // return current size of buffer
};

class DelimFieldBuffer: public VariableLengthBuffer
{
  public:
   DelimFieldBuffer (char Delim = -1, int maxBytes = 1000;
   int Pack (const void*, int size = -1);
   int Unpack (void * field, int maxBytes = -1);
 protected:
   char Delim;
};
```

Figure 4.16 Classes `VariableLengthBuffer` and `DelimFieldBuffer`.

```
int Person::Unpack (IOBuffer & Buffer)
{
   Clear ();
   int numBytes;
   numBytes = Buffer . Unpack (LastName);
   if (numBytes == -1) return FALSE;
   LastName[numBytes] = 0;
   numBytes = Buffer . Unpack (FirstName);
   if (numBytes == -1) return FALSE;
   . . . // unpack the other fields
   return TRUE;
}
```

This method illustrates the power of virtual functions. The parameter of `Person::Unpack` is an object of type `IOBuffer`, but a call to `Unpack` supplies an argument that can be an object of any subclass of `IOBuffer`. The calls to `Buffer.Unpack` in the method `Person::Unpack` are virtual function calls. In calls of this type, the determination of exactly which `Unpack` method to call is not made during compilation as it is with nonvirtual calls. Instead, the actual type of the object `Buffer` is used to determine which function to call. In the following example of calling `Unpack`, the calls to `Buffer.Unpack` use the method `DelimFieldBuffer::Unpack`.

```
Person MaryAmes;
DelimFieldBuffer Buffer;
MaryAmes . Unpack (Buffer);
```

The full implementation of the I/O buffer classes includes class `LengthFieldBuffer`, which supports field packing with length plus value representation. This class is like `DelimFieldBuffer` in that it is implemented by specifying only the pack and unpack methods. The read and write operations are supported by its base class, `VariableLengthBuffer`.

4.4 Managing Fixed-Length, Fixed-Field Buffers

Class `FixedLengthBuffer` is the subclass of `IOBuffer` that supports read and write of fixed-length records. For this class, each record is of the same size. Instead of storing the record size explicitly in the file along with the record, the write method just writes the fixed-size record. The read method must know the size in order to read the record correctly. Each `FixedLengthBuffer` object has a protected field that records the record size.

Class `FixedFieldBuffer`, as shown in Fig. 4.17 and files `fixfld.h` and `fixfld.cpp`, supports a fixed set of fixed-length fields. One difficulty with this strategy is that the unpack method has to know the length of all of the fields. To make it convenient to keep track of the

```
class FixedFieldBuffer: public FixedLengthBuffer
public:
   FixedFieldBuffer (int maxFields, int RecordSize = 1000);
   FixedFieldBuffer (int maxFields, int * fieldSize);
   int AddField (int fieldSize); // define the next field
   int Pack (const void * field, int size = -1);
   int Unpack (void * field, int maxBytes = -1);
   int NumberOfFields () const; // return number of defined fields
protected:
   int * FieldSize; // array to hold field sizes
   int MaxFields; // maximum number of fields
   int NumFields; // actual number of defined fields
};
```

Figure 4.17 Class `FixedFieldBuffer`.

field lengths, class FixedFieldBuffer keeps track of the field sizes. The protected member FieldSize holds the field sizes in an integer array. The AddField method is used to specify field sizes. In the case of using a FixedFieldBuffer to hold objects of class Person, the InitBuffer method can be used to fully initialize the buffer:

```
int Person::InitBuffer (FixedFieldBuffer & Buffer)
   initialize a FixedFieldBuffer to be used for Persons

   int result;
   result = Buffer . AddField (10); // LastName [11];
   result = result && Buffer . AddField (10); // FirstName [11];
   result = result && Buffer . AddField (15); // Address [16];
   result = result && Buffer . AddField (15); // City [16];
   result = result && Buffer . AddField (2);  // State [3];
   result = result && Buffer . AddField (9); // ZipCode [10];
   return result;
```

Starting with a buffer with no fields, InitBuffer adds the fields one at a time, each with its own size. The following code prepares a buffer for use in reading and writing objects of class Person:

```
FixedFieldBuffer Buffer(6, 61); // 6 fields, 61 bytes total
MaryAmes.InitBuffer (Buffer);
```

Unpacking FixedFieldBuffer objects has to be done carefully. The object has to include information about the state of the unpacking. The member NextByte records the next character of the buffer to be unpacked, just as in all of the IOBuffer classes. FixedFieldBuffer has additional member NextField to record the next field to be unpacked. The method FixedFieldBuffer::Unpack is implemented as follows:

```
int FixedFieldBuffer :: Unpack (void * field, int maxBytes)

   if (NextField == NumFields || Packing)
     // buffer is full or not in unpacking mode
     return -1;
   int start = NextByte; // first byte to be unpacked
   int packSize = FieldSize[NextField]; // bytes to be unpacked
   memcpy (field, &Buffer[start], packSize); //move the bytes
   NextByte += packSize; // advance NextByte to following char
   NextField ++; // advance NextField
   if (NextField == NumFields) Clear (); // all fields unpacked
      return packSize;
```

4.5 An Object-Oriented Class for Record Files

Now that we know how to transfer objects to and from files, it is appropriate to encapsulate that knowledge in a class that supports all of our file operations. Class `BufferFile` (in files `buffile.h` and `buffile.cpp` of Appendix F) supports manipulation of files that are tied to specific buffer types. An object of class `BufferFile` is created from a specific buffer object and can be used to open and create files and to read and write records. Figure 4.18 has the main data methods and members of `BufferFile`.

Once a `BufferFile` object has been created and attached to an operating system file, each read or write is performed using the same buffer. Hence, each record is guaranteed to be of the same basic type. The following code sample shows how a file can be created and used with a `DelimFieldBuffer`:

```
DelimFieldBuffer buffer;
BufferFile file (buffer);
file . Open (myfile);
file . Read ();
buffer . Unpack (myobject);
```

```
class BufferFile
{public:
   BufferFile (IOBuffer &); // create with a buffer
   int Open (char * filename, int MODE); // open an existing file
   int Create (char * filename, int MODE); // create a new file
   int Close ();
   int Rewind (); // reset to the first data record
   // Input and Output operations
   int Read (int recaddr = -1);
   int Write (int recaddr = -1);
   int Append (); // write the current buffer at the end of file
protected:
   IOBuffer & Buffer; // reference to the file's buffer
   fstream File; // the C++ stream of the file
};
```

Figure 4.18 Main data members and methods of class `BufferFile`.

A buffer is created, and the `BufferFile` object `file` is attached to it. Then `Open` and `Read` methods are called for `file`. After the `Read`, `buffer` contains the packed record, and `buffer.Unpack` puts the record into `myobject`.

When `BufferFile` is combined with a fixed-length buffer, the result is a file that is guaranteed to have every record the same size. The full implementation of `BufferFile`, which is described in Section 5.2, "More about Record Structures," puts a `header` record on the beginning of each file. For fixed-length record files, the header includes the record size. `BufferFile::Open` reads the record size from the file header and compares it with the record size of the corresponding buffer. If the two are not the same, the `Open` fails and the file cannot be used.

This illustrates another important aspect of object-oriented design. Classes can be used to guarantee that operations on objects are performed correctly. It's easy to see that using the wrong buffer to read a file record is disastrous to an application. It is the encapsulation of classes like `BufferFile` that add safety to our file operations.

SUMMARY

The lowest level of organization that we normally impose on a file is a *stream of bytes*. Unfortunately, by storing data in a file merely as a stream of bytes, we lose the ability to distinguish among the fundamental informational units of our data. We call these fundamental pieces of information *fields*. Fields are grouped together to form *records*. Recognizing fields and records requires that we impose structure on the data in the file.

There are many ways to separate one field from the next and one record from the next:

- Fix the length of each field or record.
- Begin each field or record with a count of the number of bytes that it contains.
- Use delimiters to mark the divisions between entities.

In the case of fields, another useful technique is to use a "keyword = value" form to identify fields.

In this chapter we use the record structure with a length indicator at the beginning of each record to develop programs for writing and reading a simple file of variable-length records containing names and addresses of individuals. We use buffering to accumulate the data in an individual record before we know its length to write it to the file. Buffers are also

useful in allowing us to read in a complete record at one time. We represent the length field of each record as a binary number or as a sequence of ASCII digits. In the former case, it is useful to use a *file dump* to examine the contents of our file.

The field packing and unpacking operations, in their various forms, can be encapsulated into C++ classes. The three different field representation strategies—delimited, length-based, and fixed-length—are implemented in separate classes. Almost all of the members and methods of these classes are identical. The only differences are in the exact packing and unpacking and in the minor differences in read and write between the variable-length and fixed-length record structures.

A better strategy for representing these objects lies in the use of a class hierarchy. Inheritance allows related classes to share members. For example, the two field packing strategies of delimited and length based can share the same variable-length record read and write methods. Virtual methods make the class hierarchy work.

The class `BufferFile` encapsulates the file operations of open, create, close, read, write, and seek in a single object. Each `BufferFile` object is attached to a buffer. The read and write operations move data between file and buffer. The use of `BufferFile` adds a level of protection to our file operations. Once a disk file is connected to a `BufferFile` object, it can be manipulated only with the related buffer.

KEY TERMS

Byte count field. A field at the beginning of a variable-length record that gives the number of bytes used to store the record. The use of a byte count field allows a program to transmit (or skip over) a variable-length record without having to deal with the record's internal structure.

Delimiter. One or more characters used to separate fields and records in a file.

Field. The smallest logically meaningful unit of information in a file. A record in a file is usually made up of several fields.

Fixed-length record. A file organization in which all records have the same length. Records are padded with blanks, nulls, or other characters so they extend to the fixed length. Since all the records have the same length, it is possible to calculate the beginning position of any record, making *direct access* possible.

Inheritance. A strategy for allowing classes to share data members and methods. A derived class inherits the members of its base class and may add additional members or modify the members it inherits.

Record. A collection of related fields. For example, the name, address, and so on of an individual in a mailing-list file would make up one record.

Stream of bytes. Term describing the lowest-level view of a file. If we begin with the basic *stream-of-bytes* view of a file, we can then impose our own higher levels of order on the file, including field, record, and block structures.

Variable-length record. A file organization in which the records have no predetermined length. They are just as long as they need to be and therefore make better use of space than fixed-length records do. Unfortunately, we cannot calculate the byte offset of a variable-length record by knowing only its relative record number.

Virtual method. A member function that can have different versions for different derived classes. A virtual function call dynamically selects the appropriate version for an object.

FURTHER READINGS

Object-oriented design is quite well covered in many books and articles. They range from a basic introduction, as in Irvine (1996), to the presentation of examples of solving business problems with object-oriented methods in Yourdon and Argila (1996). Booch (1991) is a comprehensive study of the use of object-oriented design methods. The use of files to store information is included in many database books, including Elmasri and Navathe (1994) and Silberschatz, Korth, and Sudarshan (1997).

EXERCISES

1. Find situations for which each of the four field structures described in the text might be appropriate. Do the same for each of the record structures described.
2. Discuss the appropriateness of using the following characters to delimit fields or records: carriage return, linefeed, space, comma, period, colon, escape. Can you think of situations in which you might want to use different delimiters for different fields?

3. Suppose you want to change class `Person` and the programs in section 4.1 to include a phone number field. What changes need to be made?
4. Suppose you need to keep a file in which every record has both fixed- and variable-length fields. For example, suppose you want to create a file of employee records, using fixed-length fields for each employee's ID (primary key), sex, birth date, and department, and using variable-length fields for each name and address. What advantages might there be to using such a structure? Should we put the variable-length portion first or last? Either approach is possible; how can each be implemented?
5. One record structure not described in this chapter is called *labeled.* In a labeled record structure each field that is represented is preceded by a label describing its contents. For example, if the labels *LN, FN, AD, CT, ST,* and *ZP* are used to describe the six fixed-length fields for a name and address record, it might appear as follows:

```
LNAmes   FNMary   AD123 Maple   CTStillwaterSTOKZP74075
```

Under what conditions might this be a reasonable, even desirable, record structure?

6. Define the terms *stream of bytes, stream of fields,* and *stream of records.*
7. Investigate the implementation of virtual functions in an implementation of C++. What data structure is used to represent the binding of function calls to function bodies? What is the interaction between the implementation of virtual functions and the constructors for classes?
8. Report on the basic field and record structures available in Ada or Cobol.
9. Compare the use of ASCII characters to represent *everything* in a file with the use of binary and ASCII data mixed together.
10. If you list the contents of a file containing both binary and ASCII characters on your terminal screen, what results can you expect? What happens when you list a completely binary file on your screen? (*Warning:* If you actually try this, do so with a very small file. You could lock up or reconfigure your terminal or even log yourself off!)
11. The following is a hex dump of the first few bytes from a file which uses variable-length records, a two-byte length, and delimited text fields. How long is the first record? What are its contents?

```
00244475  6D707C46  7265647C  38323120
4B6C7567  657C4861  636B6572  7C50417C
36353533  357C2E2E  48657861  64656369
```

12. The `Write` methods of the `IOBuffer` classes let the user change records but not delete records. How must the file structure and access procedures be modified to allow for deletion if we do not care about reusing the space from deleted records? How do the file structures and procedures change if we do want to reuse the space? Programming Exercises 21–26 of Chapter 6 ask you to implement various types of deletion.
13. What happens when method `VariableLengthBuffer::Write` is used to replace (or update) a record in a file, and the previous record had a different size? Describe possible solutions to this problem. Programming Exercise 25 of Chapter 6 asks you to implement a correct `Update` method.

PROGRAMMING EXERCISES

14. Rewrite the insertion (<<) operator of file `writestr.cpp` so that it uses the following field representations:
    a. Method 1, fixed length fields.
    b. Method 2, fields with length indicators.
    c. Method 3, fields delimited by "|".
    d. Method 4, fields with keyword tags.
15. Rewrite the extraction (>>) operator of file `readstr.cpp` so that it uses the following field representations:
    a. Method 1, fixed length fields.
    b. Method 2, fields with length indicators.
    c. Method 4, fields with keyword tags.
16. Write a program `writevar.cpp` that produces a file of `Person` objects that is formatted to be input to `readvar.cpp`.
17. Design and implement a class `KeywordBuffer` that pack buffers with keyword tags.
18. Modify class `FixedLengthBuffer` to support multiple field types within a single buffer. Make sure that the buffer does not overflow. You will need to add methods `PackFixed`, `PackLength`, and `PackDelim` and the corresponding unpack methods. You will also need to modify class `Person` or to create a new class, whose `Pack` and `Unpack` operations take advantage of these new capabilities.
19. Repeat Programming Exercise 16 for class `VariableLengthBuffer`.

20. Redesign the `IOBuffer` classes to allow arbitrary field packing as in the previous two exercises but this time via virtual pack and unpack methods. One purpose of this exercise is to allow class `BufferFile` to support these new capabilities.
21. Implement direct read by RRN for buffer class `FixedLengthBuffer`. Add a new implementation for the virtual methods `DRead` and `DWrite` in class `FixedLengthBuffer`.

PROGRAMMING PROJECT

This is the third part of the programming project. We add methods to store objects as records in files and load objects from files, using the `IOBuffer` classes of this chapter.

22. Add `Pack` and `Unpack` methods to class `Student`. Use class BufferFile to create a file of student records. Test these methods using the types of buffers supported by the IOBuffer classes.
23. Add `Pack` and `Unpack` methods to class `CourseRegistration`. Use class BufferFile to create a file of course registrations. Test these methods using the types of buffers supported by the IOBuffer classes.

The next part of the programming project is in Chapter 6.