CHAPTER 5

Managing Files of Records

CHAPTER OBJECTIVES

- Extend the file structure concepts of Chapter 4:
  - Search keys and canonical forms,
  - Sequential search,
  - Direct access, and
  - File access and file organization.
- Examine other kinds of file structures in terms of
  - Abstract data models,
  - Metadata,
  - Object-oriented file access, and
  - Extensibility.
- Examine issues of portability and standardization.

CHAPTER OUTLINE

5.1 Record Access
5.1.1 Record Keys
5.1.2 A Sequential Search
5.1.3 Unix Tools for Sequential Processing
5.1.4 Direct Access
5.2 More about Record Structures
5.2.1 Choosing a Record Structure and Record Length
5.2.2 Header Records
5.2.3 Adding Headers to C++ Buffer Classes
5.3 Encapsulating Record I/O Operations in a Single Class
5.4 File Access and File Organization
5.5 Beyond Record Structures
5.5.1 Abstract Data Models for File Access
5.5.2 Headers and Self-Describing Files
5.5.3 Metadata
5.5.4 Color Raster Images
5.5.5 Mixing Object Types in One File
5.5.6 Representation-Independent File Access
5.5.7 Extensibility
5.6 Portability and Standardization
5.6.1 Factors Affecting Portability
5.6.2 Achieving Portability

5.1 Record Access

5.1.1 Record Keys

Since our new file structure so clearly focuses on a record as the quantity of information that is being read or written, it makes sense to think in terms of retrieving just one specific record rather than reading all the way through the file, displaying everything. When looking for an individual record, it is convenient to identify the record with a *key* based on the record's contents. For example, in our name and address file we might want to access the "Ames record" or the "Mason record" rather than thinking in terms of the "first record" or "second record." (Can you remember

which record comes first?) This notion of a *key* is another fundamental conceptual tool. We need to develop a more exact idea of what a key is.

When we are looking for a record containing the last name Ames, we want to recognize it even if the user enters the key in the form "AMES," "ames," or "Ames." To do this, we must define a standard form for keys, along with associated rules and procedures for converting keys into this standard form. A standard form of this kind is often called a *canonical form* for the key. One meaning of the word *canon* is rule, and the word *canonical* means conforming to the rule. A canonical form for a search key is the *single* representation for that key that conforms to the rule.

As a simple example, we could state that the canonical form for a key requires that the key consist solely of uppercase letters and have no extra blanks at the end. So, if someone enters "Ames," we would convert the key to the canonical form "AMES" before searching for it.

It is often desirable to have *distinct keys,* or keys that uniquely identify a single record. If there is not a one-to-one relationship between the key and a single record, the program has to provide additional mechanisms to allow the user to resolve the confusion that can result when more than one record fits a particular key. Suppose, for example, that we are looking for Mary Ames's address. If there are several records in the file for several different people named Mary Ames, how should the program respond? Certainly it should not just give the address of the first Mary Ames it finds. Should it give all the addresses at once? Should it provide a way of scrolling through the records?

The simplest solution is to *prevent* such confusion. The prevention takes place as new records are added to the file. When the user enters a new record, we form a unique canonical key for that record and then search the file for that key. This concern about uniqueness applies only to *primary keys.* A primary key is, by definition, the key that is used to identify a record uniquely.

It is also possible, as we see later, to search on *secondary keys.* An example of a secondary key might be the city field in our name and address file. If we wanted to find all the records in the file for people who live in towns named Stillwater, we would use some canonical form of "Stillwater" as a secondary key. Typically, secondary keys do not uniquely identify a record.

Although a person's name might at first seem to be a good choice for a primary key, a person's name runs a high risk of failing the test for uniqueness. A name is a perfectly fine secondary key and in fact is often an important secondary key in a retrieval system, but there is too great a likelihood that two names in the same file will be identical.

The reason a name is a risky choice for a primary key is that it contains a real data value. In general, *primary keys should be dataless.* Even when we think we are choosing a unique key, if it contains data, there is a danger that unforeseen identical values could occur. Sweet (1985) cites an example of a file system that used a person's social security number as a primary key for personnel records. It turned out that, in the particular population that was represented in the file, there was a large number of people who were not United States citizens, and in a different part of the organization, all of these people had been assigned the social security number 999-99-9999!

Another reason, other than uniqueness, that a primary key should be dataless is that a primary key should be *unchanging.* If information that corresponds to a certain record changes and that information is contained in a primary key, what do you do about the primary key? You probably cannot change the primary key, in most cases, because there are likely to be reports, memos, indexes, or other sources of information that refer to the record by its primary key. As soon as you change the key, those references become useless.

A good rule of thumb is to avoid putting data into primary keys. If we want to access records according to data content, we should assign this content to secondary keys. We give a more detailed look at record access by primary and secondary keys in Chapter 6. For the rest of this chapter, we suspend our concern about whether a key is primary or secondary and concentrate on finding things by key.

5.1.2 A Sequential Search

Now that you know about keys, you should be able to write a program that reads through the file, record by record, looking for a record with a particular key. Such *sequential searching* is just a simple extension of our `readvar` program—adding a comparison operation to the main loop to see if the key for the record matches the key we are seeking. We leave the program as an exercise.

Evaluating Performance of Sequential Search

In the chapters that follow, we find ways to search for records that are faster than the sequential search mechanism. We can use sequential searching as a kind of baseline against which to measure the improvements we make. It is important, therefore, to find some way of expressing the amount of time and work expended in a sequential search.

Developing a performance measure requires that we decide on a unit of work that usefully represents the constraints on the performance of the whole process. When we describe the performance of searches that take place in electronic memory, where comparison operations are more expensive than fetch operations to bring data in from memory, we usually use the *number of comparisons* required for the search as the measure of work. But, given that the cost of a comparison in memory is so small compared with the cost of a disk access, comparisons do not fairly represent the performance constraints for a search through a file on secondary storage. Instead, we count low-level `Read` calls. We assume that each `Read` call requires a seek and that any one `Read` call is as costly as any other. We know from the discussions of matters, such as system buffering in Chapter 3, that these assumptions are not strictly accurate. But in a multiuser environment where many processes are using the disk at once, they are close enough to correct to be useful.

Suppose we have a file with a thousand records, and we want to use a sequential search to find Al Smith's record. How many `Read` calls are required? If Al Smith's record is the first one in the file, the program has to read in only a single record. If it is the last record in the file, the program makes a thousand `Read` calls before concluding the search. For an average search, 500 calls are needed.

If we double the number of records in a file, we also double both the average and the maximum number of `Read` calls required. Using a sequential search to find Al Smith's record in a file of two thousand records requires, on the average, a thousand calls. In other words, the amount of work required for a sequential search is directly proportional to the number of records in the file.

In general, the work required to search sequentially for a record in a file with n records is proportional to n: it takes at most n comparisons; on average it takes approximately $n/2$ comparisons. A sequential search is said to be of the order $O(n)$ because the time it takes is proportional to n.[1]

Improving Sequential Search Performance with Record Blocking

It is interesting and useful to apply some of the information from Chapter 3 about disk performance to the problem of improving sequential search performance. We learned in Chapter 3 that the major cost associated with a disk access is the time required to perform a seek to the right location on

1. If you are not familiar with this "big-oh" notation, you should look it up. Knuth (1997) is a good source.

the disk. Once data transfer begins, it is relatively fast, although still much slower than a data transfer within memory. Consequently, the cost of seeking and reading a record, then seeking and reading another record, is greater than the cost of seeking just once then reading two successive records. (Once again, we are assuming a multiuser environment in which a seek is required for each separate `Read` call.) It follows that we should be able to improve the performance of sequential searching by reading in a *block* of several records all at once and then processing that block of records in memory.

We began the previous chapter with a stream of bytes. We grouped the bytes into fields, then grouped the fields into records. Now we are considering a yet higher level of organization—grouping records into blocks. This new level of grouping, however, differs from the others. Whereas fields and records are ways of maintaining the logical organization within the file, blocking is done strictly as a performance measure. As such, the block size is usually related more to the physical properties of the disk drive than to the content of the data. For instance, on sector-oriented disks, the block size is almost always some multiple of the sector size.

Suppose that we have a file of four thousand records and that the average length of a record is 512 bytes. If our operating system uses sector-sized buffers of 512 bytes, then an unblocked sequential search requires, on the average, 2,000 `Read` calls before it can retrieve a particular record. By blocking the records in groups of sixteen per block so each `Read` call brings in 8 kilobytes worth of records, the number of reads required for an average search comes down to 125. Each `Read` requires slightly more time, since more data is transferred from the disk, but this is a cost that is usually well worth paying for such a large reduction in the number of reads.

There are several things to note from this analysis and discussion of record blocking:

- Although blocking can result in substantial performance improvements, it does not change the order of the sequential search operation. The cost of searching is still $O(n)$, increasing in direct proportion to increases in the size of the file.
- Blocking clearly reflects the differences between memory access speed and the cost of accessing secondary storage.
- Blocking does not change the number of comparisons that must be done in memory, and it probably increases the amount of data transferred between disk and memory. (We always read a whole block, even if the record we are seeking is the first one in the block.)

- Blocking saves time because it decreases the amount of seeking. We find, again and again, that this differential between the cost of seeking and the cost of other operations, such as data transfer or memory access, is the force that drives file structure design.

When Sequential Searching Is Good

Much of the remainder of this text is devoted to identifying better ways to access individual records; sequential searching is just too expensive for most serious retrieval situations. This is unfortunate because sequential access has two major practical advantages over other types of access: it is extremely easy to program, and it requires the simplest of file structures.

Whether sequential searching is advisable depends largely on how the file is to be used, how fast the computer system is that is performing the search, and how the file is structured. There are many situations in which a sequential search is reasonable. Here are some examples:

- ASCII files in which you are searching for some pattern (see `grep` in the next section);
- Files with few records (for example, ten records);
- Files that hardly ever need to be searched (for example, tape files usually used for other kinds of processing); and
- Files in which you want all records with a certain secondary key value, where a large number of matches is expected.

Fortunately, these sorts of applications do occur often in day-to-day computing—so often, in fact, that operating systems provide many utilities for performing sequential processing. Unix is one of the best examples of this, as we see in the next section.

5.1.3 Unix Tools for Sequential Processing

Recognizing the importance of having a standard file structure that is simple and easy to program, the most common file structure that occurs in Unix is *an ASCII file with the new-line character as the record delimiter and, when possible, white space as the field delimiter.* Practically all files that we create with Unix editors use this structure. And since most of the built-in C and C++ functions that perform I/O write to this kind of file, it is common to see data files that consist of fields of numbers or words separated by blanks or tabs and records separated by new-line characters. Such files are simple and easy to process. We can, for instance, generate an ASCII file with a simple program and then use an editor to browse through it or alter it.

Unix provides a rich array of tools for working with files in this form. Since this kind of file structure is inherently sequential (records are variable in length, so we have to pass from record to record to find any particular field or record), many of these tools process files sequentially.

Suppose, for instance, that we choose the white-space/new-line structure for our address file, ending every field with a tab and ending every record with a new line. While this causes some problems in distinguishing fields (a blank is white space, but it doesn't separate a field) and in that sense is not an ideal structure, it buys us something very valuable: the full use of those Unix tools that are built around the white-space/new-line structure. For example, we can print the file on our console using any of a number of utilities, some of which follow.

cat

```
% cat myfile
Ames Mary 123 Maple Stillwater OK   74075
MasonAlan 90 Eastgate    Ada   OK   74820
```

Or we can use tools like `wc` and `grep` for processing the files.

wc

The command `wc` (word count) reads through an ASCII file sequentially and counts the number of lines (delimited by new lines), words (delimited by white space), and characters in a file:

```
% wc myfile
     2    14    76
```

grep

It is common to want to know if a text file has a certain word or character string in it. For ASCII files that can reasonably be searched sequentially, Unix provides an excellent filter for doing this called `grep` (and its variants `egrep` and `fgrep`). The word `grep` stands for *generalized regular expression*, which describes the type of pattern that `grep` is able to recognize. In its simplest form, `grep` searches sequentially through a file for a pattern. It then returns to standard output (the console) all the lines in the file that contain the pattern.

```
% grep Ada myfile
MasonAlan 90 Eastgate    Ada  OK    74820
```

We can also combine tools to create, on the fly, some very powerful file processing software. For example, to find the number of lines containing the word *Ada* and the number of words and bytes in those lines we use

```
% grep Ada myfile | wc
     1     7    36
```

As we move through the text, we will encounter a number of other powerful Unix commands that sequentially process files with the basic white-space/new-line structure.

5.1.4 Direct Access

The most radical alternative to searching sequentially through a file for a record is a retrieval mechanism known as *direct access*. We have direct access to a record when we can seek directly to the beginning of the record and read it in. Whereas sequential searching is an $O(n)$ operation, direct access is $O(1)$. No matter how large the file is, we can still get to the record we want with a single seek. Class `IOBuffer` includes direct read (`DRead`) and write (`DWrite`) operations using the byte address of the record as the record reference:

```
int IOBuffer::DRead (istream & stream, int recref)
// read specified record
{
  stream . seekg (recref, ios::beg);
  if (stream . tellg () != recref) return -1;
  return Read (stream);
}
```

The `DRead` function begins by seeking to the requested spot. If this does not work, the function fails. Typically this happens when the request is beyond the end-of-file. After the seek succeeds, the regular, sequential `Read` method of the buffer object is called. Because `Read` is virtual, the system selects the correct one.

Here we are able to write the direct read and write methods for the base class `IOBuffer`, even though that class does not have sequential read and write functions. In fact, even when we add new derived classes with their own different `Read` and `Write` methods, we still do not have to change `Dread`. Score another one for inheritance and object-oriented design!

The major problem with direct access is knowing where the beginning of the required record is. Sometimes this information about record location is carried in a separate index file. But, for the moment, we assume that

we do not have an index. We assume that we know the *relative record number* (RRN) of the record we want. RRN is an important concept that emerges from viewing a file as a collection of records rather than as a collection of bytes. If a file is a sequence of records, the RRN of a record gives its position relative to the beginning of the file. The first record in a file has RRN 0, the next has RRN 1, and so forth.[2]

In our name and address file, we might tie a record to its RRN by assigning membership numbers that are related to the order in which we enter the records in the file. The person with the first record might have a membership number of 1001, the second a number of 1002, and so on. Given a membership number, we can subtract 1001 to get the RRN of the record.

What can we do with this RRN? Not much, given the file structures we have been using so far, which consist of variable-length records. The RRN tells us the relative position of the record we want in the sequence of records, but we still have to read sequentially through the file, counting records as we go, to get to the record we want. An exercise at the end of this chapter explores a method of moving through the file called *skip sequential* processing, which can improve performance somewhat, but looking for a particular RRN is still an $O(n)$ process.

To support direct access by RRN, we need to work with records of fixed, known length. If the records are all the same length, we can use a record's RRN to calculate the *byte offset* of the start of the record relative to the start of the file. For instance, if we are interested in the record with an RRN of 546 and our file has a fixed-length record size of 128 bytes per record, we can calculate the byte offset as

$$\text{Byte offset} = 546 \times 128 = 69\,888$$

In general, given a fixed-length record file where the record size is r, the byte offset of a record with an RRN of n is

$$\text{Byte offset} = n \times r$$

Programming languages and operating systems differ regarding where this byte offset calculation is done and even whether byte offsets are used for addressing within files. In C++ (and the Unix and MS-DOS operating systems), where a file is treated as just a sequence of bytes, the application program does the calculation and uses the `seekg` and `seekp` methods to

2. In keeping with the conventions of C and C++, we assume that the RRN is a *zero-based* count. In some file systems, the count starts at 1 rather than 0.

jump to the byte that begins the record. All movement within a file is in terms of bytes. This is a very low-level view of files; the responsibility for translating an RRN into a byte offset belongs wholly to the application program and not at all to the programming language or operating system.

Class `FixedLengthBuffer` can be extended with its own methods `DRead` and `DWrite` that interpret the `recref` argument as RRN instead of byte address. The methods are defined as virtual in class `IOBuffer` to allow this. The code in Appendix F does not include this extension; it is left as an exercise.

The Cobol language and the operating environments in which Cobol is often used (OS/MVS, VMS) are examples of a much different, higher-level view of files. The notion of a sequence of bytes is simply not present when you are working with record-oriented files in this environment. Instead, files are viewed as collections of records that are accessed by keys. The operating system takes care of the translation between a key and a record's location. In the simplest case, the key isjust the record's RRN, but the determination of location within the file is still not the programmer's concern.

5.2 More about Record Structures

5.2.1 Choosing a Record Structure and Record Length

Once we decide to fix the length of our records so we can use the RRN to give us direct access to a record, we have to decide on a record length. Clearly, this decision is related to the size of the fields we want to store in the record. Sometimes the decision is easy. Suppose we are building a file of sales transactions that contain the following information about each transaction:

- A six-digit account number of the purchaser,
- Six digits for the date field,
- A five-character stock number for the item purchased,
- A three-digit field for quantity, and
- A ten-position field for total cost.

These are all fixed-length fields; the sum of the field lengths is 30 bytes. Normally we would stick with this record size, but if performance is so important that we need to squeeze every bit of speed out of our retrieval system, we might try to fit the record size to the block organization of our

disk. For instance, if we intend to store the records on a typical sectored disk (see Chapter 3) with a sector size of 512 bytes or some other power of 2, we might decide to pad the record out to 32 bytes so we can place an integral number of records in a sector. That way, records will never span sectors.

The choice of a record length is more complicated when the lengths of the fields can vary, as in our name and address file. If we choose a record length that is the sum of our estimates of the largest possible values for all the fields, we can be reasonably sure that we have enough space for everything, but we also waste a lot of space. If, on the other hand, we are conservative in our use of space and fix the lengths of fields at smaller values, we may have to leave information out of a field. Fortunately, we can avoid this problem to some degree by appropriate design of the field structure *within* a record.

In our earlier discussion of record structures, we saw that there are two general approaches we can take toward organizing fields within a fixed-length record. The first, illustrated in Fig. 5.1(a) and implemented in class `FixedFieldBuffer`, uses fixed-length fields inside the fixed-length record. This is the approach we took for the sales transaction file previously described. The second approach, illustrated in Fig. 5.1(b), uses the fixed-length record as a kind of standard-sized container for holding something that looks like a variable-length record.

The first approach has the virtue of simplicity: it is very easy to "break out" the fixed-length fields from within a fixed-length record. The second approach lets us take advantage of an averaging-out effect that usually occurs: the longest names are not likely to appear in the same record as the longest address field. By letting the field boundaries vary, we can make

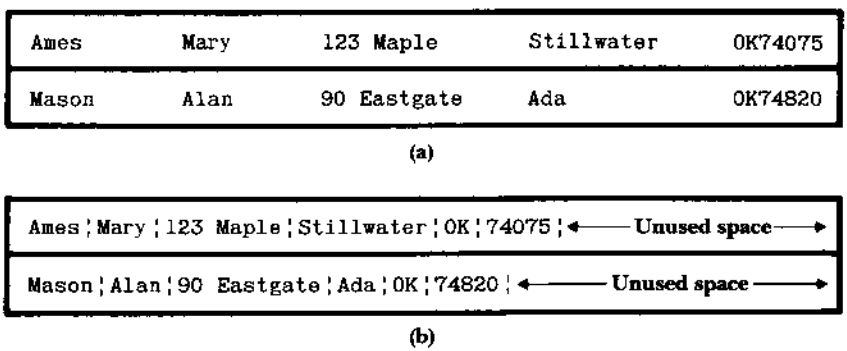


Figure 5.1 Two fundamental approaches to field structure within a fixed-length record. (a) Fixed-length records with fixed-length fields. (b) Fixed-length records with variable-length fields.

more efficient use of a fixed amount of space. Also, note that the two approaches are not mutually exclusive. Given a record that contains a number of truly fixed-length fields and some fields that have variable-length information, we might design a record structure that combines these two approaches.

One interesting question that must be resolved in the design of this kind of structure is that of distinguishing the real-data portion of the record from the unused-space portion. The range of possible solutions parallels that of the solutions for recognizing variable-length records in any other context: we can place a record-length count at the beginning of the record, we can use a special delimiter at the end of the record, we can count fields, and so on. As usual, there is no single right way to implement this file structure; instead we seek the solution that is most appropriate for our needs and situation.

Figure 5.2 shows the hex dump output from the two styles of representing variable-length fields in a fixed-length record. Each file has a *header record* that contains three 2-byte values: the size of the header, the number of records, and the size of each record. A full discussion of headers is deferred to the next section. For now, however, just look at the structure of the data records. We have italicized the length fields at the start of the records in the file dump. Although we filled out the records in Fig. 5.2b with blanks to make the output more readable, this blank fill is unnecessary. The length field at the start of the record guarantees that we do not read past the end of the data in the record.

5.2.2 Header Records

It is often necessary or useful to keep track of some general information about a file to assist in future use of the file. A *header record* is often placed at the beginning of the file to hold this kind of information. For example, in some languages there is no easy way to jump to the end of a file, even though the implementation supports direct access. One simple solution is to keep a count of the number of records in the file and to store that count somewhere. We might also find it useful to include information such as the length of the data records, the date and time of the file's most recent update, the name of the file, and so on. Header records can help make a file a self-describing object, freeing the software that accesses the file from having to know *a priori* everything about its structure, hence making the file-access software able to deal with more variation in file structures.

The header record usually has a different structure than the data records in the file. The file of Fig. 5.2a, for instance, uses a 32-byte header

```
0000000  0020  0002  0040  0000  0000  0000  0000  0000  ................  Header: header size (32),
0000020  0000  0000  0000  0000  0000  0000  0000  0000  ................  record count (2), record size (64)
0000040  416d  6573  7c4d  6172  797c  3132  3320  4d61  Ames|Mary|123 Ma  First record
0000060  706c  657c  5374  696c  6c77  6174  6572  7c4f  ple|Stillwater|O
0000100  4b7c  3734  3037  357c  0000  0000  0000  0000  K|74075|........
0000120  0000  0000  0000  0000  0000  0000  0000  0000  ................
0000140  4d61  736f  6e7c  416c  16e   7c39  3020  4561  Mason|Alan|90 Ea  Second record
0000160  7374  6761  7465  7c41  6461  7c4f  4b7c  3734  stgate|Ada|OK|74
0000200  3832  307c  0000  0000  0000  0000  0000  0000  820|............
0000220  0000  0000  0000  0000  0000  0000  0000  0000  ................
```

(a)

```
0000000  0042  0002  0044  0000  0000  0000  0000  0000  ................  Header: header size (66)
0000020  0000  0000  0000  0000  0000  0000  0000  0000  ................  record count (2), record size (68)
0000040  0000  0000  0000  0000  0000  0000  0000  0000  ................
0000060  0000  0000  0000  0000  0000  0000  0000  0000  ................
0000100  0000
0000102        0028  416d  6573  7c4d  6172  797c  3132    (.Ames|Mary|12  First record
0000120  3320  4d61  706c  657c  5374  696c  6c77  6174  3 Maple|Stillwat  Integer in first
0000140  6572  7c4f  4b7c  3734  3037  357c  0020  2020  er|OK|74075|      two bytes contains
0000160  2020  2020  2020  2020  2020  2020  2020  2020                    the number of
0000200  2020  2020                                                        bytes of data in the record
0000204              0024  4d61  736f  6e7c  416c  616e  $.Mason|Alan      Second record
0000220  7c39  3020  4561  7374  6761  7465  7c41  6461  |90 Eastgate|Ada
0000240  7c4f  4b7c  3734  3832  307c  0020  2020  2020  |OK|74820|
0000260  2020  2020  2020  2020  2020  2020  2020  2020
0000300  2020  2020  2020
```

(b)

Figure 5.2 Two different record structures that carry variable-length fields in a fixed-length record. (a) File containing a 32- (20_{16}) byte header and two fixed-length records (64 bytes each) containing variable-length fields that are terminated by a null character. (b) File containing a 66- (42_{16}) byte header and fixed-length records (68 bytes each) beginning with a fixed-length (2-byte) field that indicates the number of usable bytes in the record's variable-length fields.

record, whereas the data records each contain 64 bytes. Furthermore, the data records of this file contain only character data, whereas the header record contains integer fields that record the header record size, the number of data records, and the data record size.

Header records are a widely used, important file design tool. For example, when we reach the point at which we are discussing the construction of tree-structured indexes for files, we will see that header records are often placed at the beginning of the index to keep track of such matters as the RRN of the record that is the root of the index.

5.2.3 Adding Headers to C++ Buffer Classes

This section is an example of how to add header processing to the `IOBuffer` class hierarchy. It is not intended to show an optimal strategy for headers. However, these headers are used in all further examples in the book. The `Open` methods of new classes take advantage of this header strategy to verify that the file being opened is appropriate for its use. The important principle is that each file contains a header that incorporates information about the type of objects stored in the file.

The full definition of our buffer class hierarchy, as given in Appendix F, has been extended to include methods that support header records. Class `IOBuffer` includes the following methods:

```
virtual int ReadHeader ();
virtual int WriteHeader ();
```

Most of the classes in the hierarchy include their own versions of these methods. The write methods add a header to a file and return the number of bytes in the header. The read methods read the header and check for consistency. If the header at the beginning of a file is not the proper header for the buffer object, a `FALSE` value is returned; if it is the correct header, `TRUE` is returned.

To illustrate the use of headers, we look at fixed-length record files as defined in classes `IOBuffer` and `FixedLengthBuffer`. These classes were introduced in Chapter 4 and now include methods `ReadHeader` and `WriteHeader`. Appendix F contains the implementation of these methods of all of the buffer classes. The `WriteHeader` method for `IOBuffer` writes the string `IOBuffer` at the beginning of the file. The header for `FixedLengthBuffer` adds the string `Fixed` and the record size.

The `ReadHeader` method of `FixedLengthBuffer` reads the record size from the header and checks that its value is the same as that of the `BufferSize` member of the buffer object. That is, `ReadHeader`

verifies that the file was created using fixed-size records that are the right size for using the buffer object for reading and writing.

Another aspect of using headers in these classes is that the header can be used to initialize the buffer. At the end of `FixedLengthBuffer::ReadHeader` (see Appendix F), after the buffer has been found to be uninitialized, the record size of the buffer is set to the record size that was read from the header.

You will recall that in Section 4.5, "An Object-Oriented Class for Record Files," we introduced class `BufferFile` as a way to guarantee the proper interaction between buffers and files. Now that the buffer classes support headers, `BufferFile::Create` puts the correct header in every file, and `Buffer::Open` either checks for consistency or initializes the buffer, as appropriate. `BufferFile::ReadHeader` is called by `Open` and does all of its work in a single virtual function call. Appendix F has the details of the implementation of these methods.

`BufferFile::Rewind` repositions the get and put file pointers to the beginning of the first data record—that is, after the header record. This method is required because the `HeaderSize` member is protected. Without this method, it would be impossible to initiate a sequential read of the file.

5.3 Encapsulating Record I/O Operations in a Single Class

A good object-oriented design for making objects persistent should provide operations to read and write objects directly. So far, the write operation requires two separate operations: pack into a buffer and write the buffer to a file. In this section, we introduce class `RecordFile` which supports a read operation that takes an object of some class and writes it to a file. The use of buffers is hidden inside the class.

The major problem with defining class `RecordFile` is how to make it possible to support files for different object types without needing different versions of the class. Consider the following code that appears to read a `Person` from one file and a `Recording` (a class defined in Chapter 7) from another file:

```
Person p; RecordFile pFile; pFile . Read (p);
Recording r;    RecordFile rFile; rFile . Read (r);
```

Is it possible that class `RecordFile` can support read and unpack for a `Person` and a `Recording` without change? Certainly the objects are different—they have different unpacking methods. Virtual function calls

do not help because `Person` and `Recording` do not have a common base type. It is the C++ *template* feature that solves our problem by supporting parameterized function and class definitions. Figure 5.3 gives the definition of the template class `RecordFile`.

```
#include "buffile.h"
#include "iobuffer.h"
   template class to support direct read and write of records
   The template parameter RecType must support the following
       int Pack (BufferType &); pack record into buffer
       int Unpack (BufferType &); unpack record from buffer

template <class RecType>
class RecordFile: public BufferFile
{public:
       int Read (RecType & record, int recaddr = -1);
       int Write (const RecType & record, int recaddr = -1);
       RecordFile (IOBuffer & buffer): BufferFile (buffer) {}
};

// template method bodies
template <class RecType>
int RecordFile<RecType>::Read (RecType & record, int recaddr = -1)
{
       int writeAddr, result;
       writeAddr = BufferFile::Read (recaddr);
       if (!writeAddr) return -1;
       result = record . Unpack (Buffer);
       if (!result) return -1;
       return writeAddr;
}

template <class RecType>
int RecordFile<RecType>::Write (const RecType & record, int recad-
dr = -1)
{
       int result;
       result = record . Pack (Buffer);
       if (!result) return -1;
       return BufferFile::Write (recaddr);
}
```

Figure 5.3 Template class `RecordFile`.

The definition of class `RecordFile` is a template in the usual sense of the word: a pattern that is used as a guide to make something accurately. The definition does not define a specific class but rather shows how particular record file classes can be constructed. When a template class is supplied with values for its parameters, it becomes a real class. For instance, the following defines an object called `PersonFile`:

```
RecordFile<Person> PersonFile (Buffer);
```

The object `Personfile` is a `RecordFile` that operates on Person objects. All of the operations of `RecordFile<Person>` are available, including those from the parent class `BufferFile`. The following code includes legitimate uses of `PersonFile`:

```
Person person;
PersonFile.Create("person.dat", ios::in); // create a file
PersonFile.Read(person); // read a record into person
PersonFile.Append(person); // write person at end of file
PersonFile.Open("person.dat", ios::in); // open and check header
```

Template definitions in C++ support the reuse of code. We can write a single class and use it in multiple contexts. The same `RecordFile` class declared here and used for files of `Person` objects will be used in subsequent chapters for quite different objects. No changes need be made to `RecordFile` to support these different uses.

Program `testfile.cpp`, in Appendix F, uses `RecordFile` to test all of the buffer I/O classes. It also includes a template function, `TestBuffer`, which is used for all of the buffer tests.

5.4 File Access and File Organization

In the course of our discussions in this and the previous chapter, we have looked at

- Variable-length records,
- Fixed-length records,
- Sequential access, and
- Direct access.

The first two of these relate to aspects of *file organization*; the last two have to do with *file access*. The interaction between file organization and file access is a useful one; we need to look at it more closely before continuing.

Most of what we have considered so far falls into the category of file organization:

- Can the file be divided into fields?
- Is there a higher level of organization to the file that combines the fields into records?
- Do all the records have the same number of bytes or fields?
- How do we distinguish one record from another?
- How do we organize the internal structure of a fixed-length record so we can distinguish between data and extra space?

We have seen that there are many possible answers to these questions and that the choice of a particular file organization depends on many things, including the file-handling facilities of the language you are using and the *use you want to make of the file.*

Using a file implies access. We looked first at sequential access, ultimately developing a *sequential search.* As long as we did not know where individual records began, sequential access was the only option open to us. When we wanted *direct access,* we fixed the length of our records, and this allowed us to calculate precisely where each record began and to seek directly to it.

In other words, our desire for direct *access* caused us to choose a fixed-length record file *organization.* Does this mean that we can equate fixed-length records with direct access? Definitely not. There is nothing about our having fixed the length of the records in a file that precludes sequential access; we certainly could write a program that reads sequentially through a fixed-length record file.

Not only can we elect to read through the fixed-length records sequentially but we can also provide direct access to *variable-length* records simply by keeping a list of the byte offsets from the start of the file for the placement of each record. We chose a fixed-length record structure for the files of Fig. 5.2 because it is simple and adequate for the data we wanted to store. Although the lengths of our names and addresses vary, the variation is not so great that we cannot accommodate it in a fixed-length record.

Consider, however, the effects of using a fixed-length record organization to provide direct access to documents ranging in length from a few hundred bytes to more than a hundred kilobytes. Using fixed-length

records to store these documents would be disastrously wasteful of space, so some form of variable-length record structure would have to be found. Developing file structures to handle such situations requires that you clearly distinguish between the matter of *access* and your options regarding *organization.*

The restrictions imposed by the language and file system used to develop your applications impose limits on your ability to take advantage of this distinction between access method and organization. For example, the C++ language provides the programmer with the ability to implement direct access to variable-length records, since it allows access to any byte in the file. On the other hand, Pascal, even when seeking is supported, imposes limitations related to the language's definition of a file as a collection of elements that are all of the same *type* and, consequently, size. Since the elements must all be of the same size, direct access to variable-length records is difficult, at best, in Pascal.

5.5 Beyond Record Structures

Now that we have a grip on the concepts of organization and access, we look at some interesting new file organizations and more complex ways of accessing files. We want to extend the notion of a file beyond the simple idea of records and fields.

We begin with the idea of abstract data models. Our purpose here is to put some distance between the physical and logical organization of files to allow us to focus more on the information content of files and less on physical format.

5.5.1 Abstract Data Models for File Access

The history of file structures and file processing parallels that of computer hardware and software. When file processing first became common on computers, magnetic tape and punched cards were the primary means used to store files. Memory space was dear, and programming languages were primitive. Programmers as well as users were compelled to view file data exactly as it might appear on a tape or cards—as a sequence of fields and records. Even after the data was loaded into memory, the tools for manipulating and viewing the data were unsophisticated and reflected the magnetic tape metaphor. Data processing meant processing fields and records in the traditional sense.

Gradually, computer users began to recognize that computers could process more than just fields and records. Computers could, for instance, process and transmit sound, and they could process and display images and documents (Fig. 5.4). These kinds of applications deal with information that does not fit the metaphor of data stored as sequences of records that are divided into fields, even if, ultimately, the data might be stored physically in the form of fields and records. It is easier, in the mind's eye, to envision data objects such as documents, images, and sound as objects we manipulate in ways that are specific to the objects, rather than simply as fields and records on a disk.

The notion that we need not view data only as it appears on a particular medium is captured in the phrase *abstract data model*, a term that encourages an application-oriented view of data rather than a medium-oriented one. The organization and access methods of abstract data models are described in terms of how an application views the data rather than how the data might physically be stored.

One way we save a user from having to know about objects in a file is to keep information in the file that file-access software can use to "understand" those objects. A good example of how this might be done is to put file structure information in a header.

5.5.2 Headers and Self-Describing Files

We have seen how a header record can be used to keep track of how many records there are in a file. If our programming language permits it, we can put much more elaborate information about a file's structure in the header. When a file's header contains this sort of information, we say the file is *self-describing*. Suppose, for instance, that we store in a file the following information:

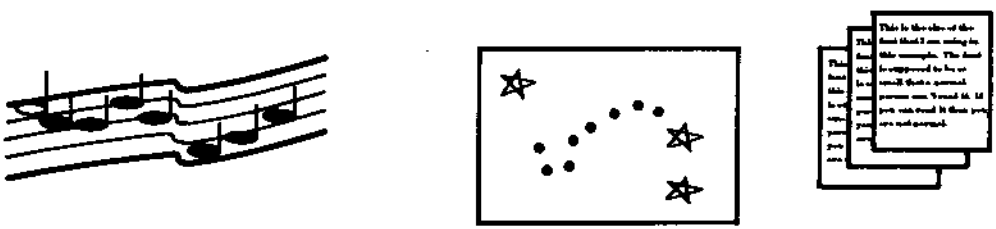


Figure 5.4 Data such as sound, images, and documents do not fit the traditional metaphor of data stored as sequences of records that are divided into fields.

- A name for each field,
- The width of each field, and
- The number of fields per record.

We can now write a program that can read and print a meaningful display of files with any number of fields per record and any variety of fixed-length field widths. In general, the more file structure information we put into a file's header, the less our software needs to know about the specific structure of an individual file.

As usual, there is a trade-off: if we do not hard-code the field and record structures of files in the programs that read and write them, the programs must be more sophisticated. They must be flexible enough to interpret the self-descriptions they find in the file headers.

Consider the class `FixedFieldBuffer`, which keeps track of the sizes of all fields. We can extend the header to be more self-describing by including the number of fields and their sizes. The final piece of the header is created by the `FixedFieldBuffer::WriteHeader` method. For this header, we want to record the number of fields and the size of each field. This information is stored in the members `NumFields` and `FieldSize`. This requires a variable-sized header, since the number of fields, hence the number of sizes in the header, are different for different record types. We choose to store this information in the file header by writing it directly into the file as a sequence of fixed-size binary fields. This strategy is very compact and is easy to implement. Now `FixedFieldBuffer::ReadHeader` can check for full consistency of file and buffer and can also fully initialize a buffer when opening a file.

The resulting file with its header for our two `Person` objects is given in Fig. 5.5. The value after "Fixed" in italics (*00 0000 3d*) is the record size, 61. The value after "Field" in italics (*0000 0006*) is the number of fields. The field sizes follow, 4 bytes each.

One advantage of putting this header in the file is that the `FixedFieldBuffer` object can be initialized from the header. The `ReadHeader` method of `FixedFieldBuffer`, after reading the header, checks whether the buffer object has been initialized. If not, the information from the header is used to initialize the object. The body of `ReadHeader` is given in Appendix F.

5.5.3 Metadata

Suppose you are an astronomer interested in studying images generated by telescopes that scan the sky, and you want to design a file structure for the

```
0000000   I   O   B   u   f   f   e   r   F   i   x   e   d  \0  \0  \0
            494f    4275    6666    6572    4669    7865    6400    0000
0000020   =   F   i   e   l   d  \0  \0  \0 006  \0  \0  \0  \n  \0  \0
            3d46    6965    6c64    0000    0006    0000    000a    0000
0000040  \0  \n  \0  \0  \0 017  \0  \0  \0 017  \0  \0  \0 002  \0  \0
            000a    0000    000f    0000    000f    0000    0002    0000
0000060  \0  \t   A   m   e   s  \0  \0  \0  \0  \0  \0   M   a   r   y
            0009    416d    6573    0000    0000    0000    4d61    7279
0000100  \0  \0  \0  \0  \0  \0   1   2   3       M   a   p   l   e  \0
            0000    0000    0000    3132    3320    4d61    706c    6500
0000120  \0  \0  \0  \0  \0   S   t   i   l   l   w   a   t   e   r  \0
            0000    0000    0053    7469    6c6c    7761    7465    7200
0000140  \0  \0  \0  \0   O   K   7   4   0   7   5  \0  \0  \0  \0   M
            0000    0000    4f4b    3734    3037    3500    0000    004d
0000160   a   s   o   n  \0  \0  \0  \0  \0   A   l   a   n  \0  \0  \0
            6173    6f6e    0000    0000    0041    6c61    6e00    0000
0000200  \0  \0  \0   9   0       E   a   s   t   g   a   t   e  \0  \0
            0000    0039    3020    4561    7374    6761    7465    0000
0000220  \0  \0   A   d   a  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
            0000    4164    6100    0000    0000    0000    0000    0000
0000240  \0   O   K   7   4   8   2   0  \0  \0  \0  \0
            004f    4b37    3438    3230    0000    0000
```

Figure 5.5 File dump of a fixed-field file with descriptive header.

digital representations of these images (Fig. 5.6). You expect to have many images, perhaps thousands, that you want to study, and you want to store one image per file. While you are primarily interested in studying the images, you will certainly need information *about* each image: where in the sky the image is from, when it was made, what telescope was used, what other images are related, and so forth.

This kind of information is called *metadata*—data that describes the primary data in a file. Metadata can be incorporated into any file whose primary data requires supporting information. If a file is going to be shared by many users, some of whom might not otherwise have easy access to its metadata, it may be most convenient to store the metadata in the file. A common place to store metadata in a file is the header record.

Typically, a community of users of a particular kind of data agrees on a standard format for holding metadata. For example, a standard format called FITS (Flexible Image Transport System) has been developed by the International Astronomers' Union for storing the kind of astronomical

Figure 5.6 To make sense of this 2-megabyte image, an astronomer needs such metadata as the kind of image it is, the part of the sky it is from, and the telescope that was used to view it. Astronomical metadata is often stored in the same file as the data itself. (This image shows polarized radio emission from the southern spiral galaxy NGC 5236 [M83] as observed with the Very Large Array radio telescope in New Mexico.)

data just described in a file's header.[3] A FITS header is a collection of 2880-byte blocks of 80-byte ASCII records, in which each record contains a single piece of metadata. Figure 5.7 shows part of a FITS header. In a FITS file, the header is followed by the numbers that describe the image, one binary number per observed point of the image.

Note that the designers of the FITS format chose to use ASCII in the header but binary values for the image. ASCII headers are easy to read and process and, since they occur only once, take up relatively little space. Because the numbers that make a FITS image are rarely read by humans but are first processed into a picture and then displayed, binary format is the preferred choice for them.

A FITS image is a good example of an abstract data model. The data is meaningless without the interpretive information contained in the header, and FITS-specific methods must be employed to convert FITS data into an understandable image. Another example is the raster image, which we will look at next.

5.5.4 Color Raster Images

From a user's point of view, a modern computer is as much a graphical device as it is a data processor. Whether we are working with documents,

3. For more details on FITS, see the references listed at the end of this chapter in "Further Readings."

```
SIMPLE  =                    T /CONFORMS TO BASIC FORMAT
BITPIX  =                   16 / BITS PER PIXEL
NAXIS   =                    2 / NUMBER OF AXES
NAXIS1  =                  256 / RA AXIS DIMENSION
NAXIS2  =                  256 / DEC AXIS DIMENSION
EXTEND  =                    F / T MEANS STANDARD EXTENSIONS EXIST
BSCALE  =          0.000100000 / TRUE = [TAPE*BSCALE]<pl>BZERO
BZERO   =          0.000000000 / OFFSET TO TRUE PIXEL VALUES
MAP_TYPE= 'REL EXPOSURE'       / INTENSITY OR RELATIVE EXPOSURE MAP
BUNIT   = '                '   / DIMENSIONLESS PEAK EXPOSURE FRACTION
CRVAL1  =              0.625 / RA    REF POINT VALUE (DEGREES)
CRPIX1  =            128.500 / RA    REF POINT PIXEL LOCATION
CDELT1  =       -0.006666700 / RA    INCREMENT ALONG AXIS (DEGREES)
CTYPE1  = 'RA---TAN'           / RA    TYPE
CROTA1  =              0.000 / RA    ROTATION
CRVAL2  =             71.967 / DEC   REF POINT VALUE (DEGREES)
CRPIX2  =            128.500 / DEC   REF POINT PIXEL LOCATION
CDELT2  =        0.006666700 / DEC   INCREMENT ALONG AXIS (DEGREES)
CTYPE2  = 'DEC--TAN'           / DEC   TYPE
CROTA2  =              0.000 / DEC   ROTATION
EPOCH   =             1950.0 / EPOCH OF COORDINATE SYSTEM
ARR TYPE=                  4 / 1=DP, 3=FP, 4=I
DATAMAX =              1.000 / PEAK INTENSITY (TRUE)
DATAMIN =              0.000 / MINIMUM INTENSITY (TRUE)
ROLL ANG=            -22.450 / ROLL ANGLE (DEGREES)
BAD ASP =                  0 / 0=good, 1=bad(Do not use roll angle)
TIME LIV=             5649.6 / LIVE TIME (SECONDS)
OBJECT  = 'REM6791           ' / SEQUENCE NUMBER
AVGOFFY =              1.899 / AVG Y OFFSET IN PIXELS, 8 ARCSEC/PIXEL
AVGOFFZ =              2.578 / AVG Z OFFSET IN PIXELS, 8 ARCSEC/PIXEL
RMSOFFY =              0.083 / ASPECT SOLN RMS Y PIXELS, 8 ARCSC/PIX
RMSOFFZ =              0.204 / ASPECT SOLN RMS Z PIXELS, 8 ARCSC/PIX
TELESCOP= 'EINSTEIN          ' / TELESCOPE
INSTRUME= 'IPC               ' / FOCAL PLANE DETECTOR
OBSERVER= '2                 ' / OBSERVER #: 0=CFA; 1=CAL; 2=MIT; 3=GSFC
GALL    =            119.370 / GALACTIC LONGITUDE OF FIELD CENTER
GALB    =              9.690 / GALACTIC LATITUDE OF FIELD CENTER
DATE OBS= '80/238            ' / YEAR & DAY NUMBER FOR OBSERVATION START
DATE STP= '80/238            ' / YEAR & DAY NUMBER FOR OBSERVATION STOP
TITLE   = 'SNR SURVEY: CTA1                                              '
ORIGIN  = 'HARVARD-SMITHSONIAN CENTER FOR ASTROPHYSICS                   '
DATE    = '22/09/1989        ' / DATE FILE WRITTEN
TIME    = '05:26:53          ' / TIME FILE WRITTEN
END
```

Figure 5.7 Sample FITS header. On each line, the data to the left of the / is the actual metadata (data about the raw data that follows in the file). For example, the second line (BITPIX = 16) indicates that the raw data in the file will be stored in 16-bit integer format. Everything to the right of a / is a comment, describing for the reader the meaning of the metadata that precedes it. Even a person uninformed about the FITS format can learn a great deal about this file just by reading through the header.

spreadsheets, or numbers, we are likely to be viewing and storing pictures in addition to whatever other information we work with. Let's examine one type of image, the color raster image, as a means to filling in our conceptual understanding of data objects.

A color raster image is a rectangular array of colored dots, or *pixels*,[4] that are displayed on a screen. A FITS image is a raster image in the sense that the numbers that make up a FITS image can be converted to colors, and then displayed on a screen. There are many different kinds of metadata that can go with a raster image, including

- The dimensions of the image—the number or pixels per row and the number of rows.
- The number of bits used to describe each pixel. This determines how many colors can be associated with each pixel. A 1-bit image can display only two colors, usually black and white. A 2-bit image can display four colors (2^2), an 8-bit image can display 256 colors (2^8), and so forth.
- A *color lookup table,* or *palette,* indicating which color is to be assigned to each pixel value in the image. A 2-bit image uses a color lookup table with 4 colors, an 8-bit image uses a table with 256 colors, and so forth.

If we think of an image as an abstract data type, what are some methods that we might associate with images? There are the usual ones associated with getting things in and out of a computer: a *read image* routine and a *store image* routine. Then there are those that deal with images as special objects:

- Display an image in a window on a console screen,
- Associate an image with a particular color lookup table,
- Overlay one image onto another to produce a composite image, and
- Display several images in succession, producing an animation.

The color raster image is an example of a type of data object that requires more than the traditional field/record file structure. This is particularly true when more than one image might be stored in a single file or when we want to store a document or other complex object together with images in a file. Let's look at some ways to mix object types in one file.

4. *Pixel* stands for "picture element."

5.5.5 Mixing Object Types in One File

Keywords

The FITS header (Fig. 5.7) illustrates an important technique, described earlier, for identifying fields and records: the use of *keywords*. In the case of FITS headers, we do not know which fields are going to be contained in any given header, so we identify each field using a *keyword = value* format.

Why does this format work for FITS files, whereas it was inappropriate for our address file? For the address file we saw that the use of keywords demanded a high price in terms of space, possibly even doubling the size of the file. In FITS files the amount of overhead introduced by keywords is quite small. When the image is included, the FITS file in the example contains approximately 2 megabytes. The keywords in the header occupy a total of about 400 bytes, or about 0.02 percent of the total file space.

Tags

With the addition via keywords of file structure information and metadata to a header, we see that a file can be more than just a collection of repeated fields and records. Can we extend this notion beyond the header to other, more elaborate objects? For example, suppose an astronomer would like to store *several* FITS images of different sizes in a file, together with the usual metadata, plus perhaps lab notes describing what the scientist learned from the image (Fig. 5.8). Now we can think of our file as a mixture of objects that may be very different in content—a view that our previous file structures do not handle well. Maybe we need a new kind of file structure.

There are many ways to address this new file design problem. One would be simply to put each type of object into a variable-length record and write our file processing programs so they know what each record looks like: the first record is a header for the first image, the second record

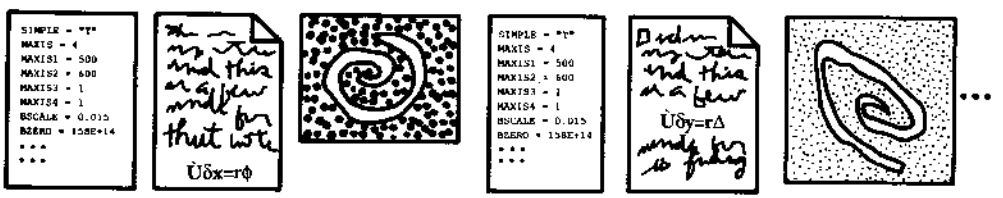


Figure 5.8 Information that an astronomer wants to include in a file.

is the image, the third record is a document, the fourth is a header for the second image, and so forth. This solution is workable and simple, but it has some familiar drawbacks:

- Objects must be accessed sequentially, making access to individual images in large files time-consuming.
- The file must contain exactly the objects that are described, in exactly the order indicated. We could not, for instance, leave out the notebook for some of the images (or in some cases leave out the notebook altogether) without rewriting all programs that access the file to reflect the changes in the file's structure.

A solution to these problems is hinted at in the FITS header: each line begins with a keyword that identifies the metadata field that follows in the line. Why not use keywords to identify *all* objects in the file—not just the fields in the headers but the headers themselves as well as the images and any other objects we might need to store? Unfortunately, the "keyword = data" format makes sense in a FITS header—it is short and fits easily in an 80-byte line—but it doesn't work at all for objects that vary enormously in size and content. Fortunately, we can generalize the keyword idea to address these problems by making two changes:

- Lift the restriction that each record be 80 bytes, and let it be big enough to hold the object that is referenced by the keyword.
- Place the keywords in an index table, together with the byte offset of the actual metadata (or data) and a length indicator that indicates how many bytes the metadata (or data) occupies in the file.

The term *tag* is commonly used in place of *keyword* in connection with this type of file structure. The resulting structure is illustrated in Fig. 5.9. In it we encounter two important conceptual tools for file design: (1) the use of an *index table* to hold descriptive information about the primary data, and (2) the use of *tags* to distinguish different types of objects. These tools allow us to store in one file a mixture of objects—objects that can vary from one another in structure and content.

Tag structures are common among standard file formats in use today. For example, a structure called TIFF (Tagged Image File Format) is a very popular tagged file format used for storing images. HDF (Hierarchical Data Format) is a standard tagged structure used for storing many different kinds of scientific data, including images. In the world of document storage and retrieval, SGML (Standard General Markup Language) is a *language* for describing document structures and for defining tags used to mark up that structure. Like FITS, each of these provides an interesting

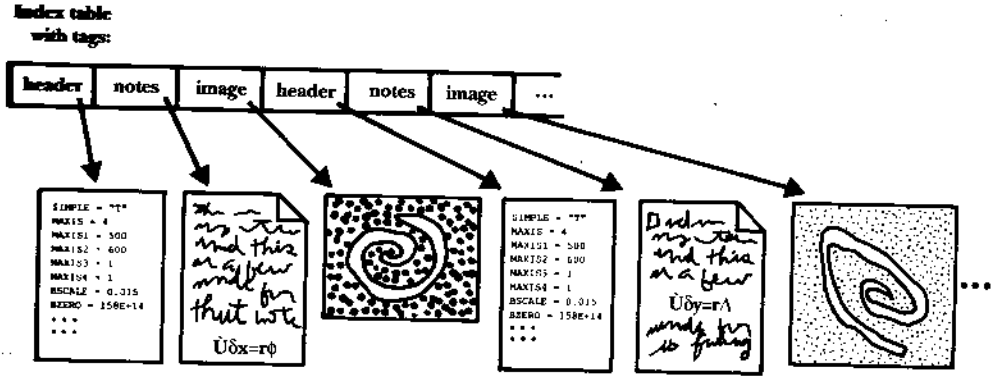


Figure 5.9 Same as Fig. 5.8, except with tags identifying the objects.

study in file design and standardization. References to further information on each are provided at the end of this chapter, in "Further Readings."

Accessing Files with Mixtures of Data Objects

The idea of allowing files to contain widely varying objects is compelling, especially for applications that require large amounts of metadata or unpredictable mixes of different kinds of data, for it frees us of the requirement that all records be fundamentally the same. As usual, we must ask what this freedom costs us. To gain some insight into the costs, imagine that you want to write a program to access objects in such a file. You now have to read and write tags as well as data, and the structure and format for different data types are likely to be different. Here are some questions you will have to answer almost immediately:

- When we want to read an object of a particular type, how do we search for the object?
- When we want to store an object in the file, how and where do we store its tag, and where exactly do we put the object?
- Given that different objects will have very different appearances within a file, how do we determine the correct method for storing or retrieving the object?

The first two questions have to do with accessing the table that contains the tags and pointers to the objects. Solutions to this problem are dealt with in detail in Chapter 6, so we defer their discussion until then. The third question, how to determine the correct methods for accessing objects, has implications that we briefly touch on here.

5.5.6 Representation-Independent File Access

We have used the term *abstract data model* to describe the view that an application has of a data object. This is essentially an in-memory, application-oriented view of an object, one that ignores the physical format of objects as they are stored in files. Taking this view of objects buys our software two things:

- It delegates to separate modules the responsibility of translating to and from the physical format of the object, letting the application modules concentrate on the task at hand. (For example, an image processing program that can operate in memory on 8-bit images should not have to worry about the fact that a particular image comes from a file that uses the 32-bit FITS format.)
- It opens up the possibility of working with objects that at some level fit the same abstract data model, even though they are stored in different formats. The in-memory representations of the images could be identical, even though they come from files with quite different formats.)

As an example that illustrates both points, suppose you have an image processing application program (we'll call it `find_star`) that operates in memory on 8-bit images, and you need to process a collection of images. Some are stored in FITS files in a FITS format, and some in TIFF files in a different format. A representation-independent approach (Fig. 5.10) would provide the application program with a routine (let's call it `read_image`) for reading images into memory in the expected 8-bit form, letting the application concentrate on the image processing task. For its part, the routine `read_image`, given a file to get an image from, determines the format of the image within the file, invokes the proper procedure to read the image in that format, and converts it from that format into the 8-bit memory format that the application needs.

Tagged file formats are one way to implement this conceptual view of file organization and file access. The specification of a tag can be accompanied by a specification of methods for reading, writing, and otherwise manipulating the corresponding data object according to the needs of an application. Indeed, any specification that separates the definition of the abstract data model from that of the corresponding file format lends itself to the representation-independent approach.

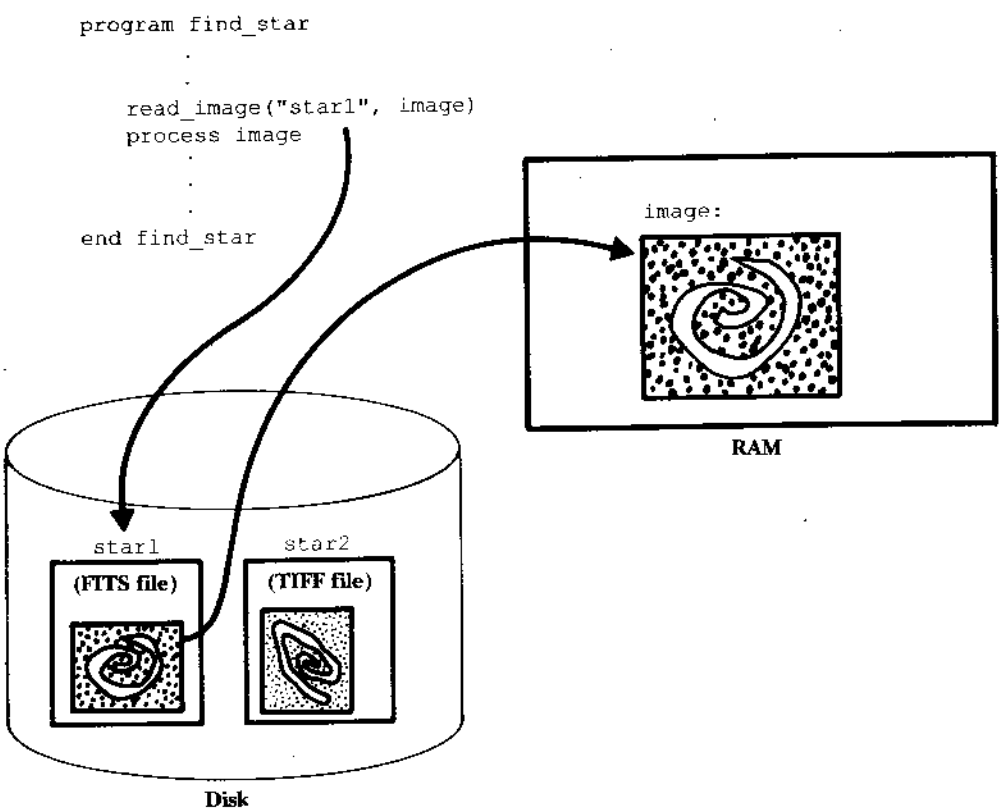


Figure 5.10 Example of object-oriented access. The program `find_star` knows nothing about the file format of the image that it wants to read. The routine `read_image` has methods to convert the image from whatever format it is stored in on disk into the 8-bit in-memory format required by `find_star`.

5.5.7 Extensibility

One of the advantages of using tags to identify objects within files is that we do not have to know *a priori* what all of the objects that our software may eventually have to deal with will look like. We have just seen that if our program is to be able to access a mixture of objects in a file, it must have methods for reading and writing each object. Once we build into our software a mechanism for choosing the appropriate methods for a given type of object, it is easy to imagine extending, at some future time, the types of objects that our software can support. Every time we encounter a new type of object that we would like to accommodate in our files, we can imple-

ment methods for reading and writing that object and add those methods to the repertoire of methods available to our file processing software.

5.6 Portability and Standardization

A recurring theme in several of the examples we have just seen is that people often want to share files. Sharing files means making sure that they are accessible on all of the different computers that they might turn up on and that they are somehow compatible with all of the different programs that will access them. In this final section, we look at two complementary topics that affect the sharability of files: portability and standardization.

5.6.1 Factors Affecting Portability

Imagine that you work for a company that wishes to share simple data files such as our address file with some other business. You get together with the other business to agree on a common field and record format, and you discover that your business does all of its programming and computing in C on a Sun computer and the other business uses Turbo Pascal on a PC. What sorts of issues would you expect to arise?

Differences among Operating Systems

In Chapter 2 in the section "Special Characters in Files," we saw that MS-DOS adds an extra line-feed character every time it encounters a carriage return character, whereas on most other file systems this is not the case. This means that every time our address file has a byte with hex value 0x0d, even if that byte is not meant to be a carriage return, the file is not extended by an extra 0x0a byte.

This example illustrates the fact that *the ultimate physical format of the same logical file can vary depending on differences among operating systems.*

Differences among Languages

Earlier in this chapter, when discussing header records, we chose to make header records and data records different sizes, but a Pascal programmer must use the same size for every record in the file. C++ allows us to mix and match fixed record lengths according to our needs, but Pascal requires that all records in a nontext file be the same size.

This illustrates a second factor impeding portability among files: *the physical layout of files produced with different languages may be constrained by the way the languages let you define structures within a file.*

Differences in Machine Architectures

Consider the hex dump in Fig. 5.2 which shows a file generated by a C program running on a Sun Ultra. The first line of the hex dump contains part of the header record:

```
0000000  0020 0002 0040 0000 0000 0000 0000 0000
```

The first pair of bytes contains the size of the header record, in this case 20_{16}—or 32_{10}. The next two pairs of bytes also contain integer values. If the same program is compiled and executed on a PC or a VAX, the hex dump of the first line will look like this:

```
0000000  2000 0200 4000 0000 0000 0000 0000 0000
```

Why are the bytes reversed in this version of the program? The answer is that in both cases the numbers were written to the file exactly as they appeared in memory, and the two different machines represent 2-byte integers differently—the Sun stores the high-order byte, followed by the low-order byte; the PC and VAX store the low-order byte, followed by the high-order byte.

This reverse order also applies to 4-byte integers on these machines. For example, in our discussion of file dumps we saw that the hexadecimal value of $500\,000\,000_{10}$ is $1dcd6500_{16}$. If you write this value out to a file on a PC, or some other reverse-order machine, a hex dump of the file created looks like this:

```
0000000  0065 cd1d
```

The problem of data representation is not restricted only to byte order of binary numbers. The way structures are laid out in memory can vary from machine to machine and compiler to compiler. For example, suppose you have a C program containing the following lines of code:

```
struct {
   int  cost;
   char ident[4];
} item;
...
write (fd, &item, sizeof(item));
```

and you want to write files using this code on two different machines, a Cray T90 and a Sun Ultra. Because it likes to operate on 64-bit words, Cray's C compiler allocates a minimum of 8 bytes for any element in a `struct`, so it allocates 16 bytes for the `struct item`. When it executes the `write` statement, then, the Cray writes 16 bytes to the file. The same program compiled on a Sun Ultra writes only 8 bytes, as you probably would expect, and on most PCs it writes 6 bytes: same exact program; same language; three different results.

Text is also encoded differently on different platforms. In this case the differences are primarily restricted to two different types of systems: those that use EBCDIC[5] and those that use ASCII. EBCDIC is a standard created by IBM, so machines that need to maintain compatibility with IBM must support EBCDIC. Most others support ASCII. A few support both. Hence, text written to a file from an EBCDIC-based machine may well not be readable by an ASCII-based machine.

Equally serious, when we go beyond simple English text, is the problem of representing different character sets from different national languages. This is an enormous problem for developers of text databases.

5.6.2 Achieving Portability

Differences among languages, operating systems, and machine architectures represent three major problems when we need to generate portable files. Achieving portability means determining how to deal with these differences. And the differences are often not just differences between two platforms, for many different platforms could be involved.

The most important requirement for achieving portability is to recognize that it is not a trivial matter and to take steps ahead of time to insure it. Following are some guidelines.

Agree on a Standard Physical Record Format and Stay with It

A physical standard is one that is represented the same physically, no matter what language, machine, or operating system is used. FITS is a good example of a physical standard, for it specifies exactly the physical format of each header record, the keywords that are allowed, the order in which keywords may appear, and the bit pattern that must be used to represent the binary numbers that describe the image.

5. EBCDIC stands for Extended Binary Coded Decimal Interchange Code.

Unfortunately, once a standard is established, it is very tempting to improve on it by changing it in some way, thereby rendering it no longer a standard. If the standard is sufficiently extensible, this temptation can sometimes be avoided. FITS, for example, has been extended a few times over its lifetime to support data objects that were not anticipated in its original design, yet all additions have remained compatible with the original format.

One way to make sure that a standard has staying power is to make it simple enough that files can be written in the standard format from a wide range of machines, languages, and operating systems. FITS again exemplifies such a standard. FITS headers are ASCII 80-byte records in blocks of thirty-six records each, and FITS images are stored as one contiguous block of numbers, both very simple structures that are easy to read and write in most modern operating systems and languages.

Agree on a Standard Binary Encoding for Data Elements

The two most common types of basic data elements are text and numbers. In the case of text, ASCII and EBCDIC represent the most common encoding schemes, with ASCII standard on virtually all machines except IBM mainframes. Depending on the anticipated environment, one of these should be used to represent all text.[6]

The situation for binary numbers is a little cloudier. Although the number of different encoding schemes is not large, the likelihood of having to share data among machines that use different binary encodings can be quite high, especially when the same data is processed both on large mainframes and on smaller computers. Two standards efforts have helped diminish the problem, however: IEEE Standard formats and External Data Representation (XDR).

IEEE has established standard format specifications for 32-bit, 64-bit, and 128-bit floating point numbers, and for 8-bit, 16-bit, and 32-bit integers. With a few notable exceptions (for example, IBM mainframes, Cray, and Digital), most computer manufacturers have followdd these guidelines in designing their machines. This effort goes a long way toward providing portable number encoding schemes.

XDR is an effort to go the rest of the way. XDR not only specifies a set of standard encodings for all files (the IEEE encodings) but provides for a

6. Actually, there are different versions of both ASCII and EBCDIC. However, for most applications and for the purposes of this text, it is sufficient to consider each as a single character set.

set of routines for each machine for converting from its binary encoding when writing to a file and vice versa (Fig. 5.11). Hence, when we want to store numbers in XDR, we can read or write them by replacing read and write routines in our program with XDR routines. The XDR routines take care of the conversions.[7]

Once again, FITS provides us with an excellent example: the binary numbers that constitute a FITS image must conform to the IEEE Standard. Any program written on a machine with XDR support can thus read and write portable FITS files.

Number and Text Conversion

Sometimes the use of standard data encodings is not feasible. For example, suppose you are working primarily on IBM mainframes with software that deals with floating point numbers and text. If you choose to store your data in IEEE Standard formats, every time your program reads or writes a

7. XDR is used for more than just number conversions. It allows a C programmer to describe arbitrary data structures in a machine-independent fashion. XDR originated as a Sun protocol for transmitting data that is accessed by more than one type of machine. For further information, see Sun (1986 or later).

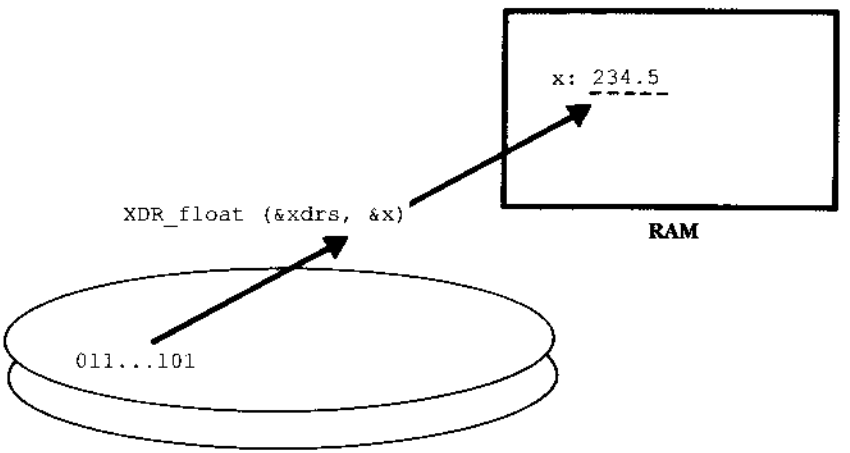


Figure 5.11 XDR specifies a standard external data representation for numbers stored in a file. XDR routines are provided for converting to and from the XDR representation to the encoding scheme used on the host machine. Here a routine called `XDR_float` translates a 32-bit floating point number from its XDR representation on disk to that of the host machine.

number or character, it must translate the number from the IBM format to the corresponding IEEE format. This is not only time-consuming but can result in loss of accuracy. It is probably better in this case to store your data in native IBM format in your files.

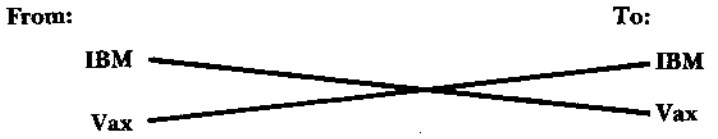
What happens, then, when you want to move your files back and forth between your IBM and a VAX, which uses a different native format for numbers and generally uses ASCII for text? You need a way to convert from the IBM format to the VAX format and back. One solution is to write (or borrow) a program that translates IBM numbers and text to their VAX equivalents, and vice versa. This simple solution is illustrated in Fig. 5.12(a).

But what if, in addition to IBM and VAX computers, you find that your data is likely to be shared among many different platforms that use different numeric encodings? One way to solve this problem is to write a program to convert from each of the representations to every other representation. This solution, illustrated in Fig. 5.12(b), can get rather complicated. In general, if you have n different encoding schemes, you will need $n(n-1)$ different translators. If n is large, this can be very messy. Not only do you need many translators, but you need to keep track, for each file, of where the file came from and/or where it is going in order to know which translator to use.

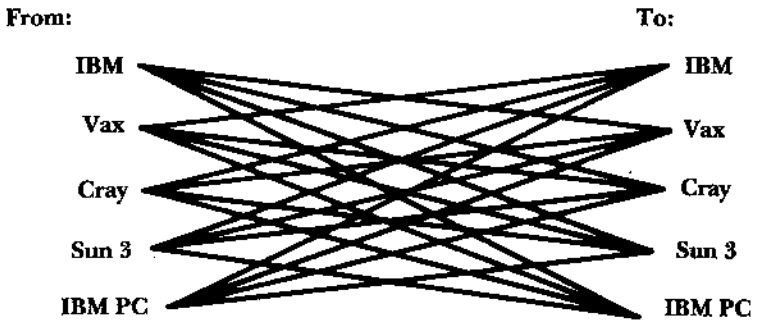
In this case, a better solution would probably be to agree on a standard intermediate format, such as XDR, and translate files into XDR whenever they are to be exported to a different platform. This solution is illustrated in Fig. 5.12(c). Not only does it reduce the number of translators from $n(n-1)$ to $2n$, but it should be easy to find translators to convert from most platforms to and from XDR. One negative aspect of this solution is that it requires *two* conversions to go from any one platform to another, a cost that has to be weighed against the complexity of providing $n(n-1)$ translators.

File Structure Conversion

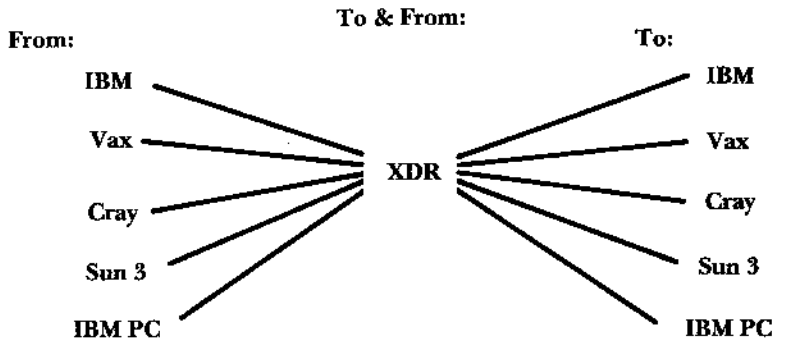
Suppose you are a doctor and you have X-ray raster images of a particular organ taken periodically over several minutes. You want to look at a certain image in the collection using a program that lets you zoom in and out and detect special features in the image. You have another program that lets you animate the collection of images, showing how it changes over several minutes. Finally, you want to annotate the images and store them in a special X-ray archive—and you have another program for doing that. What do you do if each of these three programs requires that your image be in a different format?



(a) Converting between IBM and Vax native format requires two conversion routines.



(b) Converting directly between five different native formats requires 20 conversion routines.



(c) Converting between five different native formats via an intermediate standard format requires 10 conversion routines.

Figure 5.12 Direct conversion between n native machines formats requires $n(n-1)$ conversion routines, as illustrated in (a) and (b). Conversion via an intermediate standard format requires $2n$ conversion routines, as illustrated in (c).

The conversion problems that apply to atomic data encoding also apply to file structures for more complex objects, like images, but at a different level. Whereas character and number encoding are tied closely to specific platforms, more complex objects and their representations just as often are tied to specific *applications*.

For example, there are many software packages that deal with images and very little agreement about a file format for storing them. When we look at this software, we find different solutions to this problem.

- Require that the user supply images in a format that is compatible with the one used by the package. This places the responsibility on the user to convert from one format to another. For such situations, it may be preferable to provide utility programs that translate from one format to another and that are invoked whenever translating.
- Process only images that adhere to some predefined standard format. This places the responsibility on a community of users and software developers for agreeing on and enforcing a standard. FITS is a good example of this approach.
- Include different sets of I/O methods capable of converting an image from several different formats into a standard memory structure that the package can work with. This places the burden on the software developer to develop I/O methods for file object types that may be stored differently but for the purposes of an application are conceptually the same. You may recognize this approach as a variation on the concept of *object-oriented access* that we discussed earlier.

File System Differences

Finally, if you move files from one file system to another, chances are you will find differences in the way files are organized physically. For example, Unix systems write files to tapes in 512-byte blocks, but non-Unix systems often use different block sizes, such as 2880-bytes—thirty-six 80-byte records. (Guess where the FITS blocking format comes from?) When transferring files between systems, you may need to deal with this problem.

Unix and Portability

Recognizing problems such as the block-size problem just described, Unix provides a utility called `dd`. Although `dd` is intended primarily for copying tape data to and from Unix systems, it can be used to convert data

from any physical source. The dd utility provides the following options, among others:

- Convert from one block size to another,
- Convert fixed-length records to variable-length, or vice versa,
- Convert ASCII to EBCDIC, or vice versa,
- Convert all characters to lowercase (or to uppercase), and
- Swap every pair of bytes.

Of course, the greatest contribution Unix makes to the problems discussed here is Unix itself. By its simplicity and ubiquity, Unix encourages the use of the same operating system, the same file system, the same views of devices, and the same general views of file organization, no matter what particular hardware platform you happen to be using.

For example, one of the authors works in an organization with a nationwide constituency that operates many different computers, including two Crays, a Connection Machine, and many Sun, Apple, IBM, Silicon Graphics, and Digital workstations. Because each runs some flavor of Unix, they all incorporate precisely the same view of all external storage devices, they all use ASCII, and they all provide the same basic programming environment and file management utilities. Files are not perfectly portable within this environment, for reasons that we have covered in this chapter; but the availability of Unix goes a long way toward facilitating the rapid and easy transfer of files among the applications, programming environments, and hardware systems that the organization supports.

SUMMARY

One higher level of organization, in which records are grouped into *blocks*, is also often imposed on files. This level is imposed to improve I/O performance rather than our logical view of the file.

Sometimes we identify individual records by their *relative record numbers* (RRNs) in a file. It is also common, however, to identify a record by a *key* whose value is based on some of the record's content. Key values must occur in, or be converted to, some predetermined *canonical form* if they are to be recognized accurately and unambiguously by programs. If every record's key value is distinct from all others, the key can be used to identify and locate the unique record in the file. Keys that are used in this way are called *primary keys*.

In this chapter we look at the technique of searching sequentially through a file looking for a record with a particular key. Sequential search can perform poorly for long files, but there are times when sequential searching is reasonable. Record blocking can be used to improve the I/O time for a sequential search substantially. Two useful Unix utilities that process files sequentially are `wc` and `grep`.

In our discussion of ways to separate records, it is clear that some of the methods provide a mechanism for looking up or calculating the *byte offset* of the beginning of a record. This, in turn, opens up the possibility of accessing the record *directly*, by RRN, rather than sequentially.

The simplest record formats for permitting direct access by RRN involve the use of fixed-length records. When the data comes in fixed-size quantities (for example, zip codes), fixed-length records can provide good performance and good space utilization. If there is a lot of variation in the amount and size of data in records, however, the use of fixed-length records can result in expensive waste of space. In such cases the designer should look carefully at the possibility of using variable-length records.

Sometimes it is helpful to keep track of general information about files, such as the number of records they contain. A *header record*, stored at the beginning of the file it pertains to, is a useful tool for storing this kind of information. Header records have been added to the I/O buffer class and class `BufferFile`. These headers support a guarantee of consistent access to records in files.

It is important to be aware of the difference between *file access* and *file organization*. We try to organize files in such a way that they give us the types of access we need for a particular application. For example, one of the advantages of a fixed-length record organization is that it allows access that is either sequential or direct.

In addition to the traditional view of a file as a more or less regular collection of fields and records, we present a more purely logical view of the contents of files in terms of *abstract data models*, a view that lets applications ignore the physical structure of files altogether.

Defining a single class to support file operations for arbitrary data objects requires the use of C++ *templates*. Class `RecordFile` implements this abstract data model approach as a template class with a single parameter. The application programmer need only define `Pack` and `Unpack` methods, using the buffer classes defined in Chapter 4, and `RecordFile` does the rest. The application can create, open, and close files, and read and write records with no additional concern about file structures.

This abstract data model view is often more appropriate to data objects such as sound, images, and documents. We call files *self-describing* when they do not require an application to reveal their structure but provide that information themselves. Another concept that deviates from the traditional view is *metadata,* in which the file contains data that describes the primary data in the file. FITS files, used for storing astronomical images, contain extensive headers with metadata.

The use of abstract data models, self-describing files, and metadata makes it possible to mix a variety of different types of data objects in one file. When this is the case, file access is more object oriented. Abstract data models also facilitate *extensible* files—files whose structures can be extended to accommodate new kinds of objects.

Portability becomes increasingly important as files are used in more heterogeneous computing environments. Differences among operating systems, languages, and machine architectures all lead to the need for portability. One important way to foster portability is *standardization,* which means agreeing on physical formats, encodings for data elements, and file structures.

If a standard does not exist and it becomes necessary to convert from one format to another, it is still often much simpler to have one standard format that all converters convert into and out of. Unix provides a utility called dd that facilitates data conversion. The Unix environment supports portability simply by being commonly available on a large number of platforms.

KEY TERMS

Block. A collection of records stored as a physically contiguous unit on secondary storage. In this chapter, we use record blocking to improve I/O performance during sequential searching.

Canonical form. A standard form for a key that can be derived, by the application of well-defined rules, from the particular, nonstandard form of the data found in a record's key field(s) or provided in a search request supplied by a user.

Direct access. A file accessing mode that involves jumping to the exact location of a record. Direct access to a fixed-length record is usually accomplished by using its *relative record number* (RRN), computing its byte offset, and then seeking to the first byte of the record.

Extensibility. A characteristic of some file organizations that makes it possible to extend the types of objects that the format can accommodate without having to redesign the format. For example, tagged file formats lend themselves to extensibility, for they allow the addition of new tags for new data objects and associated new methods for accessing the objects.

File-access method. The approach used to locate information in a file. In general, the two alternatives are *sequential access* and *direct access.*

File organization method. The combination of conceptual and physical structures used to distinguish one record from another and one field from another. An example of a kind of file organization is fixed-length records containing variable numbers of variable-length delimited fields.

Header record. A record placed at the beginning of a file that is used to store information about the file contents and the file organization.

Key. An expression derived from one or more of the fields within a record that can be used to locate that record. The fields used to build the key are sometimes called the *key fields.* Keyed access provides a way of performing content-based retrieval of records, rather than retrieval based merely on a record's position.

Metadata. Data in a file that is not the primary data but describes the primary data in a file. Metadata can be incorporated into any file whose primary data requires supporting information. If a file is going to be shared by many users, some of whom might not otherwise have easy access to its metadata, it may be most convenient to store the metadata in the file itself. A common place to store metadata in a file is the header record.

Portability. That characteristic of files that describes how amenable they are to access on a variety of different machines, via a variety of different operating systems, languages, and applications.

Primary key. A key that uniquely identifies each record and is used as the primary method of accessing the records.

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

Relative record number (RRN). An index giving the position of a record relative to the beginning of its file. If a file has fixed-length records, the RRN can be used to calculate the *byte offset* of a record so the record can be accessed directly.

Representation-independent file access. A form of file access in which applications access data objects in terms of the applications' in-memory view of the objects. Separate methods associated with the objects are responsible for translating to and from the physical format of the object, letting the application concentrate on the task at hand.

Self-describing files. Files that contain information such as the number of records in the file and formal descriptions of the file's record structure, which can be used by software in determining how to access the file. A file's header is a good place for this information.

Sequential access. Sequential access to a file means reading the file from the beginning and continuing until you have read in everything that you need. The alternative is direct access.

Sequential search. A method of searching a file by reading the file from the beginning and continuing until the desired record has been found.

Template class. A parameterized class definition. Multiple classes can share the same definition and code through the use of template classes and template functions in C++.

FURTHER READINGS

Sweet (1985) is a short but stimulating article on key field design. A number of interesting algorithms for improving performance in sequential searches are described in Gonnet (1984) and, of course, in Knuth (1973b).

Self-describing file formats like FITS—see Wells, Greisen, and Harten (1981)—for scientific files have had significant development over the past years. Two of the most prominent format strategies are the Hierarchical Data Format (HDF), available from the HDF Web site at http://hdf.ncsa.uiuc.edu, and the Common Data Format (CDF) which has a web site at http://nssdc.gsfc.nasa.gov/cdf/cdf_home.html.

EXERCISES

1. If a key in a record is already in canonical form and the key is the first field of the record, it is possible to search for a record by key without ever separating out the key field from the rest of the fields. Explain.
2. It has been suggested (Sweet, 1985) that primary keys should be "dataless, unchanging, unambiguous, and unique." These concepts are

interrelated since, for example, a key that contains data runs a greater risk of changing than a dataless key. Discuss the importance of each of these concepts, and show by example how their absence can cause problems. The primary key used in our example file violates at least one of the criteria. How might you redesign the file (and possibly its corresponding information content) so primary keys satisfy these criteria?

3. How many comparisons would be required on the average to find a record using sequential search in a 100 000-record disk file? If the record is not in the file, how many comparisons are required? If the file is blocked so that 50 records are stored per block, how many disk accesses are required on average? What if only one record is stored per block?
4. In our evaluation of performance for sequential search, we assume that every read results in a seek. How do the assumptions change on a single-user machine with access to a magnetic disk? How do these changed assumptions affect the analysis of sequential searching?
5. Design a header structure for a `Person` file of fixed-sized records that stores the names of the fields in addition to the sizes of the fields. How would you have to modify class `FixedFieldBuffer` to support the use of such a header?
6. Separate code must be generated for each instantiation of a template class, but there is no standard for controlling this code generation. What is the mechanism in your C++ compiler that is used to describe when to generate code for template instances?
7. In our discussion of the uses of relative record numbers (RRNs), we suggest that you can create a file in which there is a direct correspondence between a primary key, such as membership number, and RRN, so we can find a person's record by knowing just the name or membership number. What kinds of difficulties can you envision with this simple correspondence between membership number and RRN? What happens if we want to delete a name? What happens if we change the information in a record in a variable-length record file and the new record is longer?
8. Assume that we have a variable-length record file with long records (greater than 1000 bytes each, on the average). Assume that we are looking for a record with a particular RRN. Describe the benefits of using the contents of a byte count field to skip sequentially from

record to record to find the one we want. This is called *skip sequential* processing. Use your knowledge of system buffering to describe why this is useful only for long records. If the records are sorted in order by key and blocked, what information do you have to place at the start of each block to permit even faster skip sequential processing?

9. Suppose you have a fixed-length record with fixed-length fields, and the sum of the field lengths is 30 bytes. A record with a length of 30 bytes would hold them all. If we intend to store the records on a sectored disk with 512-byte sectors (see Chapter 3), we might decide to pad the record out to 32 bytes so we can place an integral number of records in a sector. Why would we want to do this?
10. Why is it important to distinguish between file access and file organization?
11. What is an abstract data model? Why did the early file processing programs not deal with abstract data models? What are the advantages of using abstract data models in applications? In what way does the Unix concept of standard input and standard output conform to the notion of an abstract data model? (See "Physical Files and Logical Files in Unix" in Chapter 2.)
12. What is metadata?
13. In the FITS header in Fig. 5.7, some metadata provides information about the file's structure, and some provides information about the scientific context in which the corresponding image was recorded. Give three examples of each.
14. In the FITS header in Fig. 5.7, there is enough information for a program to determine how to read the entire file. Assuming that the size of the block containing the header must be a multiple of 2,880 bytes, how large is the file? What proportion of the file contains header information?
15. In the discussion of field organization, we list the "keyword = value" construct as one possible type of field organization. How is this notion applied in tagged file structures? How does a tagged file structure support object-oriented file access? How do tagged file formats support extensibility?
16. List three factors that affect portability in files.
17. List three ways that portability can be achieved in files.
18. What is XDR? XDR is actually much more extensive than what we described in this chapter. If you have access to XDR documentation

(see "Further Readings" at the end of this chapter), look up XDR and list the ways that it supports portability.

19. What is the IEEE standard format for 32-bit, 64-bit, and 128-bit floating point values? Does your computer implement floating point values in the IEEE format?

PROGRAMMING EXERCISES

20. Implement methods such as `findByLastName(char*)`, `findByFirstName(char*)`, and so on, that search through a `BufferFile<Person>` for a record that has the appropriate field that matches the argument.

21. Write a `ReadByRRN` method for variable-length record files that finds a record on the basis of its position in the file. For example, if requested to find the 547th record in a file, it would read through the first 546 records and then print the contents of the 547th record. Implement skip sequential search (see Exercise 8) to avoid reading the contents of unwanted records.

22. Write a driver for `findByLastName` that reads names from a separate transaction file that contains only the keys of the records to be extracted. Write the selected records to a separate output file. First, assume that the records are in no particular order. Then assume that both the main file and the transaction file are sorted by key. In the latter case, how can you make your program more efficient?

23. Implement an update operation for class `BufferFile` that works for fixed-length record file. Write a driver program that allows a user to select a record-by-record number and enter new values for all of the fields.

24. Make any or all of the following alterations to the update function from Exercise 23.
    a. Let the user identify the record to be changed by name, rather than RRN.
    b. Let the user change individual fields without having to change an entire record.
    c. Let the user choose to view the entire file.

25. Write a program that reads a file and outputs the file contents as a file dump. The file dump should have a format similar to the one used in the examples in this chapter. The program should accept the name of

the input file on the command line. Output should be to standard output (terminal screen).

26. Develop a set of rules for translating the dates August 7, 1949, Aug. 7, 1949, 8-7-49, 08-07-49, 8/7/49, and other, similar variations into a common canonical form. Write a function that accepts a string containing a date in one of these forms and returns the canonical form, according to your rules. Be sure to document the limitations of your rules and function.