File Structures

An Object-Oriented Approach with C++

Michael J. Folk
University of Illinois

Bill Zoellick
CAP Ventures

Greg Riccardi
Florida State University

ADDISON-WESLEY

Addison-Wesley is an imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

Contents

Preface vii

Chapter 1 Introduction to the Design and Specification of File Structures 1

1.1 The Heart of File Structure Design 2
1.2 A Short History of File Structure Design 3
1.3 A Conceptual Toolkit: File Structure Literacy 5
1.4 An Object-Oriented Toolkit: Making File Structures Usable 6
1.5 Using Objects in C++ 6
Summary 10 Key Terms 11 Further Readings 12
Programming Project 12

Chapter 2 Fundamental File Processing Operations 13

2.1 Physical Files and Logical Files 14
2.2 Opening Files 15
2.3 Closing Files 19
2.4 Reading and Writing 20
2.4.1 Read and Write Functions 20
2.4.2 Files with C Streams and C++ Stream Classes 21
2.4.3 Programs in C++ to Display the Contents of a File 23
2.4.4 Detecting End-of-File 26
2.5 Seeking 27
2.5.1 Seeking with C Streams 27
2.5.2 Seeking with C++ Stream Classes 29
2.6 Special Characters in Files 29
2.7 The Unix Directory Structure 30
2.8 Physical Devices and Logical Files 32
2.8.1 Physical Devices as Files 32
2.8.2 The Console, the Keyboard, and Standard Error 32
2.8.3 I/O Redirection and Pipes 33
2.9 File-Related Header Files 34
2.10 Unix File System Commands 35
Summary 35 Key Terms 37 Further Readings 39 Exercises 40
Programming Exercises 41 Programming Project 42

Chapter 3 Secondary Storage and System Software 43

3.1 Disks 46
3.1.1 The Organization of Disks 46
3.1.2 Estimating Capacities and Space Needs 48
3.1.3 Organizing Tracks by Sector 50
3.1.4 Organizing Tracks by Block 55
3.1.5 Nondata Overhead 56
3.1.6 The Cost of a Disk Access 58
3.1.7 Effect of Block Size on Performance: A Unix Example 62
3.1.8 Disk as Bottleneck 63
3.2 Magnetic Tape 65
3.2.1 Types of Tape Systems 66
3.2.2 An Example of a High Performance Tape System 67
3.2.3 Organization of Data on Nine-Track Tapes 67
3.2.4 Estimating Tape Length Requirements 69
3.2.5 Estimating Data Transmission Times 71
3.3 Disk versus Tape 72
3.4 Introduction to CD-ROM 73
3.4.1 A Short History of CD-ROM 73
3.4.2 CD-ROM as a File Structure Problem 76
3.5 Physical Organization of CD-ROM 76
3.5.1 Reading Pits and Lands 77
3.5.2 CLV instead of CAV 78
3.5.3 Addressing 79
3.5.4 Structure of a Sector 80
3.6 CD-ROM Strengths and Weaknesses 82
3.6.1 Seek Performance 82
3.6.2 Data Transfer Rate 82
3.6.3 Storage Capacity 83
3.6.4 Read-Only Access 83
3.6.5 Asymmetric Writing and Reading 83
3.7 Storage as a Hierarchy 84
3.8 A Journey of a Byte 85
3.8.1 The File Manager 86
3.8.2 The I/O Buffer 87
3.8.3 The Byte Leaves Memory: The I/O Processor and Disk Controller 87
3.9 Buffer Management 90
3.9.1 Buffer Bottlenecks 90
3.9.2 Buffering Strategies 91
3.10 I/O in Unix 94
3.10.1 The Kernel 94
3.10.2 Linking File Names to Files 98

3.10.3 Normal Files, Special Files, and Sockets 100
3.10.4 Block I/O 100
3.10.5 Device Drivers 101
3.10.6 The Kernel and File Systems 101
3.10.7 Magnetic Tape and Unix 102
Summary 103 Key Terms 105 Further Readings 110 Exercises 110

Chapter 4 Fundamental File Structure Concepts 117

4.1 Field and Record Organization 118
4.1.1 A Stream File 119
4.1.2 Field Structures 120
4.1.3 Reading a Stream of Fields 123
4.1.4 Record Structures 125
4.1.5 A Record Structure That Uses a Length Indicator 127
4.1.6 Mixing Numbers and Characters: Use of a File Dump 131
4.2 Using Classes to Manipulate Buffers 134
4.2.1 Buffer Class for Delimited Text Fields 134
4.2.2 Extending Class Person with Buffer Operations 136
4.2.3 Buffer Classes for Length-Based and Fixed-Length Fields 137
4.3 Using Inheritance for Record Buffer Classes 139
4.3.1 Inheritance in the C++ Stream Classes 139
4.3.2 A Class Hierarchy for Record Buffer Objects 140
4.4 Managing Fixed-Length, Fixed-Field Buffers 144
4.5 An Object-Oriented Class for Record Files 146
Summary 147 Key Terms 148 Further Readings 149 Exercises 149
Programming Exercises 151 Programming Project 152

Chapter 5 Managing Files of Records 153

5.1 Record Access 154
5.1.1 Record Keys 154
5.1.2 A Sequential Search 156
5.1.3 Unix Tools for Sequential Processing 159
5.1.4 Direct Access 161
5.2 More about Record Structures 163
5.2.1 Choosing a Record Structure and Record Length 163
5.2.2 Header Records 165
5.2.3 Adding Headers to C++ Buffer Classes 167
5.3 Encapsulating Record I/O Operations in a Single Class 168
5.4 File Access and File Organization 170

5.5 **Beyond Record Structures** 172
5.5.1 Abstract Data Models for File Access 172
5.5.2 Headers and Self-Describing Files 173
5.5.3 Metadata 174
5.5.4 Color Raster Images 176
5.5.5 Mixing Object Types in One File 179
5.5.6 Representation-Independent File Access 182
5.5.7 Extensibility 183
5.6 **Portability and Standardization** 184
5.6.1 Factors Affecting Portability 184
5.6.2 Achieving Portability 186
Summary 192 Key Terms 194 Further Readings 196
Exercises 196 Programming Exercises 199

Chapter 6 Organizing Files for Performance 201

6.1 **Data Compression** 203
6.1.1 Using a Different Notation 203
6.1.2 Suppressing Repeating Sequences 204
6.1.3 Assigning Variable-Length Codes 206
6.1.4 Irreversible Compression Techniques 207
6.1.5 Compression in Unix 207
6.2 **Reclaiming Space in Files** 208
6.2.1 Record Deletion and Storage Compaction 209
6.2.2 Deleting Fixed-Length Records for Reclaiming Space Dynamically 210
6.2.3 Deleting Variable-Length Records 214
6.2.4 Storage Fragmentation 217
6.2.5 Placement Strategies 220
6.3 **Finding Things Quickly: An Introduction to Internal Sorting and Binary Searching** 222
6.3.1 Finding Things in Simple Field and Record Files 222
6.3.2 Search by Guessing: Binary Search 223
6.3.3 Binary Search versus Sequential Search 225
6.3.4 Sorting a Disk File in Memory 226
6.3.5 The Limitations of Binary Searching and Internal Sorting 226
6.4 **Keysorting** 228
6.4.1 Description of the Method 229
6.4.2 Limitations of the Keysort Method 232
6.4.3 Another Solution: Why Bother to Write the File Back? 232
6.4.4 Pinned Records 234
Summary 234 Key Terms 238 Further Readings 240 Exercises 241
Programming Exercises 243 Programming Project 245

CHAPTER 1

Introduction to the Design and Specification of File Structures

CHAPTER OBJECTIVES

- Introduce the primary design issues that characterize file structure design.
- Survey the history of file structure design, since tracing the developments in file structures teaches us much about how to design our own file structures.
- Introduce the notions of file structure literacy and of a *conceptual toolkit* for file structure design.
- Discuss the need for precise specification of data structures and operations and the development of an object-oriented toolkit that makes file structures easy to use.
- Introduce classes and overloading in the C++ language.

CHAPTER OUTLINE

1.1 The Heart of File Structure Design
1.2 A Short History of File Structure Design
1.3 A Conceptual Toolkit: File Structure Literacy
1.4 An Object-Oriented Toolkit: Making File Structures Usable
1.5 Using Objects in C++

1.1 The Heart of File Structure Design

Disks are slow. They are also technological marvels: one can pack thousands of megabytes on a disk that fits into a notebook computer. Only a few years ago, disks with that kind of capacity looked like small washing machines. However, relative to other parts of a computer, disks are slow.

How slow? The time it takes to get information back from even relatively slow electronic random access memory (RAM) is about 120 nanoseconds, or 120 billionths of a second. Getting the same information from a typical disk might take 30 milliseconds, or 30 thousandths of a second. To understand the size of this difference, we need an analogy. Assume that memory access is like finding something in the index of this book. Let's say that this local, book-in-hand access takes 20 seconds. Assume that accessing a disk is like sending to a library for the information you cannot find here in this book. Given that our "memory access" takes 20 seconds, how long does the "disk access" to the library take, keeping the ratio the same as that of a real memory access and disk access? The disk access is a quarter of a million times longer than the memory access. This means that getting information back from the library takes 5 million seconds, or almost 58 days. Disks are *very* slow compared with memory.

On the other hand, disks provide enormous capacity at much less cost than memory. They also keep the information stored on them when they are turned off. The tension between a disk's relatively slow access time and its enormous, nonvolatile capacity is the driving force behind file structure design. Good file structure design will give us access to all the capacity without making our applications spend a lot of time waiting for the disk.

A *file structure* is a combination of representations for data in files and of operations for accessing the data. A file structure allows applications to read, write, and modify data. It might also support finding the data that matches some search criteria or reading through the data in some particular order. An improvement in file structure design may make an application hundreds of times faster. The details of the representation of the data and the implementation of the operations determine the efficiency of the file structure for particular applications.

A tremendous variety in the types of data and in the needs of applications makes file structure design very important. What is best for one situation may be terrible for another.

1.2 A Short History of File Structure Design

Our goal is to show you how to think creatively about file structure design problems. Part of our approach draws on history: after introducing basic principles of design, we devote the last part of this book to studying some of the key developments in file design over the last thirty years. The problems that researchers struggle with reflect the same issues that you confront in addressing any substantial file design problem. Working through the approaches to major file design issues shows you a lot about how to approach new design problems.

The general goals of research and development in file structures can be drawn directly from our library analogy.

- Ideally, we would like to get the information we need with one access to the disk. In terms of our analogy, we do not want to issue a series of fifty-eight-day requests before we get what we want.
- If it is impossible to get what we need in one access, we want structures that allow us to find the target information with as few accesses as possible. For example, you may remember from your studies of data structures that a binary search allows us to find a particular record among fifty thousand other records with no more than sixteen comparisons. But having to look sixteen places on a disk before finding what we want takes too much time. We need file structures that allow us to find what we need with only two or three trips to the disk.
- We want our file structures to group information so we are likely to get everything we need with only one trip to the disk. If we need a client's name, address, phone number, and account balance, we would prefer to get all that information at once, rather than having to look in several places for it.

It is relatively easy to come up with file structure designs that meet these goals when we have files that never change. Designing file structures that maintain these qualities as files change, grow, or shrink when information is added and deleted is much more difficult.

Early work with files presumed that files were on tape, since most files were. Access was sequential, and the cost of access grew in direct proportion to the size of the file. As files grew intolerably large for unaided sequential access and as storage devices such as disk drives became available, indexes were added to files. The indexes made it possible to keep a list of keys and pointers in a smaller file that could be searched more quickly. With the key and pointer, the user had direct access to the large, primary file.

Unfortunately, simple indexes had some of the same sequential flavor as the data files, and as the indexes grew, they too became difficult to manage, especially for dynamic files in which the set of keys changes. Then, in the early 1960s, the idea of applying tree structures emerged. Unfortunately, trees can grow very unevenly as records are added and deleted, resulting in long searches requiring many disk accesses to find a record.

In 1963 researchers developed the tree, an elegant, self-adjusting binary tree structure, called an AVL tree, for data in memory. Other researchers began to look for ways to apply AVL trees, or something like them, to files. The problem was that even with a balanced binary tree, dozens of accesses were required to find a record in even moderate-sized files. A method was needed to keep a tree balanced when each node of the tree was not a single record, as in a binary tree, but a file block containing dozens, perhaps even hundreds, of records.

It took nearly ten more years of design work before a solution emerged in the form of the *B-tree*. Part of the reason finding a solution took so long was that the approach required for file structures was very different from the approach that worked in memory. Whereas AVL trees grow from the top down as records are added, B-trees grow from the bottom up.

B-trees provided excellent access performance, but there was a cost: no longer could a file be accessed sequentially with efficiency. Fortunately, this problem was solved almost immediately by adding a linked list structure at the bottom level of the B-tree. The combination of a B-tree and a sequential linked list is called a *B^+ tree.*

Over the next ten years, B-trees and B^+ trees became the basis for many commercial file systems, since they provide access times that grow in proportion to $\log_k N$, where N is the number of entries in the file and k is the number of entries indexed in a single block of the B-tree structure. In practical terms, this means that B-trees can guarantee that you can find one file entry among millions of others with only three or four trips to the disk. Further, B-trees guarantee that as you add and delete entries, performance stays about the same.

Being able to retrieve information with just three or four accesses is pretty good. But how about our goal of being able to get what we want with a single request? An approach called *hashing* is a good way to do that with files that do not change size greatly over time. From early on, hashed indexes were used to provide fast access to files. However, until recently, hashing did not work well with volatile, dynamic files. After the development of B-trees, researchers turned to work on systems for extendible, dynamic hashing that could retrieve information with one or, at most, two disk accesses no matter how big the file became.

1.3 A Conceptual Toolkit: File Structure Literacy

As we move through the developments in file structures over the last three decades, watching file structure design evolve as it addresses dynamic files first sequentially, then through tree structures, and finally through direct access, we see that the same design problems and design tools keep emerging. We decrease the number of disk accesses by collecting data into buffers, blocks, or buckets; we manage the growth of these collections by splitting them, which requires that we find a way to increase our address or index space, and so on. Progress takes the form of finding new ways to combine these basic tools of file design.

We think of these tools as *conceptual* tools. They are methods of framing and addressing a design problem. Each tool combines ways of representing data with specific operations. Our own work in file structures has shown us that by understanding the tools thoroughly and by studying how the tools have evolved to produce such diverse approaches as B-trees and extendible hashing, we develop mastery and flexibility in our own use of the tools. In other words, we acquire literacy with regard to file structures. This text is designed to help readers acquire file structure literacy. Chapters 2 through 6 introduce the basic tools; Chapters 7 through 11 introduce readers to the highlights of the past several decades of file structure design, showing how the basic tools are used to handle efficient sequential access—B-trees, B^+ trees, hashed indexes, and extendible, dynamic hashed files.

1.4 An Object-Oriented Toolkit: Making File Structures Usable

Making file structures usable in application development requires turning this conceptual toolkit into application programming interfaces— collections of data types and operations that can be used in applications. We have chosen to employ an object-oriented approach in which data types and operations are presented in a unified fashion as *class* definitions. Each particular approach to representing some aspect of a file structure is represented by one or more classes of objects.

A major problem in describing the classes that can be used for file structure design is that they are complicated and progressive. New classes are often modifications or extensions of other classes, and the details of the data representations and operations become ever more complex. The most effective strategy for describing these classes is to give specific representations in the simplest fashion. In this text, use the C++ programming language to give precise specifications to the file structure classes. From the first chapter to the last, this allows us to build one class on top of another in a concise and understandable fashion.

1.5 Using Objects in C++

In an object-oriented information system, data content and behavior are integrated into a single design. The objects of the system are divided into classes of objects with common characteristics. Each class is described by its *members*, which are either data attributes *(data members)* or functions *(member functions* or *methods)*. This book illustrates the principles of object-oriented design through implementations of file structures and file operations as C++ classes. These classes are also an extensive presentation of the features of C++. In this section, we look at some of the features of objects in C++, including class definitions, constructors, public and private sections, and operator overloading. Later chapters show how to make effective use of inheritance, virtual functions, and templates.

An example of a very simple C++ class is Person, as given below.

```
class Person
{ public:
   // data members
   char LastName [11], FirstName [11], Address [16];
   char City [16], State [3], ZipCode [10];
   // method
   Person (); // default constructor
};
```

Each Person object has first and last names, address, city, state, and zip code, which are declared as members, just as they would be in a C struct. For an object p of type Person, p.LastName refers to its LastName member.

The public label specifies that the following members and methods are part of the interface to objects of the class. These members and methods can be freely accessed by any users of Person objects. There are three levels of access to class members: public, private, and protected. The last two restrict access and will be described later in the book. The only significant difference in C++ between struct and class is that for struct members the default access is public, and for class members the default access is private.

Each of these member fields is represented by a character array of fixed size. However, the usual style of dealing with character arrays in C++ is to represent the value of the array as a null-delimited, variable-sized string with a maximum length. The number of characters in the representation of a string is one more than the number of characters in the string. The LastName field, for example, is represented by an array of eleven characters and can hold a string of length between 0 and 10. Proper use of strings in C++ is dependent on ensuring that every string variable is initialized before it is used.

C++ includes special methods called *constructors* that are used to provide a guarantee that every object is properly initialized.[1] A constructor is a method with no return type whose name is the same as the class. Whenever an object is created, a constructor is called. The two ways that objects are created in C++ are by the declaration of a variable (automatic creation) and by the execution of a new operation (dynamic creation):

1. A *destructor* is a method of a class that is executed whenever an object is destroyed. A destructor for class Person has definition ~Person (). Examples of destructors are given in later chapters.

```
Person p; // automatic creation
Person * p_ptr = new Person; // dynamic creation
```

Execution of either of the object creation statements above includes the execution of the `Person` constructor. Hence, we are sure that every `Person` object has been properly initialized before it is used. The code for the `Person` constructor initializes each member to an empty string by assigning 0 (null) to the first character:

```
Person::Person ()
{ // set each field to an empty string
   LastName [0] = 0; FirstName [0] = 0; Address [0] = 0;
   City [0] = 0; State [0] = 0; ZipCode [0] = 0;
}
```

The symbol :: is the *scope resolution operator*. In this case, it tells us that `Person()` is a method of class `Person`. Notice that within the method code, the members can be referenced without the dot (.) operator. Every call on a member function has a pointer to an object as a hidden argument. The implicit argument can be explicitly referred to with the keyword `this`. Within the method, `this->LastName` is the same as `LastName`.

Overloading of symbols in programming languages allows a particular symbol to have more than one meaning. The meaning of each instance of the symbol depends on the context. We are very familiar with overloading of arithmetic operators to have different meanings depending on the operand type. For example, the symbol + is used for both integer and floating point addition. C++ supports the use of overloading by programmers for a wide variety of symbols. We can create new meanings for operator symbols and for named functions.

The following class `String` illustrates extensive use of overloading: there are three constructors, and the operators = and == are overloaded with new meanings:

```
class String
{public:
   String (); // default constructor
   String (const String&); //copy constructor
   String (const char *); // create from C string
   ~String (); // destructor
   String & operator = (const String &); // assignment
   int operator == (const String &) const; // equality
   char * operator char*() // conversion to char *
      {return strdup(string);} // inline body of method
private:
   char * string; // represent value as C string
   int MaxLength;
};
```

The data members, `string` and `MaxLength`, of class `String` are in the `private` section of the class. Access to these members is restricted. They can be referenced only from inside the code of methods of the class. Hence, users of `String` objects cannot directly manipulate these members. A conversion operator (`operator char *`) has been provided to allow the use of the value of a `String` object as a C string. The body of this operator is given *inline*, that is, directly in the class definition. To protect the value of the `String` from direct manipulation, a copy of the string value is returned. This operator allows a `String` object to be used as a `char *`. For example, the following code creates a `String` object `s1` and copies its value to normal C string:

```
String s1 ("abcdefg"); // uses String::String (const char *)
char str[10];
strcpy (str, s1); // uses String::operator char * ()
```

The new definition of the assignment operator (`operator =`) replaces the standard meaning, which in C and C++ is to copy the bit pattern of one object to another. For two objects `s1` and `s2` of class `String`, `s1 = s2` would copy the value of `s1.string` (a pointer) to `s2.string`. Hence, `s1.string` and `s2.string` point to the same character array. In essence, `s1` and `s2` become aliases. Once the two fields point to the same array, a change in the string value of `s1` would also change `s2`. This is contrary to how we expect variables to behave. The implementation of the assignment operator and an example of its use are:

```
String & String::operator = (const String & str)
{ // code for assignment operator
     strcpy (string, str.string);
     return *this;
}
String s1, s2;
s1 = s2; // using overloaded assignment
```

In the assignment `s1 = s2`, the hidden argument (`this`) refers to `s1`, and the explicit argument `str` refers to `s2`. The line `strcpy (string, str.string);` copies the contents of the string member of `s2` to the string member of `s1`. This assignment operator does not create the alias problem that occurs with the standard meaning of assignment.

To complete the class `String`, we add the copy constructor, which is used whenever a copy of a string is needed, and the equality operator (`operator ==`), which makes two `String` objects equal if the array contents are the same. The predefined meaning for these operators performs pointer copy and pointer comparison, respectively. The full specification and implementation of class `String` are given in Appendix G.

SUMMARY

The key design problem that shapes file structure design is the relatively large amount of time that is required to get information from a disk. All file structure designs focus on minimizing disk accesses and maximizing the likelihood that the information the user will want is already in memory.

This text begins by introducing the basic concepts and issues associated with file structures. The last half of the book tracks the development of file structure design as it has evolved over the last thirty years. The key problem addressed throughout this evolution has been finding ways to minimize disk accesses for files that keep changing in content and size. Tracking these developments takes us first through work on sequential file access, then through developments in tree-structured access, and finally to relatively recent work on direct access to information in files.

Our experience has been that the study of the principal research and design contributions to file structures—focusing on how the design work uses the same tools in new ways—provides a solid foundation for thinking creatively about new problems in file structure design. The presentation of these tools in an object-oriented design makes them tremendously useful in solving real problems.

Object-oriented programming supports the integration of data content and behavior into a single design. C++ class definitions contain both data and function members and allow programmers to control precisely the manipulation of objects. The use of overloading, constructors, and private members enhances the programmer's ability to control the behavior of objects.

KEY TERMS

AVL tree. A self-adjusting binary tree structure that can guarantee good access times for data in memory.

B-tree. A tree structure that provides fast access to data stored in files. Unlike binary trees, in which the branching factor from a node of the tree is two, the descendants from a node of a B-tree can be a much larger number. We introduce B-trees in Chapter 9.

B$^+$ tree. A variation on the B-tree structure that provides sequential access to the data as well as fast-indexed access. We discuss B$^+$ trees at length in Chapter 10.

Class. The specification of the common data attributes (members) and functions (methods) of a collection of objects.

Constructor. A function that initializes an object when it is created. C++ automatically adds a call to a constructor for each operation that creates an object.

Extendible hashing. An approach to hashing that works well with files that over time undergo substantial changes in size.

File structures. The organization of data on secondary storage devices such as disks.

Hashing. An access mechanism that transforms the search key into a storage address, thereby providing very fast access to stored data.

Member. An attribute of an object that is included in a class specification. Members are either data fields or functions (methods).

Method. A function member of an object. Methods are included in class specifications.

Overloaded symbol. An operator or identifier in a program that has more than one meaning. The context of the use of the symbol determines its meaning.

Private. The most restrictive access control level in C++. Private names can be used only by member functions of the class.

Public. The least restrictive access control level in C++. Public names can be used in any function.

Sequential access. Access that takes records in order, looking at the first, then the next, and so on.

FURTHER READINGS

There are many good introductory textbooks on C++ and object-oriented programming, including Berry (1997), Friedman and Koffman (1994), and Sessions (1992). The second edition of Stroustrup's book on C++ (1998) is the standard reference for the language. The third edition of Stroustrup (1997) is a presentation of the Draft Standard for C++ 3.0.

PROGRAMMING PROJECT

This is the first part of an object-oriented programming project that continues throughout the book. Each part extends the project with new file structures. We begin by introducing two classes of data objects. These projects apply the concepts of the book to produce an information system that maintains and processes information about students and courses.

1. Design a class `Student`. Each object represents information about a single student. Members should be included for identifier, name, address, date of first enrollment, and number of credit hours completed. Methods should be included for intitalization (constructors), assignment (overloaded "=" operator), and modifying field values, including a method to increment the number of credit hours.
2. Design a class `CourseRegistration`. Each object represents the enrollment of a student in a course. Members should be included for a course identifier, student identifier, number of credit hours, and course grade. Methods should be included as appropriate.
3. Create a list of student and course registration information. This information will be used in subsequent exercises to test and evaluate the capabilities of the programming project.

The next part of the programming project is in Chapter 2.

CHAPTER 2

Fundamental File Processing Operations

CHAPTER OBJECTIVES

- Describe the process of linking a *logical file* within a program to an actual *physical file* or device.
- Describe the procedures used to create, open, and close files.
- Introduce the C++ input and output classes.
- Explain the use of overloading in C++.
- Describe the procedures used for reading from and writing to files.
- Introduce the concept of *position* within a file and describe procedures for *seeking* different positions.
- Provide an introduction to the organization of hierarchical file systems.
- Present the Unix view of a file and describe Unix file operations and commands based on this view.

CHAPTER OUTLINE

2.1 Physical Files and Logical Files
2.2 Opening Files
2.3 Closing Files
2.4 Reading and Writing
2.4.1 Read and Write Functions
2.4.2 Files with C Streams and C++ Stream Classes
2.4.3 Programs in C++ to Display the Contents of a File
2.4.4 Detecting End-of-File
2.5 Seeking
2.5.1 Seeking with C Streams
2.5.2 Seeking with C++ Stream Classes
2.6 Special Characters in Files
2.7 The Unix Directory Structure
2.8 Physical Devices and Logical Files
2.8.1 Physical Devices as Files
2.8.2 The Console, the Keyboard, and Standard Error
2.8.3 I/O Redirection and Pipes
2.9 File-Related Header Files
2.10 Unix File System Commands

2.1 Physical Files and Logical Files

When we talk about a file on a disk or tape, we refer to a particular collection of bytes stored there. A file, when the word is used in this sense, physically exists. A disk drive might contain hundreds, even thousands, of these *physical files.*

From the standpoint of an application program, the notion of a file is different. To the program, a file is somewhat like a telephone line connected to a telephone network. The program can receive bytes through this phone line or send bytes down it, but it knows nothing about where these bytes come from or where they go. The program knows only about its own end of the phone line. Moreover, even though there may be thousands of physical files on a disk, a single program is usually limited to the use of only about twenty files.

The application program relies on the operating system to take care of the details of the telephone switching system, as illustrated in Fig. 2.1. It could be that bytes coming down the line into the program originate from a physical file or that they come from the keyboard or some other input device. Similarly, the bytes the program sends down the line might end up in a file, or they could appear on the terminal screen. Although the program often doesn't know where bytes are coming from or where they are going, it does know which line it is using. This line is usually referred to as the *logical file* to distinguish it from the *physical files* on the disk or tape.

Before the program can open a file for use, the operating system must receive instructions about making a hookup between a logical file (for example, a phone line) and some physical file or device. When using operating systems such as IBM's OS/MVS, these instructions are provided through job control language (JCL). On minicomputers and microcomputers, more modern operating systems such as Unix, MS-DOS, and VMS provide the instructions within the program. For example, in Cobol,[1] the association between a logical file called `inp_file` and a physical file called `myfile.dat` is made with the following statement:

```
select inp_file assign to "myfile.dat".
```

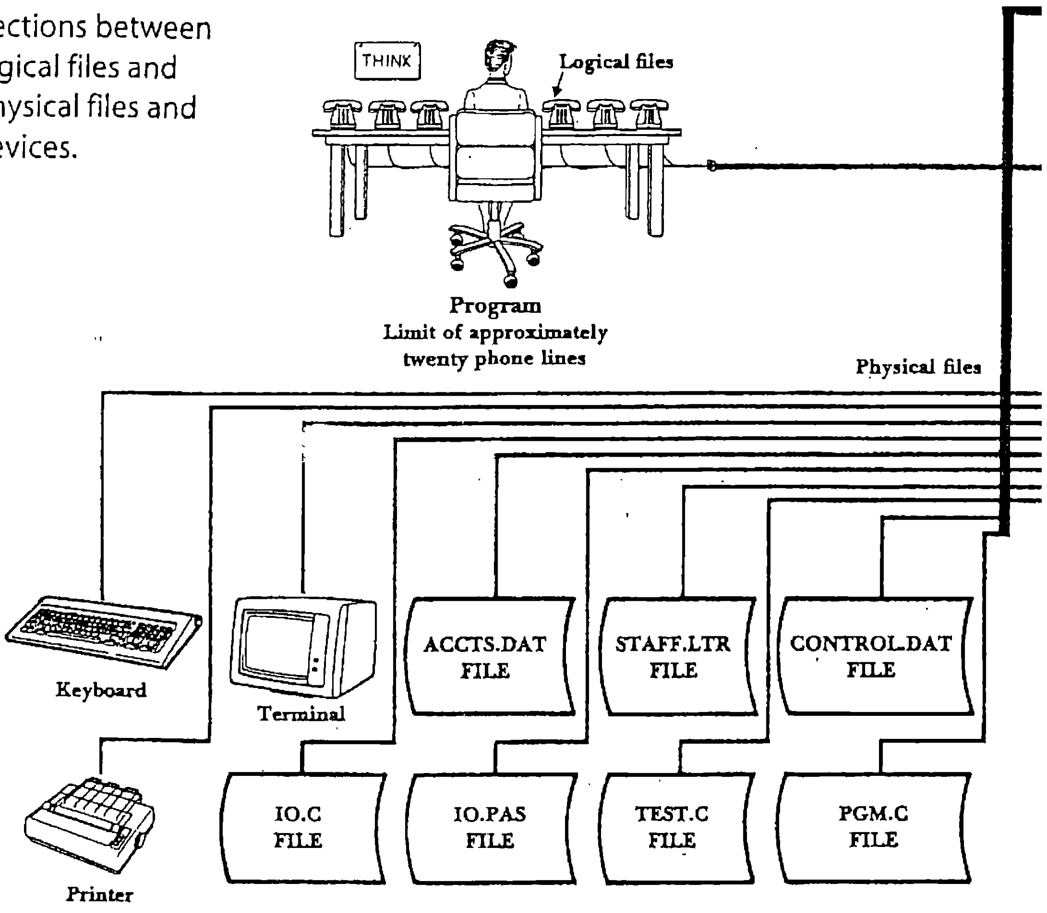
This statement asks the operating system to find the physical file named `myfile.dat` and then to make the hookup by assigning a logical file (phone line) to it. The number identifying the particular phone line that is assigned is returned through the variable `inp_file`, which is the file's *logical name*. This logical name is what we use to refer to the file inside the program. Again, the telephone analogy applies: My office phone is connected to six telephone lines. When I receive a call I get an intercom message such as, "You have a call on line three." The receptionist does not say, "You have a call from 918-123-4567." I need to have the call identified *logically*, not *physically*.

2.2 Opening Files

Once we have a logical file identifier hooked up to a physical file or device, we need to declare what we intend to do with the file. In general, we have two options: (1) open an *existing* file, or (2) create a *new* file, deleting any existing contents in the physical file. Opening a file makes it ready for use by the program. We are positioned at the beginning of the file and are

1. These values are defined in an "include" file packaged with your Unix system or C compiler. The name of the include file is often `fcntl.h` or `file.h`, but it can vary from system to system.

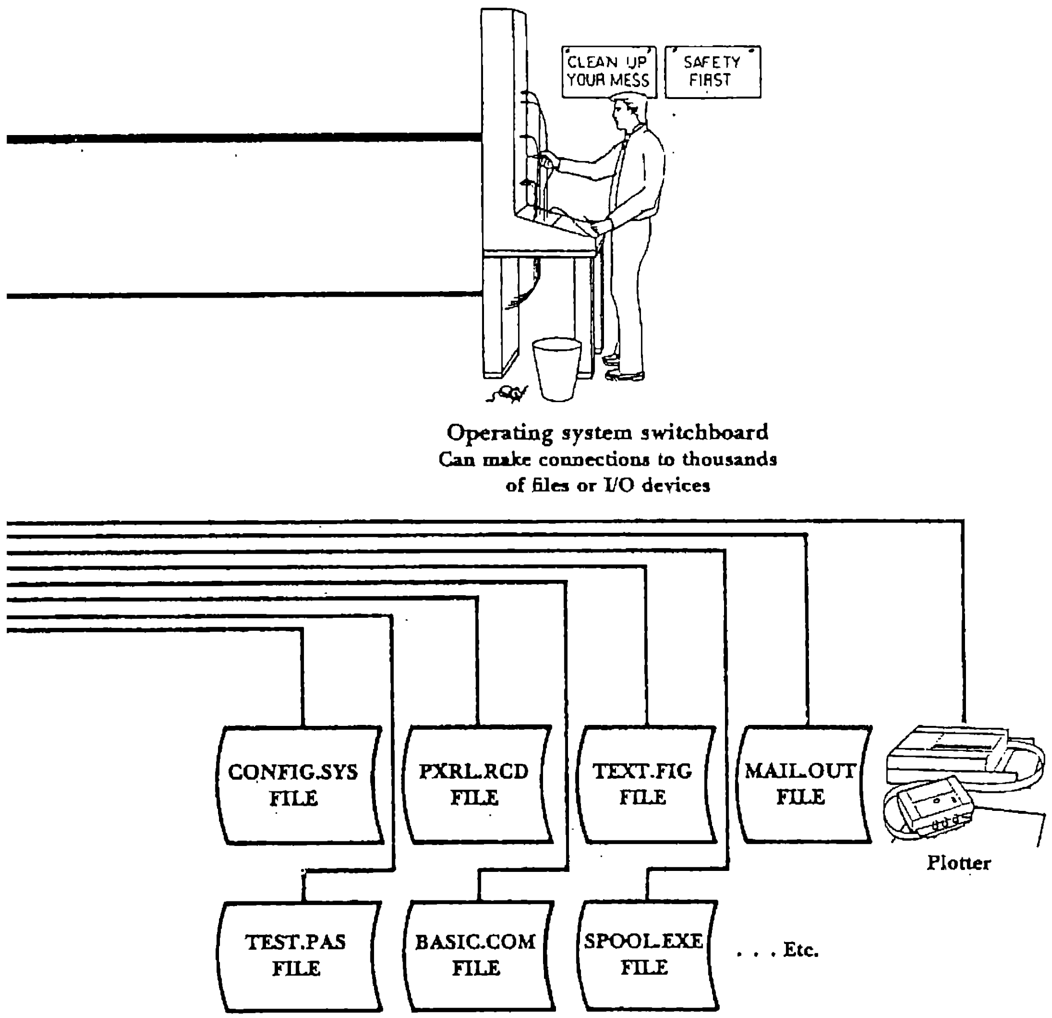
Figure 2.1 The program relies on the operating system to make connections between logical files and physical files and devices.



ready to start reading or writing. The file contents are not disturbed by the open statement. Creating a file also opens the file in the sense that it is ready for use after creation. Because a newly created file has no contents, writing is initially the only use that makes sense.

As an example of opening an existing file or creating a new one in C and C++, consider the function `open`, as defined in header file `fcntl.h`. Although this function is based on a Unix system function, many C++ implementations for MS-DOS and Windows, including Microsoft Visual C++, also support `open` and the other parts of `fcntl.h`. This function takes two required arguments and a third argument that is optional:

```
fd = open(filename, flags [, pmode]);
```



The return value `fd` and the arguments `filename`, `flags`, and `pmode` have the following meanings:

Argument	Type	Explanation
`fd`	`int`	The file descriptor. Using our earlier analogy, this is the phone line (logical file identifier) used to refer to the file within the program. It is an integer. If there is an error in the attempt to open the file, this value is negative.
`filename`	`char *`	A character string containing the physical file name. (Later we discuss pathnames that include directory information about the file's location. This argument can be a pathname.)

(continued)

Argument	Type	Explanation
`flags`	`int`	The flags argument controls the operation of the open function, determining whether it opens an existing file for reading or writing. It can also be used to indicate that you want to create a new file or open an existing file but delete its contents. The value of flags is set by performing a bit-wise OR of the following values, among others.
		`O_APPEND` Append every write operation to the end of the file.
		`O_CREAT` Create and open a file for writing. This has no effect if the file already exists.
		`O_EXCL` Return an error if O_CREATE is specified and the file exists.
		`O_RDONLY` Open a file for reading only.
		`O_RDWR` Open a file for reading and writing.
		`O_TRUNC` If the file exists, truncate it to a length of zero, destroying its contents.
		`O_WRONLY` Open a file for writing only.
		Some of these flags cannot be used in combination with one another. Consult your documentation for details and for other options.
`pmode`	`int`	If O_CREAT is specified, pmode is required. This integer argument specifies the protection mode for the file. In Unix, the pmode is a three-digit octal number that indicates how the file can be used by the owner (first digit), by members of the owner's group (second digit), and by everyone else (third digit). The first bit of each octal digit indicates read permission, the second write permission, and the third execute permission. So, if pmode is the octal number 0751, the file's owner has read, write, and execute permission for the file; the owner's group has read and execute permission; and everyone else has only execute permission:

```
                 r w e   r w e   r w e
pmode = 0751  =  1 1 1   1 0 1   0 0 1
                 owner   group   world
```

Given this description of the open function, we can develop some examples to show how it can be used to open and create files in C. The following function call opens an existing file for reading and writing or creates a new one if necessary. If the file exists, it is opened without change; reading or writing would start at the file's first byte.

```
fd = open(filename, O_RDWR | O_CREAT, 0751);
```

The following call creates a new file for reading and writing. If there is already a file with the name specified in `filename`, its contents are truncated.

```
fd = open(filename, O_RDWR | O_CREAT | O_TRUNC, 0751);
```

Finally, here is a call that will create a new file only if there is not already a file with the name specified in `filename`. If a file with this name exists, it is not opened, and the function returns a negative value to indicate an error.

```
fd = open(filename, O_RDWR | O_CREAT | O_EXCL, 0751);
```

File protection is tied more to the host operating system than to a specific language. For example, implementations of C running on systems that support file protection, such as VAX/VMS, often include extensions to standard C that let you associate a protection status with a file when you create it.

2.3 Closing Files

In terms of our telephone line analogy, closing a file is like hanging up the phone. When you hang up the phone, the phone line is available for taking or placing another call; when you close a file, the logical file name or file descriptor is available for use with another file. Closing a file that has been used for output also ensures that everything has been written to the file. As you will learn in a later chapter, it is more efficient to move data to and from secondary storage in blocks than it is to move data one byte at a time. Consequently, the operating system does not immediately send off the bytes we write but saves them up in a buffer for transfer as a block of data. Closing a file ensures that the buffer for that file has been flushed of data and that everything we have written has been sent to the file.

Files are usually closed automatically by the operating system when a program terminates normally. Consequently, the execution of a close statement within a program is needed only to protect it against data loss in the event that the program is interrupted and to free up logical filenames for reuse.

Now that you know how to connect and disconnect programs to and from physical files and how to open the files, you are ready to start sending and receiving data.

2.4 Reading and Writing

Reading and *writing* are fundamental to file processing; they are the actions that make file processing an *input/output* (I/O) operation. The form of the read and write statements used in different languages varies. Some languages provide very high-level access to reading and writing and automatically take care of details for the programmer. Other languages provide access at a much lower level. Our use of C and C++ allows us to explore some of these differences.[2]

2.4.1 Read and Write Functions

We begin with reading and writing at a relatively low level. It is useful to have a kind of systems-level understanding of what happens when we send and receive information to and from a file.

A low-level read call requires three pieces of information, expressed here as arguments to a generic `Read` function:

```
Read (Source_file, Destination_addr, Size)
```

`Source_file`	The `Read` call must know where it is to read from. We specify the source by logical file name (phone line) through which data is received. (Remember, before we do any reading, we must have already opened the file so the connection between a logical file and a specific physical file or device exists.)
`Destination_addr`	`Read` must know where to place the information it reads from the input file. In this generic function we specify the destination by giving the first address of the memory block where we want to store the data.
`Size`	Finally, `Read` must know how much information to bring in from the file. Here the argument is supplied as a byte count.

2. To accentuate the differences and view I/O operations at something close to a systems level, we use the `fread` and `fwrite` functions in C rather than the higher-level functions such as `fgetc`, `fgets`, and so on.

A `Write` statement is similar; the only difference is that the data moves in the other direction:

```
Write(Destination_file, Source_addr, Size)
```

`Destination_file`	The logical file name that is used for sending the data.
`Source_addr`	`Write` must know where to find the information it will send. We provide this specification as the first address of the memory block where the data is stored.
`Size`	The number of bytes to be written must be supplied.

2.4.2 Files with C Streams and C++ Stream Classes

I/O operations in C and C++ are based on the concept of a stream, which can be a file or some other source or consumer of data. There are two different styles for manipulating files in C++. The first uses the standard C functions defined in header file `stdio.h`. This is often referred to as *C streams* or *C input/output.* The second uses the stream classes of header files `iostream.h` and `fstream.h`. We refer to this style as *C++ stream classes.*

The header file `stdio.h` contains definitions of the types and the operations defined on C streams. The standard input and output of a C program are streams called `stdin` and `stdout`, respectively. Other files can be associated with streams through the use of the `fopen` function:

```
file = fopen (filename, type);
```

The return value `file` and the arguments `filename` and `type` have the following meanings:

Argument	Type	Explanation
`file`	`FILE *`	A pointer to the file descriptor. Type `FILE` is another name for `struct _iobuf`. If there is an error in the attempt to open the file, this value is null, and the variable `errno` is set with the error number.
`filename`	`char *`	The file name, just as in the Unix open function.
`type`	`char *`	The `type` argument controls the operation of the open function, much like the flags argument to open. The following values are supported:

`"r"`	Open an existing file for input.
`"w"`	Create a new file, or truncate an existing one, for output.

`"a"`	Create a new file, or append to an existing one, for output.
`"r+"`	Open an existing file for input and output.
`"w+"`	Create a new file, or truncate an existing one, for input and output.
`"a+"`	Create a new file, or append to an existing one, for input and output.

Read and write operations are supported by functions `fread`, `fget`, `fwrite`, and `fput`. Functions `fscanf` and `fprintf` are used for formatted input and output.

Stream classes in C++ support open, close, read, and write operations that are equivalent to those in `stdio.h`, but the syntax is considerably different. Predefined stream objects `cin` and `cout` represent the standard input and standard output files. The main class for access to files, `fstream`, as defined in header files `iostream.h` and `fstream.h`, has two constructors and a wide variety of methods. The following constructors and methods are included in the class:

```
fstream (); // leave the stream unopened
fstream (char * filename, int mode);
int open (char * filename, int mode);
int read (unsigned char * dest_addr, int size);
int write (unsigned char * source_addr, int size);
```

The argument `filename` of the second constructor and the method `open` are just as we've seen before. These two operations attach the `fstream` to a file. The value of `mode` controls the way the file is opened, like the `flags` and `type` arguments previously described. The value is set with a bit-wise or of constants defined in class `ios`. Among the options are `ios::in` (input), `ios::out` (output), `ios::nocreate` (fail if the file does not exist), and `ios::noreplace` (fail if the file does exist). One additional, nonstandard option, `ios::binary`, is supported on many systems to specify that a file is binary. On MS-DOS systems, if `ios::binary` is not specified, the file is treated as a text file. This can have some unintended consequences, as we will see later.

A large number of functions are provided for formatted input and output. The overloading capabilities of C++ are used to make sure that objects are formatted according to their types. The infix operators >>(extraction) and <<(insertion) are overloaded for input and output, respectively. The header file `iostream.h` includes the following overloaded definitions of the insertion operator (and many others):

```
ostream& operator<<(char c);
ostream& operator<<(unsigned char c);
ostream& operator<<(signed char c);
ostream& operator<<(const char *s);
ostream& operator<<(const unsigned char *s);
ostream& operator<<(const signed char *s);
ostream& operator<<(const void *p);
ostream& operator<<(int n);
ostream& operator<<(unsigned int n);
ostream& operator<<(long n);
ostream& operator<<(unsigned long n);
```

The overloading resolution rules of C++ specify which function is selected for a particular call depending on the types of the actual arguments and the types of the formal parameters. In this case, the insertion function that is used to evaluate an expression depends on the type of the arguments, particularly the right argument. Consider the following statements that include insertions into `cout` (an object of class `ostream`):

```
int n = 25;
cout << "Value of n is "<< n << endl;
```

The insertion operators are evaluated left to right, and each one returns its left argument as the result. Hence, the stream `cout` has first the string "Value of n is " inserted, using the fourth function in the list above, then the decimal value of `n`, using the eighth function in the list. The last operand is the I/O manipulator `endl`, which causes an end-of-line to be inserted. The insertion function that is used for `<< endl` is not in the list above. The header file `iostream.h` includes the definition of `endl` and the operator that is used for this insertion.

Appendix C includes definitions and examples of many of the formatted input and output operations.

2.4.3 Programs in C++ to Display the Contents of a File

Let's do some reading and writing to see how these functions are used. This first simple file processing program opens a file for input and reads it, character by character, sending each character to the screen after it is read from the file. This program includes the following steps:

1. Display a prompt for the name of the input file.
2. Read the user's response from the keyboard into a variable called `filename`.

3. Open the file for input.
4. While there are still characters to be read from the input file,
   a. read a character from the file;
   b. write the character to the terminal screen.
5. Close the input file.

Figures 2.2 and 2.3 are C++ implementations of this program using C streams and C++ stream classes, respectively. It is instructive to look at the differences between these implementations. The full implementations of these programs are included in Appendix D.

Steps 1 and 2 of the program involve writing and reading, but in each of the implementations this is accomplished through the usual functions for handling the screen and keyboard. Step 4a, in which we read from the input file, is the first instance of actual file I/O. Note that the `fread` call using C streams parallels the low-level, generic `Read` statement we described earlier; in truth, we used the `fread` function as the model for our low-level `Read`. The function's first argument gives the *address* of a character variable used as the *destination* for the data, the second and third arguments are the element size and the number of elements (in this case the size is 1 byte, and the number of elements is one), and the fourth argument gives a pointer to the file descriptor (the C stream version of a logical file name) as the *source* for the input.

```
// listc.cpp
// program using C streams to read characters from a file
// and write them to the terminal screen
#include <stdio.h>
main( ) {
  char ch;
  FILE * file; // pointer to file descriptor
  char filename[20];
  printf("Enter the name of the file: ");    // Step 1
  gets(filename);                            // Step 2
  file =fopen(filename, "r");                // Step 3
  while (fread(&ch, 1, 1, file) != 0)        // Step 4a
    fwrite(&ch, 1, 1, stdout);               // Step 4b
  fclose(file);                              // Step 5
}
```

Figure 2.2 The file listing program using C streams (`listc.cpp`).

```
// listcpp.cpp
// list contents of file using C++ stream classes
#include <fstream.h>
main () {
  char ch;
  fstream file; // declare unattached fstream
  char filename[20];
  cout <<"Enter the name of the file: " // Step 1
    <<flush; // force output
  cin >> filename;                      // Step 2
  file . open(filename, ios::in);       // Step 3
  file . unsetf(ios::skipws);// include white space in read
  while (1)
  {
    file >> ch;                         // Step 4a
    if (file.fail()) break;
    cout << ch;                         // Step 4b
  }
  file . close();                       // Step 5
}
```

Figure 2.3 The file listing program using C++ stream classes (`listcpp.cpp`).

The arguments for the call to operator >> communicate the same information at a higher level. The first argument is the logical file name for the input source. The second argument is the name of a character variable, which is interpreted as the address of the variable. The overloading resolution selects the >> operator whose right argument is a `char` variable. Hence, the code *implies* that only a single byte is to be transferred. In the C++ version, the call `file.unsetf(ios::skipws)` causes operator >> to include white space (blanks, end-of-line, tabs, and so on). The default for formatted `read` with C++ stream classes is to skip white space.

After a character is read, we write it to standard output in Step 4b. Once again the differences between C streams and C++ stream classes indicate the range of approaches to I/O used in different languages. Everything must be stated explicitly in the `fwrite` call. Using the special assigned file descriptor of `stdout` to identify the terminal screen as the destination for our writing,

```
fwrite(&ch, 1, 1, stdout);
```

means: "Write to standard output the contents from memory starting at the address `&ch`. Write only one element of one byte." Beginning C++ programmers should pay special attention to the use of the `&` symbol in the `fwrite` call here. This particular call, as a very low-level call, requires that the programmer provide the starting *address* in memory of the bytes to be transferred.

`Stdout`, which stands for "standard output," is a pointer to a struct defined in the file `stdio.h`, which has been included at the top of the program. The concept of standard output and its counterpart standard input are covered later in Section 2.8 "Physical and Logical Files."

Again the C++ stream code operates at a higher level. The right operand of operator << is a character value. Hence a single byte is transferred to `cout`.

```
cout << ch;
```

As in the call to operator >>, C++ takes care of finding the *address* of the bytes; the programmer need specify only the name of the variable `ch` that is associated with that address.

2.4.4 Detecting End-of-File

The programs in Figs. 2.2 and 2.3 have to know when to end the `while` loop and stop reading characters. C streams and C++ streams signal the end-of-file condition differently. The function `fread` returns a value that indicates whether the read succeeded. However, an explicit test is required to see if the C++ stream `read` has failed.

The `fread` call returns the number of elements read as its value In this case, if `fread` returns a value of zero, the program has reached the end of the file. So we construct the `while` loop to run as long as the `fread` call finds something to read.

Each C++ stream has a state that can be queried with function calls. Figure 2.3 uses the function `fail`, which returns true (1) if the previous operation on the stream failed. In this case, `file.fail()` returns false if the previous read failed because of trying to read past end-of-file. The following statement exits the `while` loop when end-of-file is encountered:

```
if (file.fail()) break;
```

In some languages, including Ada, a function `end_of_file` can be used to test for end-of-file. As we read from a file, the operating system keeps track of our location in the file with a *read/write pointer*. This is necessary: when the next byte is read, the system knows where to get it. The `end_of_file` function queries the system to see whether the read/write pointer has moved past the last element in the file. If it has, `end_of_file` returns true; otherwise it returns false. In Ada, it is necessary to call `end_of_file` before trying to read the next byte. For an empty file, `end_of_file` immediately returns `true`, and no bytes can be read.

2.5 Seeking

In the preceding sample programs we read through the file *sequentially*, reading one byte after another until we reach the end of the file. Every time a byte is read, the operating system moves the read/write pointer ahead, and we are ready to read the next byte.

Sometimes we want to read or write without taking the time to go through every byte sequentially. Perhaps we know that the next piece of information we need is ten thousand bytes away, so we want to jump there. Or perhaps we need to jump to the end of the file so we can add new information there. To satisfy these needs we must be able to control the movement of the read/write pointer.

The action of moving directly to a certain position in a file is often called *seeking*. A *seek* requires at least two pieces of information, expressed here as arguments to the generic pseudocode function `Seek`:

```
Seek(Source_file, Offset)
```

`Source_file`	The logical file name in which the seek will occur.
`Offset`	The number of positions in the file the pointer is to be moved from the start of the file.

Now, if we want to move directly from the origin to the 373d position in a file called `data`, we don't have to move sequentially through the first 372 positions. Instead, we can say

```
Seek(data, 373)
```

2.5.1 Seeking with C Streams

One of the features of Unix that has been incorporated into C streams is the ability to view a file as a potentially very large *array of bytes* that just happens to be kept on secondary storage. In an array of bytes in memory, we can move to any particular byte using a subscript. The C stream seek function, `fseek`, provides a similar capability for files. It lets us set the read/write pointer to any byte in a file.

The `fseek` function has the following form:

```
pos = fseek(file, byte_offset, origin)
```

where the variables have the following meanings:

`pos`	A long integer value returned by `fseek` equal to the position (in bytes) of the read/write pointer after it has been moved.
`file`	The file descriptor of the file to which the `fseek` is to be applied.
`byte_offset`	The number of bytes to move from some origin in the file. The byte offset must be specified as a long integer, hence the name `fseek` for long seek. When appropriate, the `byte_offset` can be negative.
`origin`	A value that specifies the starting position from which the `byte_offset` is to be taken. The origin can have the value 0, 1, or 2[3] 0–fseek from the beginning of the file; 1–fseek from the current position; 2–fseek from the end of the file.

The following definitions are included in `stdio.h` to allow symbolic reference to the origin values.

```
#define SEEK_SET     0
#define SEEK_CUR     1
#define SEEK_END     2
```

The following program fragment shows how you could use `fseek` to move to a position that is 373 bytes into a file.

```
long pos;
fseek(File * file, long offset, int origin);
File * file;
.
.
pos=fseek(file, 373L, 0);
```

3. Although the values 0, 1, and 2 are almost always used here, they are not guaranteed to work for all C implementations. Consult your documentation.

2.5.2 Seeking with C++ Stream Classes

Seeking in C++ stream classes is almost exactly the same as it is in C streams. There are two mostly syntactic differences:

- An object of type `fstream` has two file pointers: a get pointer for input and a put pointer for output. Two functions are supplied for seeking: `seekg` which moves the get pointer, and `seekp` which moves the put pointer. It is not guaranteed that the pointers move separately, but they might. We have to be very careful in our use of these seek functions and often call both functions together.
- The seek operations are methods of the stream classes. Hence the syntax is `file.seekg(byte_offset,origin)` and `file.seekp(byte_offset,origin)`. The value of origin comes from class ios, which is described in more detail in Chapter 4. The values are ios::beg (beginning of file), ios::cur (current position), and ios::end (end of file).

The following moves both get and put pointers to a byte 373:

```
file.seekg(373, ios::beg);
file.seekp(373, ios::beg);
```

2.6 Special Characters in Files

As you create the file structures described in this text, you may encounter some difficulty with extra, unexpected characters that turn up in your files with characters that disappear and with numeric counts that are inserted into your files. Here are some examples of the kinds of things you might encounter:

- On many computers you may find that a Control-Z (ASCII value of 26) is appended at the end of your files. Some applications use this to indicate end-of-file even if you have not placed it there. This is most likely to happen on MS-DOS systems.
- Some systems adopt a convention of indicating end-of-line in a text file[4] as a pair of characters consisting of a carriage return (CR: ASCII

4. When we use the term "text file" in this text, we are referring to a file consisting entirely of characters from a specific standard character set, such as ASCII or EBCDIC. Unless otherwise specified, the ASCII character set will be assumed. Appendix B contains a table that describes the ASCII character set.

value of 13) and a line feed (LF: ASCII value of 10). Sometimes I/O procedures written for such systems automatically expand single CR characters or LF characters into CR-LF pairs. This unrequested addition of characters can cause a great deal of difficulty. Again, you are most likely to encounter this phenomenon on MS-DOS systems. Using flag "b" in a C file or mode ios::bin in a C++ stream will suppress these changes.

- Users of larger systems, such as VMS, may find that they have just the opposite problem. Certain file formats under VMS *remove* carriage return characters from your file without asking you, replacing them with a *count* of the characters in what the system has perceived as a line of text.

These are just a few examples of the kinds of uninvited modifications that record management systems or that I/O support packages might make to your files. You will find that they are usually associated with the concepts of a line of text or the end of a file. In general, these modifications to your files are an attempt to make your life easier by doing things for you automatically. This might, in fact, work out for those who want to do nothing more than store some text in a file. Unfortunately, however, programmers building sophisticated file structures must sometimes spend a lot of time finding ways to disable this automatic assistance so they can have complete control over what they are building. Forewarned is forearmed: readers who encounter these kinds of difficulties as they build the file structures described in this text can take some comfort from the knowledge that the experience they gain in disabling automatic assistance will serve them well, over and over, in the future.

2.7 The Unix Directory Structure

No matter what computer system you have, even if it is a small PC, chances are there are hundreds or even thousands of files you have access to. To provide convenient access to such large numbers of files, your computer has some method for organizing its files. In Unix this is called the *file system*.

The Unix file system is a tree-structured organization of *directories*, with the *root* of the tree signified by the character /. All directories, including the root, can contain two kinds of files: regular files with programs and

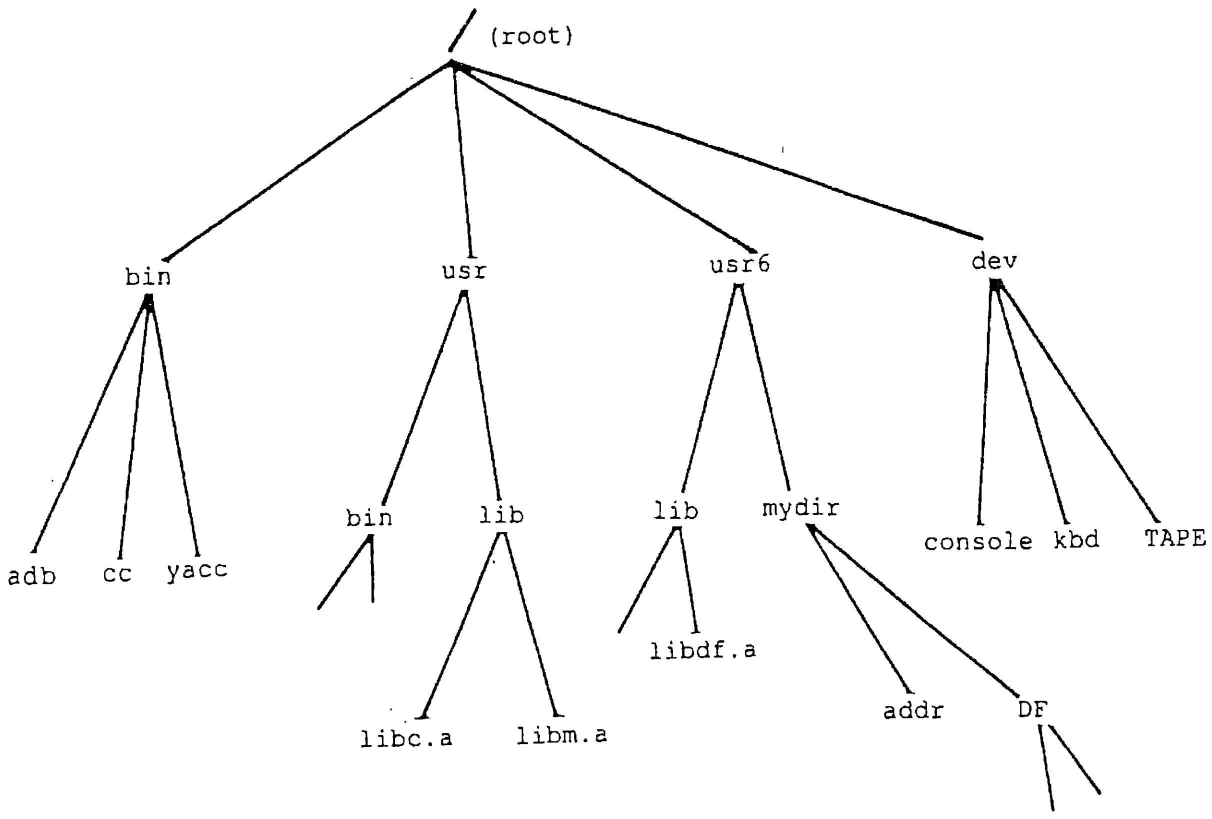


Figure 2.4 Sample Unix directory structure.

data, and directories (Fig. 2.4). Since devices such as tape drives are also treated like files in Unix, directories can also contain references to devices, as shown in the `dev` directory in Fig. 2.4. The file name stored in a Unix directory corresponds to what we call its physical name.

Since every file in a Unix system is part of the file system that begins with the root, any file can be uniquely identified by giving its *absolute pathname*. For instance, the true, unambiguous name of the file "addr" in Fig. 2.4 is `/usr6/mydir/addr`. (Note that the / is used both to indicate the root directory and to separate directory names from the file name.)

When you issue commands to a Unix system, you do so within a directory, which is called your *current directory*. A pathname for a file that does not begin with a / describes the location of a file relative to the current directory. Hence, if your current directory in Fig. 2.4 is `mydir`, `addr` uniquely identifies the file `/usr6/mydir/addr`.

The special filename . stands for the current directory, and .. stands for the parent of the current directory. Hence, if your current directory is `/usr6/mydir/DF`, `../addr` refers to the file `/usr6/mydir/addr`.

2.8 Physical Devices and Logical Files

2.8.1 Physical Devices as Files

One of the most powerful ideas in Unix is reflected in its notion of what a file is. In Unix, a file is a sequence of bytes without any implication of how or where the bytes are stored or where they originate. This simple conceptual view of a file makes it possible to do with very few operations what might require several times as many operations on a different operating system. For example, it is easy to think of a magnetic disk as the source of a file because we are used to the idea of storing such things on disks. But in Unix, devices like the keyboard and the console are also files—in Fig. 2.4, `/dev/kbd` and `/dev/console`, respectively. The keyboard produces a sequence of bytes that are sent to the computer when keys are pressed; the console accepts a sequence of bytes and displays their corresponding symbols on a screen.

How can we say that the Unix concept of a file is simple when it allows so many different physical things to be called files? Doesn't this make the situation more complicated, not less so? The trick in Unix is that no matter what physical representation a file may take, the logical view of a Unix file is the same. In its simplest form, a Unix file is represented logically by an integer–the file descriptor. This integer is an index to an array of more complete information about the file. A keyboard, a disk file, and a magnetic tape are all represented by integers. Once the integer that describes a file is identified, a program can access that file. If it knows the logical name of a file, a program can access that file without knowing whether the file comes from a disk, a tape, or a connection to another computer.

Although the above discussion is directed at Unix files, the same capability is available through the `stdio` functions `fopen`, `fread`, and so on. Similar capabilities are present in MS-DOS, Windows, and other operating systems.

2.8.2 The Console, the Keyboard, and Standard Error

We see an example of the duality between devices and files in the listc.cpp program in Fig. 2.2:

```
file =fopen(filename, "r");                   // Step 3
while (fread(&ch, 1, 1, file) != 0)           // Step 4a
  fwrite(&ch, 1, 1, stdout);                  // Step 4b
```

The logical file is represented by the value returned by the `fopen` call. We assign this integer to the variable `file` in Step 3. In Step 4b, we use the value `stdout`, defined in `stdio.h`, to identify the console as the file to be written to.

There are two other files that correspond to specific physical devices in most implementations of C streams: the keyboard is called `stdin` *(standard input),* and the error file is called `stderr` *(standard error).* Hence, `stdin` is the keyboard on your terminal. The statement

```
fread(&ch, 1, 1, stdin);
```

reads a single character from your terminal. `Stderr` is an error file which, like `stdout`, is usually just your console. When your compiler detects an error, it generally writes the error message to this file, which normally means that the error message turns up on your screen. As with `stdin`, the values `stdin` and `stderr` are usually defined in `stdio.h`.

Steps 1 and 2 of the file listing program also involve reading and writing from `stdin` or `stdout`. Since an enormous amount of I/O involves these devices, most programming languages have special functions to perform console input and output—in list.cpp, the C functions `printf` and `gets` are used. Ultimately, however, `printf` and `gets` send their output through `stdout` and `stdin`, respectively. But these statements hide important elements of the I/O process. For our purposes, the second set of read and write statements is more interesting and instructive.

2.8.3 I/O Redirection and Pipes

Suppose you would like to change the file listing program so it writes its output to a regular file rather than to `stdout`. Or suppose you wanted to use the output of the file listing program as input to another program. Because it is common to want to do both of these, operating systems provide convenient shortcuts for switching between standard I/O (`stdin` and `stdout`) and regular file I/O. These shortcuts are called *I/O redirection* and *pipes.*[5]

I/O redirection lets you specify at execution time alternate files for input or output. The notations for input and output redirection on the command line in Unix are

```
< file                  (redirect stdin to "file")
> file                  (redirect stdout to "file")
```

5. Strictly speaking, I/O redirection and pipes are part of a Unix shell, which is the command interpreter that sits on top of the core Unix operating system, the kernel. For the purpose of this discussion, this distinction is not important.

For example, if the executable file listing program is called "list.exe," we redirect the output from `stdout` to a file called "myfile" by entering the line

```
list.exe > myfile
```

What if, instead of storing the output from the list program in a file, you wanted to use it immediately in another program to sort the results? Pipes let you do this. The notation for a pipe in Unix and in MS-DOS is `|`. Hence,

```
program1 | program2
```

means take any `stdout` output from program1 and use it in place of any `stdin` input to program2. Because Unix has a special program called `sort`, which takes its input from `stdin`, you can sort the output from the list program, without using an intermediate file, by entering

```
list | sort
```

Since `sort` writes its output to `stdout`, the sorted listing appears on your terminal screen unless you use additional pipes or redirection to send it elsewhere.

2.9 File-Related Header Files

Unix, like all operating systems, has special names and values that you must use when performing file operations. For example, some C functions return a special value indicating end-of-file (EOF) when you try to read beyond the end of a file.

Recall the flags that you use in an open call to indicate whether you want read-only, write-only, or read/write access. Unless we know just where to look, it is often not easy to find where these values are defined. Unix handles the problem by putting such definitions in special header files such as `/usr/include`, which can be found in special directories.

Header files relevant to the material in this chapter are `stdio.h`, `iostream.h`, `fstream.h`, `fcntl.h`, and `file.h`. The C streams are in `stdio.h`; C++ streams in `iostream.h` and `fstream.h`. Many Unix operations are in `fcntl.h` and `file.h`. EOF, for instance, is defined on many Unix and MS-DOS systems in `stdio.h`, as are the file pointers `stdin`, `stdout`, and `stderr`. And the flags O_RDONLY, O_WRONLY, and O_RDWR can usually be found in `file.h` or possibly in one of the files that it includes.

It would be instructive for you to browse through these files as well as others that pique your curiosity.

2.10 Unix File System Commands

Unix provides many commands for manipulating files. We list a few that are relevant to the material in this chapter. Most of them have many options, but the simplest uses of most should be obvious. Consult a Unix manual for more information on how to use them.

`cat` *filenames*	Print the contents of the named text files.
`tail` *filename*	Print the last ten lines of the text file.
`cp` *file1 file2*	Copy file1 to file2.
`mv` *file1 file2*	Move (rename) file1 to file2.
`rm` *filenames*	Remove (delete) the named files.
`chmod` *mode filename*	Change the protection mode on the named files.
`ls`	List the contents of the directory.
`mkdir` *name*	Create a directory with the given name.
`rmdir` *name*	Remove the named directory.

SUMMARY

This chapter introduces the fundamental operations of file systems: Open, Create, Close, Read, Write, and Seek. Each of these operations involves the creation or use of a link between a *physical file* stored on a secondary device and a *logical file* that represents a program's more abstract view of the same file. When the program describes an operation using the *logical file name,* the equivalent physical operation gets performed on the corresponding physical file.

The six operations appear in programming languages in many different forms. Sometimes they are built-in commands, sometimes they are functions, and sometimes they are direct calls to an operating system.

Before we can use a physical file, we must link it to a logical file. In some programming environments, we do this with a statement (`select/assign` in Cobol) or with instructions outside of the program (operating system shell scripts). In other languages, the link between the physical file and a logical file is made with open or create.

The operations create and open make files ready for reading or writing. Create causes a new physical file to be created. Open operates on an already existing physical file, usually setting the read/write pointer to the beginning of the file. The close operation breaks the link between a logical file and its corresponding physical file. It also makes sure that the file buffer is flushed so everything that was written is actually sent to the file.

The I/O operations `Read` and `Write`, when viewed at a low systems level, require three items of information:

- The *logical name* of the file to be read from or written to;
- An *address* of a memory area to be used for the "inside of the computer" part of the exchange;
- An indication of *how much data* is to be read or written.

These three fundamental elements of the exchange are illustrated in Fig. 2.5.

`Read` and `Write` are sufficient for moving sequentially through a file to any desired position, but this form of access is often very inefficient. Some languages provide seek operations that let a program move directly to a certain position in a file. C provides direct access by means of the `fseek` operation. The `fseek` operation lets us view a file as a kind of large array, giving us a great deal of freedom in deciding how to organize a file.

Another useful file operation involves knowing when the end of a file has been reached. End-of-file detection is handled in different ways by different languages.

Much effort goes into shielding programmers from having to deal with the physical characteristics of files, but inevitably there are little details about the physical organization of files that programmers must know. When we try to have our program operate on files at a very low level

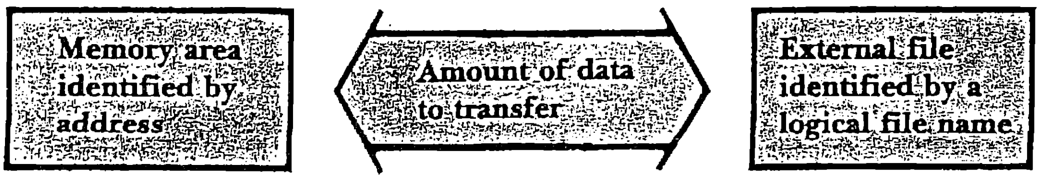

Memory area identified by address
Amount of data to transfer
External file identified by a logical file name

Figure 2.5 The exchange between memory and external device.

(as we do a great deal in this text), we must be on the lookout for little surprises inserted in our file by the operating system or applications.

The Unix file system, called the *file system,* organizes files in a tree structure, with all files and subdirectories expressible by their *pathnames.* It is possible to navigate around the file system as you work with Unix files.

Unix views both physical devices and traditional disk files as files, so, for example, a keyboard (`stdin`), a console (`stdout`), and a tape drive are all considered files. This simple conceptual view of files makes it possible in Unix to do with a very few operations what might require many times the operations on a different operating system.

I/O redirection and *pipes* are convenient shortcuts provided in Unix for transferring file data between files and *standard I/O. Header files* in Unix, such as `stdio.h`, contain special names and values that you must use when performing file operations. It is important to be aware of the most common of these in use on your system.

KEY TERMS

Access mode. Type of file access allowed. The variety of access modes permitted varies from operating system to operating system.

Buffering. When input or output is saved up rather than sent off to its destination immediately, we say that it is *buffered.* In later chapters, we find that we can dramatically improve the performance of programs that read and write data if we buffer the I/O.

Byte offset. The distance, measured in bytes, from the beginning of the file. The first byte in the file has an offset of 0, the second byte has an offset of 1, and so on.

Close. A function or system call that breaks the link between a logical file name and the corresponding physical file name.

Create. A function or system call that causes a file to be created on secondary storage and may also bind a logical name to the file's physical name—see Open. A call to create also results in the generation of information used by the system to manage the file, such as time of creation, physical location, and access privileges for anticipated users of the file.

End-of-file (EOF). An indicator within a file that the end of the file has occurred, a function that tells if the end of a file has been encountered (`end_of_file` in Ada), or a system-specific value that is returned by file-processing functions indicating that the end of a file has been encountered in the process of carrying out the function (EOF in Unix).

File descriptor. A small, nonnegative integer value returned by a Unix `open` or `creat` call that is used as a logical name for the file in later Unix system calls. This value is an index into an array of `FILE` structs that contain information about open files. The C stream functions use `FILE` pointers for their file descriptors.

File system. The name used in Unix and other operating systems to describe a collection of files and directories organized into a tree-structured hierarchy.

Header file. A file that contains definitions and declarations commonly shared among many other files and applications. In C and C++, header files are included in other files by means of the "#include" statement (see Figs. 2.2 and 2.3). The header files `iostream.h`, `stdio.h`, `file.h`, and `fcntl.h` described in this chapter contain important declarations and definitions used in file processing.

I/O redirection. The redirection of a stream of input or output from its normal place. For instance, the operator > can be used to redirect to a file output that would normally be sent to the console.

Logical file. The file as seen by the program. The use of logical files allows a program to describe operations to be performed on a file without knowing what physical file will be used. The program may then be used to process any one of a number of different files that share the same structure.

Open. A function or system call that makes a file ready for use. It may also bind a logical file name to a physical file. Its arguments include the logical file name and the physical file name and may also include information on how the file is expected to be accessed.

Pathname. A character string that describes the location of a file or directory. If the pathname starts with a /, then it gives the *absolute pathname*—the complete path from the root directory to the file. Otherwise it gives the *relative pathname*—the path relative to the current working directory.

Physical file. A file that actually exists on secondary storage. It is the file as known by the computer operating system and that appears in its file directory.

Pipe. A Unix operator specified by the symbol | that carries data from one process to another. The originating process specifies that the data is to go to `stdout`, and the receiving process expects the data from `stdin`. For example, to send the standard output from a program `makedata` to the standard input of a program called `usedata`, use the command `makedata | usedata`.

Protection mode. An indication of how a file can be accessed by various classes of users. In Unix, the protection mode is a three-digit octal number that indicates how the file can be read, written to, and executed by the owner, by members of the owner's group, and by everyone else.

Read. A function or system call used to obtain input from a file or device. When viewed at the lowest level, it requires three arguments: (1) a Source file logical name corresponding to an open file; (2) the Destination address for the bytes that are to be read; and (3) the Size or amount of data to be read.

Seek. A function or system call that sets the read/write pointer to a specified position in the file. Languages that provide seeking functions allow programs to access specific elements of a file *directly,* rather than having to read through a file from the beginning *(sequentially)* each time a specific item is desired. In C, the `fseek` function provides this capability.

Standard I/O. The source and destination conventionally used for input and output. In Unix, there are three types of standard I/O: *standard input* (`stdin`), *standard output* (`stdout`), and `stderr` *(standard error)*. By default `stdin` is the keyboard, and `stdout` and `stderr` are the console screen. I/O redirection and pipes provide ways to override these defaults.

Write. A function or system call used to provide output capabilities. When viewed at the lowest level, it requires three arguments: (1) a Destination file name corresponding to an open file; (2) the Source address of the bytes that are to be written; and (3) the Size or amount of the data to be written.

FURTHER READINGS

Introductory textbooks on C and C++ tend to treat the fundamental file operations only briefly, if at all. This is particularly true with regard to C, since there are higher-level standard I/O functions in C, such as the read operations `fget` and `fgetc`. Some books on C and/or UNIX that do provide treatment of the fundamental file operations are Kernighan and Pike (1984) and Kernighan and Ritchie (1988). These books also provide discussions of higher-level I/O functions that we omitted from our text.

An excellent explanation of the input and output classes of C++ is found in Plaugher (1995), which discusses the current (C++ version 2) and proposed draft standard for C++ input and output.

As for Unix specifically, as of this writing there are many flavors of Unix including Unix System V from AT&T, the originators of Unix, BSD (Berkeley Software Distribution) Unix from the University of California at Berkeley, and Linux from the Free Software Foundation. Each manufacturer of Unix workstations has its own operating system. There are efforts to standardize on either Posix, the international standard (ISO) Unix or OSF, the operating system of the Open Software Foundation. All of the versions are close enough that learning about any one will give you a good understanding of Unix generally. However, as you begin to use Unix, you will need reference material on the specific version that you are using. There are many accessible texts, including Sobell (1995) which covers a variety of Unix versions, including Posix, McKusick, et al (1996) on BSD, and Hekman (1997) on Linux.

EXERCISES

1. Look up operations equivalent to `Open`, `Close`, `Create`, `Read`, `Write`, and `Seek` in other high-level languages, such as Ada, Cobol, and Fortran. Compare them with the C streams or C++ stream classes.
2. For the C++ language:
   a. Make a list of the different ways to perform the file operations Create, Open, Close, Read, and Write. Why is there more than one way to do each operation?
   b. How would you use `fseek` to find the current position in a file?
   c. Show how to change the permissions on a file `myfile` so the owner has read and write permissions, group members have execute permission, and others have no permission.
   d. What is the difference between `pmode` and O_RDWR? What `pmodes` and O_RDWR are available on your system?
   e. In some typical C++ environments, such as Unix and MS-DOS, all of the following represent ways to move data from one place to another:

```
scanf       fgetc      read      cat (or type)
fscanf      gets       <         main (argc, argv)
getc        fgets      |
```

Describe as many of these as you can, and indicate how they might be useful. Which belong to the C++ language, and which belong to the operating system?

3. A couple of years ago a company we know of bought a new Cobol compiler. One difference between the new compiler and the old one was that the new compiler did not automatically close files when execution of a program terminated, whereas the old compiler did. What sorts of problems did this cause when some of the old software was executed after having been recompiled with the new compiler?

4. Consult a C++ reference and describe the values of the io_state of the stream classes in C++. Describe the characteristics of a stream when each of the state bits is set.

5. Design an experiment that uses methods `seekg`, `seekp`, `tellg`, and `tellp` to determine whether an implementation of C++ supports separate get and put pointers for the stream classes.

6. Look up the Unix command wc. Execute the following in a Unix environment, and explain why it gives the number of files in the directory.

   ls | wc -w

7. Find `stdio.h` on your system, and find what value is used to indicate end-of-file. Also examine `file.h` or `fcntl.h`, and describe in general what its contents are for.

PROGRAMMING EXERCISES

8. Make the `listcpp.cpp` program of Appendix D work with your compiler on your operating system.

9. Write a program to create a file and store a string in it. Write another program to open the file and read the string.

10. Implement the Unix command `tail` -*n*, where *n* is the number of lines from the end of the file to be copied to `stdout`.

11. Change the program `listcpp.cpp` so it reads from `cin`, rather than a file, and writes to a file, rather than `cout`. Show how to

execute the new version of the program in your programming environment, given that the input is actually in a file called `instuff`.

12. Write a program to read a series of names, one per line, from standard input, and write out those names spelled in reverse order to standard output. Use I/O redirection and pipes to do the following:
    a. Input a series of names that are typed in from the keyboard, and write them out, reversed, to a file called file1.
    b. Read the names in from `file1`; then write them out, re-reversed, to a file called `file2`.
    c. Read the names in from `file2`, reverse them again, and then sort the resulting list of reversed words using `sort`.
13. Write a program to read and write objects of class String. Include code that uses the assignment operator and the various constructors for the class. Use a debugger to determine exactly which methods are called for each statement in your program.

PROGRAMMING PROJECT

This is the second part of the programming project begun in Chapter 1. We add methods to read objects from standard input and to write formatted objects to an output stream for the classes of Chapter 1.

14. Add methods to class `Student` to read student field values from an input stream and to write the fields of an object to an output stream, nicely formatted. You may also want to be able to prompt a user to enter the field values. Use the C++ stream operations to implement these methods. Write a driver program to verify that the class is correctly implemented.
15. Add methods to class `CourseRegistration` to read course registration field values from an input stream and to write the fields of an object to an output stream, nicely formatted. You may also want to be able to prompt a user to enter the field values. Use the C++ stream operations to implement these methods. Write a driver program to verify that the class is correctly implemented.

The next part of the programming project is in Chapter 4.

CHAPTER 3

Secondary Storage and System Software

CHAPTER OBJECTIVES

- Describe the organization of typical disk drives, including basic units of organization and their relationships.
- Identify and describe the factors affecting disk access time, and describe methods for estimating access times and space requirements.
- Describe magnetic tapes, give an example of current high-performance tape systems, and investigate the implications of block size on space requirements and transmission speeds.
- Introduce the commercially important characteristics of CD-ROM storage.
- Examine the performance characteristics of CD-ROM, and see that they are very different from those of magnetic disks.
- Describe the directory structure of the CD-ROM file system, and show how it grows from the characteristics of the medium.
- Identify fundamental differences between media and criteria that can be used to match the right medium to an application.
- Describe in general terms the events that occur when data is transmitted between a program and a secondary storage device.
- Introduce concepts and techniques of buffer management.
- Illustrate many of the concepts introduced in the chapter, especially system software concepts, in the context of Unix.

CHAPTER OUTLINE

3.1 Disks
3.1.1 The Organization of Disks
3.1.2 Estimating Capacities and Space Needs
3.1.3 Organizing Tracks by Sector
3.1.4 Organizing Tracks by Block
3.1.5 Nondata Overhead
3.1.6 The Cost of a Disk Access
3.1.7 Effect of Block Size on Performance: A Unix Example
3.1.8 Disk as Bottleneck
3.2 Magnetic Tape
3.2.1 Types of Tape Systems
3.2.2 An Example of a High-Performance Tape System
3.2.3 Organization of Data on Nine-Track Tapes
3.2.4 Estimating Tape Length Requirements
3.2.5 Estimating Data Transmission Times
3.3 Disk versus Tape
3.4 Introduction to CD-ROM
3.4.1 A Short History of CD-ROM
3.4.2 CD-ROM as a File Structure Problem
3.5 Physical Organization of CD-ROM
3.5.1 Reading Pits and Lands
3.5.2 CLV Instead of CAV
3.5.3 Addressing
3.5.4 Structure of a Sector
3.6 CD-ROM Strengths and Weaknesses
3.6.1 Seek Performance
3.6.2 Data Transfer Rate
3.6.3 Storage Capacity
3.6.4 Read-Only Access
3.6.5 Asymmetric Writing and Reading
3.7 Storage as a Hierarchy
3.8 A Journey of a Byte
3.8.1 The File Manager
3.8.2 The I/O Buffer
3.8.3 The Byte Leaves Memory: The I/O Processor and Disk Controller
3.9 Buffer Management
3.9.1 Buffer Bottlenecks
3.9.2 Buffering Strategies

3.10 I/O in Unix
3.10.1 The Kernel
3.10.2 Linking File Names to Files
3.10.3 Normal Files, Special Files, and Sockets
3.10.4 Block I/O
3.10.5 Device Drivers
3.10.6 The Kernel and File Systems
3.10.7 Magnetic Tape and Unix

Good design is always responsive to the constraints of the medium and to the environment. This is as true for file structure design as it is for carvings in wood and stone. Given the ability to create, open, and close files, and to seek, read, and write, we can perform the fundamental operations of file *construction*. Now we need to look at the nature and limitations of the devices and systems used to store and retrieve files in order to prepare ourselves for file design.

If files were stored just in memory, there would be no separate discipline called file structures. The general study of data structures would give us all the tools we need to build file applications. But secondary storage devices are very different from memory. One difference, as already noted, is that accesses to secondary storage take much more time than do accesses to memory. An even more important difference, measured in terms of design impact, is that not all accesses are equal. Good file structure design uses knowledge of disk and tape performance to arrange data in ways that minimize access costs.

In this chapter we examine the characteristics of secondary storage devices. We focus on the constraints that shape our design work in the chapters that follow. We begin with a look at the major media used in the storage and processing of files, magnetic disks, and tapes. We follow this with an overview of the range of other devices and media used for secondary storage. Next, by following the journey of a byte, we take a brief look at the many pieces of hardware and software that become involved when a byte is sent by a program to a file on a disk. Finally, we take a closer look at one of the most important aspects of file management—buffering.

3.1 Disks

Compared with the time it takes to access an item in memory, disk accesses are always expensive. However, not all disk accesses are *equally* expensive. This has to do with the way a disk drive works. Disk drives[1] belong to a class of devices known as *direct access storage devices* (DASDs) because they make it possible to access data *directly*. DASDs are contrasted with *serial devices*, the other major class of secondary storage devices. Serial devices use media such as magnetic tape that permit only serial access, which means that a particular data item cannot be read or written until all of the data preceding it on the tape have been read or written in order.

Magnetic disks come in many forms. So-called *hard disks* offer high capacity and low cost per bit. Hard disks are the most common disk used in everyday file processing. *Floppy* disks are inexpensive, but they are slow and hold relatively little data. Floppies are good for backing up individual files or other floppies and for transporting small amounts of data. Removable disks use disk cartridges that can be mounted on the same drive at different times, providing a convenient form of backup storage that also makes it possible to access data directly. The Iomega Zip (100 megabytes per cartridge) and Jaz (1 gigabyte per cartridge) have become very popular among PC users.

Nonmagnetic disk media, especially optical discs, are becoming increasingly important for secondary storage. (See Sections 3.4 and 3.5 and Appendix A for a full treatment of optical disc storage and its applications.)

3.1.1 The Organization of Disks

The information stored on a disk is stored on the surface of one or more platters (Fig. 3.1). The arrangement is such that the information is stored in successive *tracks* on the surface of the disk (Fig. 3.2). Each track is often divided into a number of *sectors*. A sector is the smallest addressable portion of a disk. When a `read` statement calls for a particular byte from a disk file, the computer operating system finds the correct surface, track, and sector, reads the entire sector into a special area in memory called a *buffer*, and then finds the requested byte within that buffer.

1. When we use the terms *disks* or *disk drives*, we are referring to *magnetic* disk media.

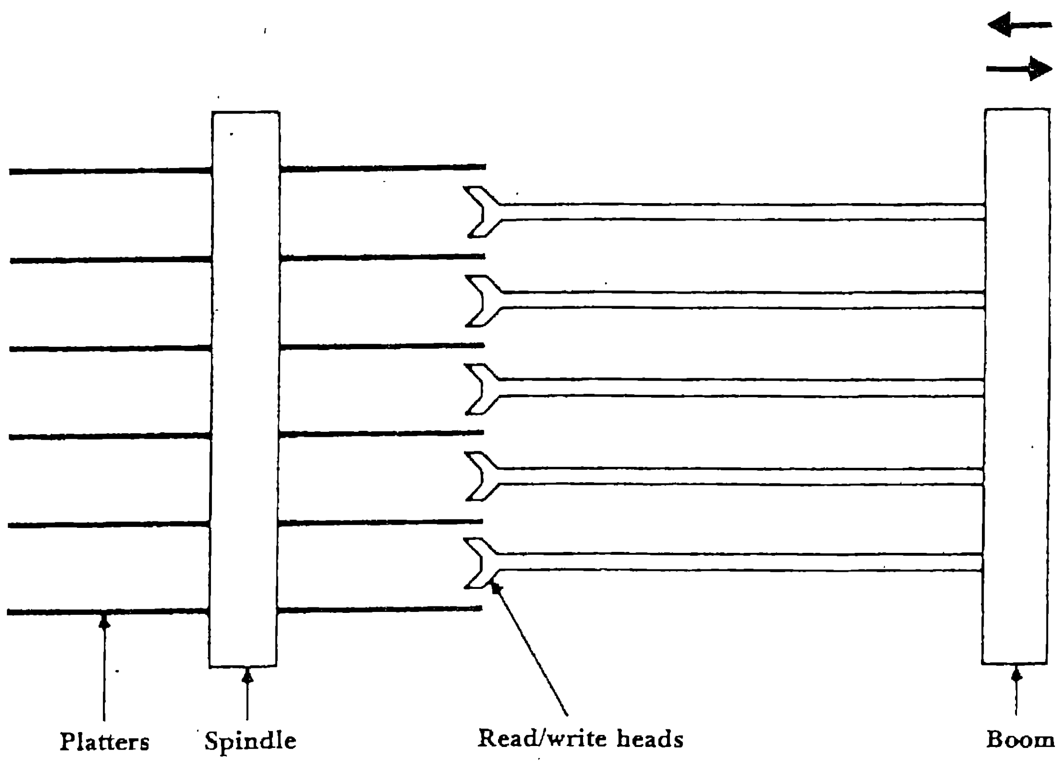


Figure 3.1 Schematic illustration of disk drive.

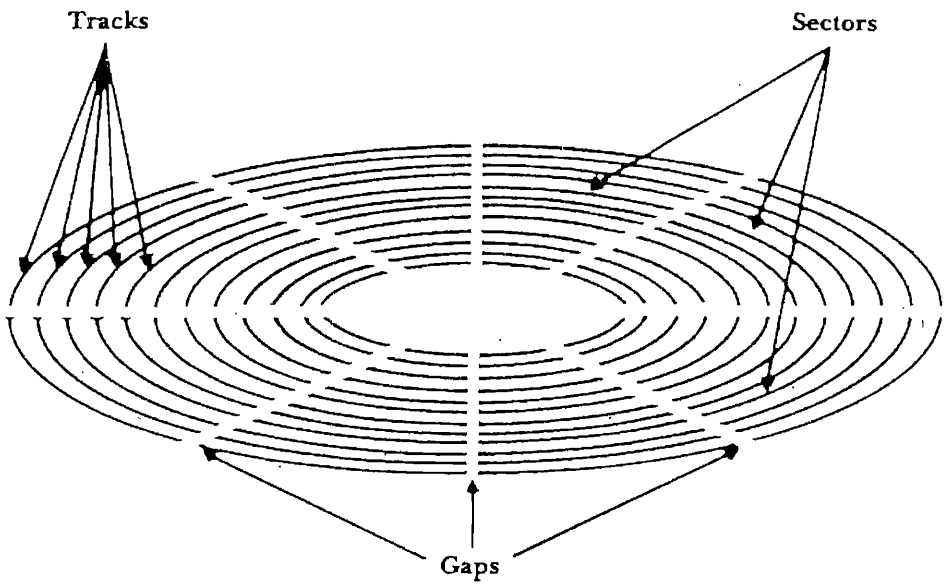


Figure 3.2 Surface of disk showing tracks and sectors.

Disk drives typically have a number of platters. The tracks that are directly above and below one another form a *cylinder* (Fig. 3.3). The significance of the cylinder is that all of the information on a single cylinder can be accessed without moving the arm that holds the read/write heads. Moving this arm is called *seeking*. This arm movement is usually the slowest part of reading information from a disk.

3.1.2 Estimating Capacities and Space Needs

Disks range in storage capacity from hundreds of millions to billions of bytes. In a typical disk, each platter has two surfaces, so the number of tracks per cylinder is twice the number of platters. The number of cylinders is the same as the number of tracks on a single surface, and each track has the same capacity. Hence the capacity of the disk is a function of the number of cylinders, the number of tracks per cylinder, and the capacity of a track.

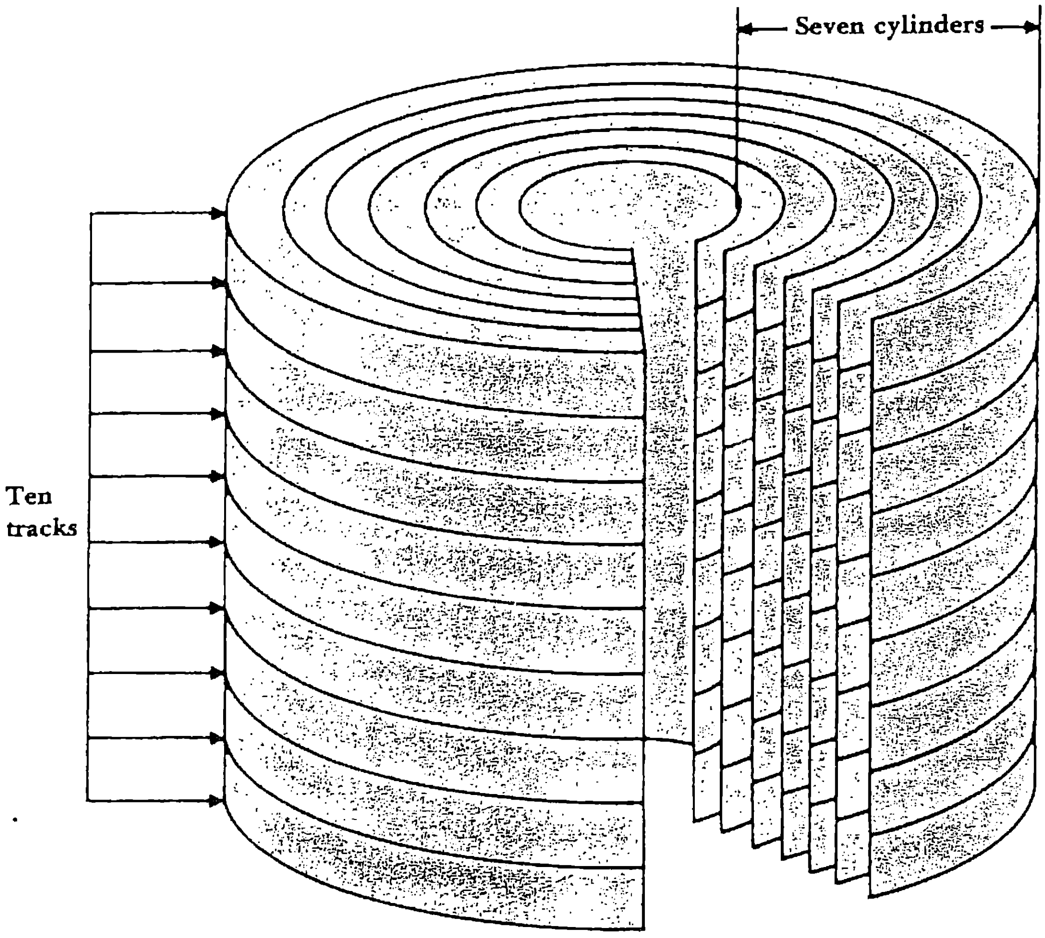


Figure 3.3 Schematic illustration of disk drive viewed as a set of seven cylinders.

The amount of data that can be held on a track and the number of tracks on a surface depend on how densely bits can be stored on the disk surface. (This in turn depends on the quality of the recording medium and the size of the read/write heads.) In 1991, an inexpensive, low-density disk held about 4 kilobytes on a track and 35 tracks on a 5-inch platter. In 1997, a Western Digital Caviar 850-megabyte disk, one of the smallest disks being manufactured, holds 32 kilobytes per track and 1,654 tracks on each surface of a 3-inch platter. A Seagate Cheetah high performance 9-gigabyte disk (still 3-inch platters) can hold about 87 kilobytes on a track and 6526 tracks on a surface. Table 3.1 shows how a variety of disk drives compares in terms of capacity, performance, and cost.

Since a cylinder consists of a group of tracks, a track consists of a group of sectors, and a sector consists of a group of bytes, it is easy to compute track, cylinder, and drive capacities.

$$\begin{aligned}\text{Track capacity} &= \text{number of sectors per track} \times \text{bytes per sector}\\ \text{Cylinder capacity} &= \text{number of tracks per cylinder} \times \text{track capacity}\\ \text{Drive capacity} &= \text{number of cylinders} \times \text{cylinder capacity.}\end{aligned}$$

Table 3.1 Specifications of the Disk Drives

Characteristic	Seagate Cheetah 9	Western Digital Caviar AC22100	Western Digital Caviar AC2850
Capacity	9000 MB	2100 MB	850 MB
Minimum (track-to-track) seek time	0.78 msec	1 msec	1 msec
Average seek time	8 msec	12 msec	10 msec
Maximum seek time	19 msec	22 msec	22 msec
Spindle speed	10000 rpm	5200 rpm	4500 rpm
Average rotational delay	3 msec	6 msec	6.6 msec
Maximum transfer rate	6 msec/track, or 14 506 bytes/msec	12 msec/track, or 2796 bytes/msec	13.3 msec/track, or 2419 bytes/msec
Bytes per sector	512	512	512
Sectors per track	170	63	63
Tracks per cylinder	16	16	16
Cylinders	526	4092	1654

If we know the number of bytes in a file, we can use these relationships to compute the amount of disk space the file is likely to require. Suppose, for instance, that we want to store a file with fifty thousand fixed-length data records on a "typical" 2.1-gigabyte small computer disk with the following characteristics:

$$\begin{aligned}\text{Number of bytes per sector} &= 512\\ \text{Number of sectors per track} &= 63\\ \text{Number of tracks per cylinder} &= 16\\ \text{Number of cylinders} &= 4092\end{aligned}$$

How many cylinders does the file require if each data record requires 256 bytes? Since each sector can hold two records, the file requires

$$\frac{50\,000}{2} = 25\,000 \text{ sectors}$$

One cylinder can hold

$$63 \times 16 = 1008 \text{ sectors}$$

so the number of cylinders required is approximately

$$\frac{25\,000}{1008} = 24.8 \text{ cylinders}$$

Of course, it may be that a disk drive with 24.8 cylinders of available space does not have 24.8 *physically contiguous* cylinders available. In this likely case, the file might, in fact, have to be spread out over dozens, perhaps even hundreds, of cylinders.

3.1.3 Organizing Tracks by Sector

There are two basic ways to organize data on a disk: by sector and by user-defined block. So far, we have mentioned only sector organizations. In this section we examine sector organizations more closely. In the following section we will look at block organizations.

The Physical Placement of Sectors

There are several views that one can have of the organization of sectors on a track. The simplest view, one that suffices for most users most of the time, is that sectors are adjacent, fixed-sized segments of a track that happen to hold a file (Fig. 3.4a). This is often a perfectly adequate way to view a file *logically*, but it may not be a good way to store sectors *physically*.

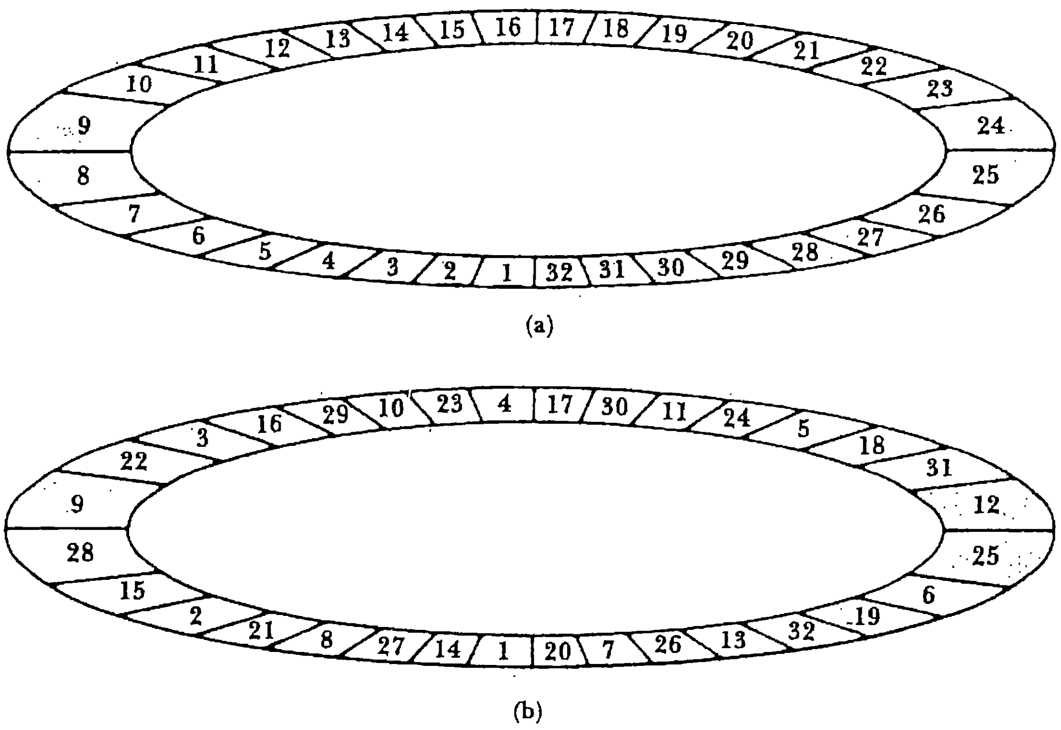


Figure 3.4 Two views of the organization of sectors on a thirty-two-sector track.

When you want to read a series of sectors that are all in the same track, one right after the other, you often cannot read *adjacent* sectors. After reading the data, it takes the disk controller a certain amount of time to process the received information before it is ready to accept more. If *logically* adjacent sectors were placed on the disk so they were also *physically* adjacent, we would miss the start of the following sector while we were processing the one we had just read in. Consequently, we would be able to read only one sector per revolution of the disk.

I/O system designers have approached this problem by *interleaving* the sectors: they leave an interval of several physical sectors between logically adjacent sectors. Suppose our disk had an *interleaving factor* of 5. The assignment of logical sector content to the thirty-two physical sectors in a track is illustrated in Fig. 3.4(b). If you study this figure, you can see that it takes five revolutions to read the entire thirty-two sectors of a track. That is a big improvement over thirty-two revolutions.

In the early 1990s, controller speeds improved so that disks can now offer 1:1 interleaving. This means that successive sectors are physically adjacent, making it possible to read an entire track in a single revolution of the disk.

Clusters

Another view of sector organization, also designed to improve performance, is the view maintained by the part of a computer's operating system that we call the *file manager.* When a program accesses a file, it is the file manager's job to map the logical parts of the file to their corresponding physical locations. It does this by viewing the file as a series of *clusters* of sectors. A cluster is a fixed number of contiguous sectors.[2] Once a given cluster has been found on a disk, all sectors in that cluster can be accessed without requiring an additional seek.

To view a file as a series of clusters and still maintain the sectored view, the file manager ties logical sectors to the physical clusters they belong to by using a *file allocation table* (FAT). The FAT contains a list of all the clusters in a file, ordered according to the logical order of the sectors they contain. With each cluster entry in the FAT is an entry giving the physical location of the cluster (Fig. 3.5).

On many systems, the system administrator can decide how many sectors there should be in a cluster. For instance, in the standard physical disk structure used by VAX systems, the system administrator sets the cluster size to be used on a disk when the disk is initialized. The default value is 3512-byte sectors per cluster, but the cluster size may be set to any value between 1 and 65 535 sectors. Since clusters represent physically contiguous groups of sectors, larger clusters will read more sectors without seeking, so the use of large clusters can lead to substantial performance gains when a file is processed sequentially.

Extents

Our final view of sector organization represents a further attempt to emphasize physical contiguity of sectors in a file and to minimize seeking even more. (If you are getting the idea that the avoidance of seeking is an important part of file design, you are right.) If there is a lot of free room on a disk, it may be possible to make a file consist entirely of contiguous clusters. When this is the case, we say that the file consists of one *extent:* all of its sectors, tracks, and (if it is large enough) cylinders form one contiguous whole (Fig. 3.6a on page 54). This is a good situation, especially if the file is to be processed sequentially, because it means that the whole file can be accessed with a minimum amount of seeking.

2. It is not always *physically* contiguous; the degree of physical contiguity is determined by the interleaving factor.

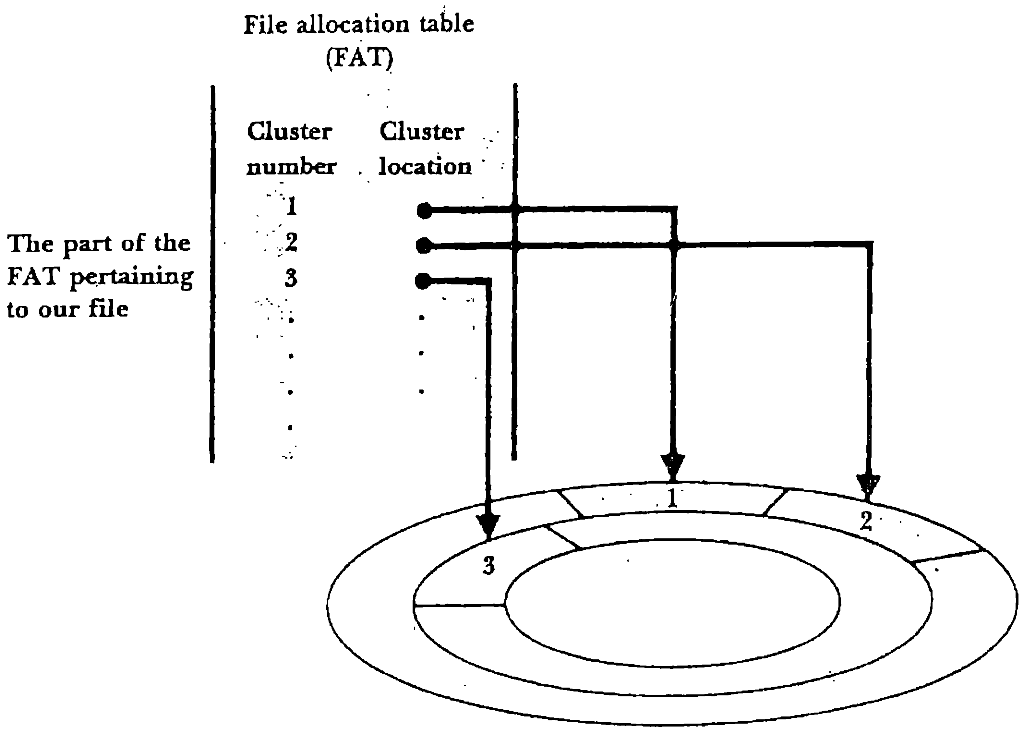


Figure 3.5 The file manager determines which cluster in the file has the sector that is to be accessed.

If there is not enough contiguous space available to contain an entire file, the file is divided into two or more noncontiguous parts. Each part is an extent. When new clusters are added to a file, the file manager tries to make them physically contiguous to the previous end of the file, but if space is unavailable, it must add one or more extents (Fig. 3.6b). The most important thing to understand about extents is that as the number of extents in a file increases, the file becomes more spread out on the disk, and the amount of seeking required to process the file increases.

Fragmentation

Generally, all sectors on a given drive must contain the same number of bytes. If, for example, the size of a sector is 512 bytes and the size of all records in a file is 300 bytes, there is no convenient fit between records and sectors. There are two ways to deal with this situation: store only one record per sector, or allow records to *span* sectors so the beginning of a record might be found in one sector and the end of it in another (Fig. 3.7).

The first option has the advantage that any record can be retrieved by retrieving just one sector, but it has the disadvantage that it might leave an enormous amount of unused space within each sector. This loss of space

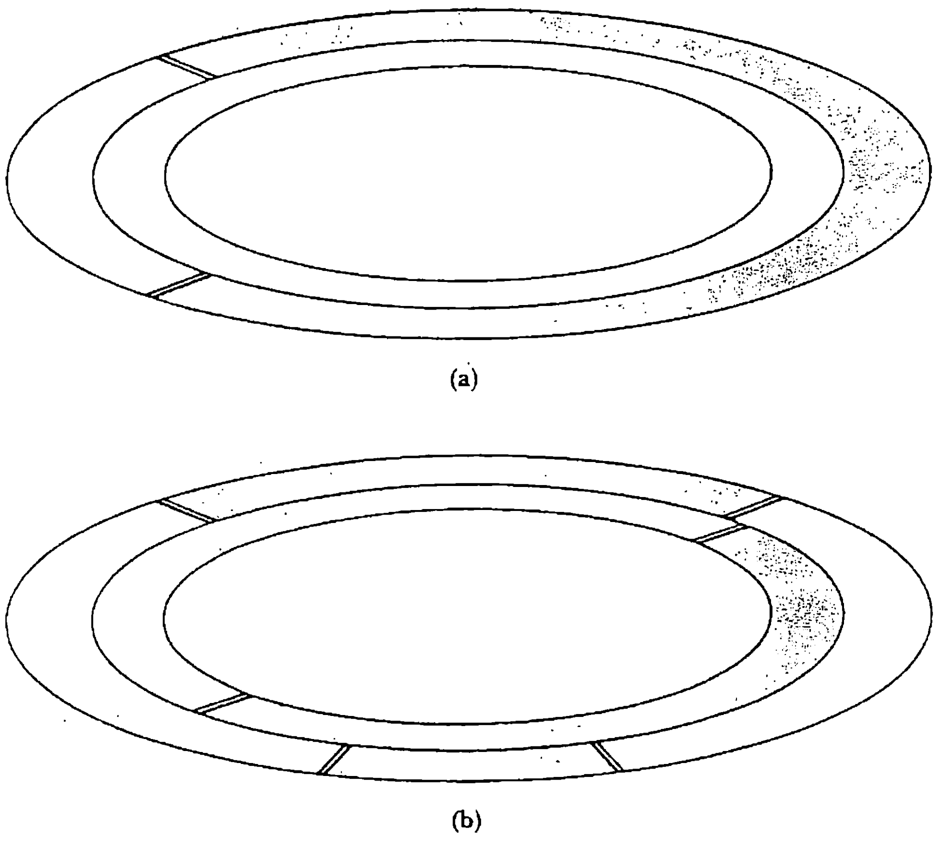


Figure 3.6 File extents (shaded area represents space on disk used by a single file).

within a sector is called *internal fragmentation.* The second option has the advantage that it loses no space from internal fragmentation, but it has the disadvantage that some records may be retrieved only by accessing two sectors.

Another potential source of internal fragmentation results from the use of clusters. Recall that a cluster is the smallest unit of space that can be allocated for a file. When the number of bytes in a file is not an exact multiple of the cluster size, there will be internal fragmentation in the last extent of the file. For instance, if a cluster consists of three 512-byte sectors, a file containing 1 byte would use up 1536 bytes on the disk; 1535 bytes would be wasted due to internal fragmentation.

Clearly, there are important trade-offs in the use of large cluster sizes. A disk expected to have mainly large files that will often be processed sequentially would usually be given a large cluster size, since internal fragmentation would not be a big problem and the performance gains might be great. A disk holding smaller files or files that are usually accessed only randomly would normally be set up with small clusters.

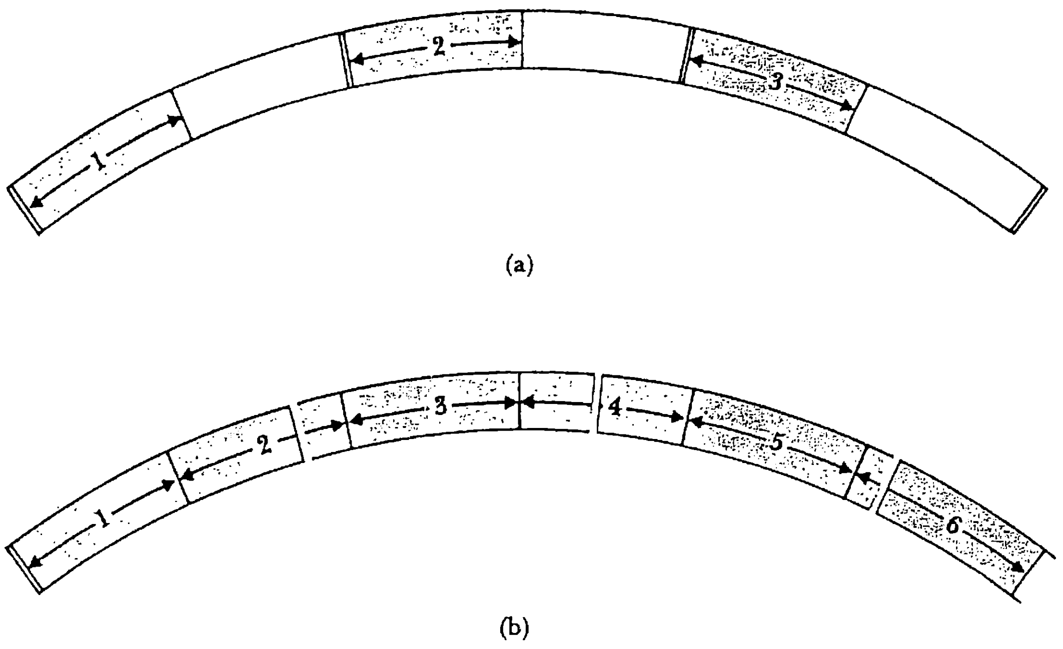


Figure 3.7 Alternate record organization within sectors (shaded areas represent data records, and unshaded areas represent unused space).

3.1.4 Organizing Tracks by Block

Sometimes disk tracks are *not* divided into sectors, but into integral numbers of user-defined *blocks* whose sizes can vary. (*Note:* The word *block* has a different meaning in the context of the Unix I/O system. See Section 3.7 for details.) When the data on a track is organized by block, this usually means that the amount of data transferred in a single I/O operation can vary depending on the needs of the software designer, not the hardware. Blocks can normally be either fixed or variable in length, depending on the requirements of the file designer and the capabilities of the operating system. As with sectors, blocks are often referred to as physical records. In this context, the physical record is the smallest unit of data that the operating system supports on a particular drive. (Sometimes the word *block* is used as a synonym for a sector or group of sectors. To avoid confusion, we do not use it in that way here.) Figure 3.8 illustrates the difference between one view of data on a sectored track and that on a blocked track.

A *block* organization does not present the sector-spanning and fragmentation problems of sectors because blocks can vary in size to fit the logical organization of the data. A block is usually organized to hold an integral number of logical records. The term *blocking factor* is used to indicate the number of records that are to be stored in each block in a file.

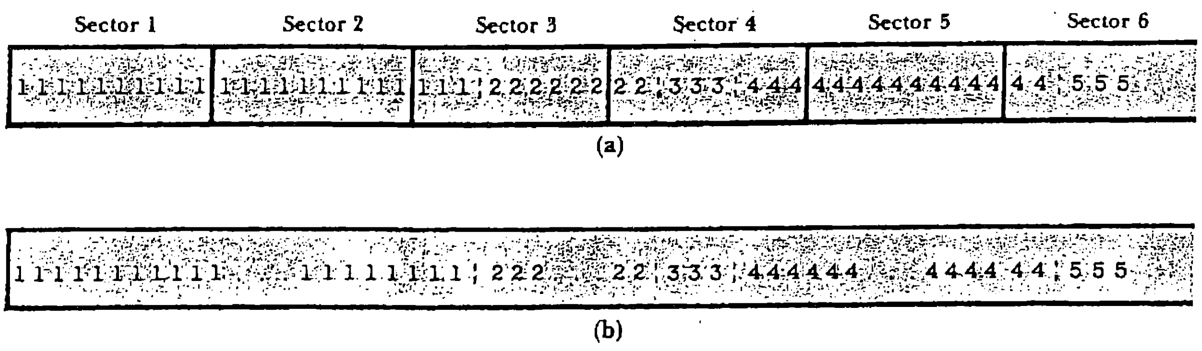


Figure 3.8 Sector organization versus block organization.

Hence, if we had a file with 300-byte records, a block-addressing scheme would let us define a block to be some convenient multiple of 300 bytes, depending on the needs of the program. No space would be lost to internal fragmentation, and there would be no need to load two blocks to retrieve one record.

Generally speaking, blocks are superior to sectors when it is desirable to have the physical allocation of space for records correspond to their logical organization. (There are disk drives that allow both sector addressing and block addressing, but we do not describe them here. See Bohl, 1981.)

In block-addressing schemes, each block of data is usually accompanied by one or more *subblocks* containing extra information about the data block. Typically there is a *count subblock* that contains (among other things) the number of bytes in the accompanying data block (Fig. 3.9a). There may also be a *key subblock* containing the key for the last record in the data block (Fig. 3.9b). When *key* subblocks are used, the disk controller can search a track for a block or record identified by a given key. This means that a program can ask its disk drive to search among all the blocks on a track for a block with a desired key. This approach can result in much more efficient searches than are normally possible with sector-addressable schemes, in which keys generally cannot be interpreted without first loading them into primary memory.

3.1.5 Nondata Overhead

Both blocks and sectors require that a certain amount of space be taken up on the disk in the form of *nondata overhead*. Some of the overhead consists of information that is stored on the disk during *preformatting*, which is done before the disk can be used.

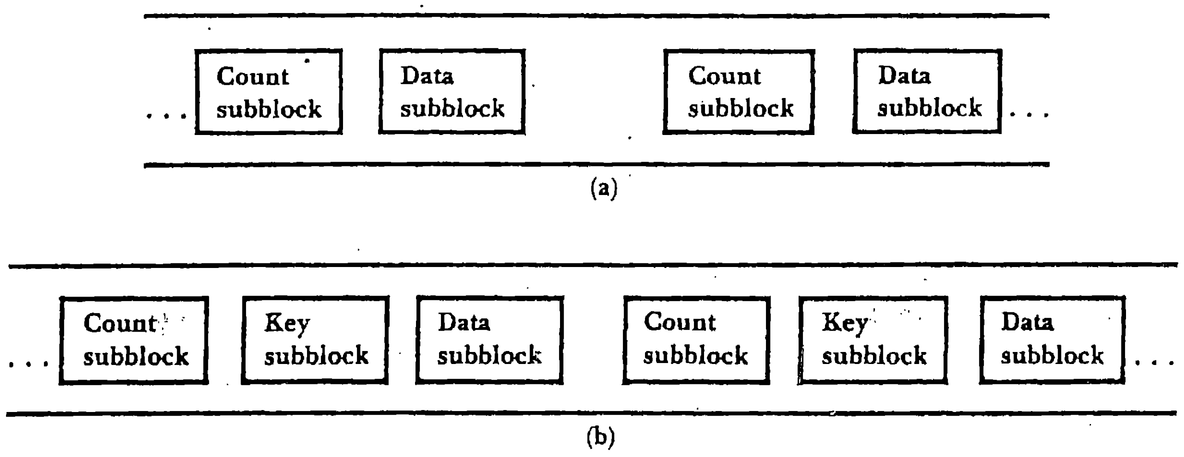


Figure 3.9 Block addressing requires that each physical data block be accompanied by one or more subblocks containing information about its contents.

On sector-addressable disks, preformatting involves storing, at the beginning of each sector, information such as sector address, track address, and condition (whether the sector is usable or defective). Preformatting also involves placing gaps and synchronization marks between fields of information to help the read/write mechanism distinguish between them. This nondata overhead usually is of no concern to the programmer. When the sector size is given for a certain drive, the programmer can assume that this is the amount of actual data that can be stored in a sector.

On a block-organized disk, some of the nondata overhead is invisible to the programmer, but some of it must be accounted for. Since subblocks and interblock gaps have to be provided with every block, there is generally more nondata information provided with blocks than with sectors. Also, since the number and size of blocks can vary from one application to another, the relative amount of space taken up by overhead can vary when block addressing is used. This is illustrated in the following example.

Suppose we have a block-addressable disk drive with 20 000 bytes per track and the amount of space taken up by subblocks and interblock gaps is equivalent to 300 bytes per block. We want to store a file containing 100-byte records on the disk. How many records can be stored per track if the blocking factor is 10? If it is 60?

1. If there are ten 100-byte records per block, each block holds 1000 bytes of data and uses 300 + 1000, or 1300, bytes of track space when overhead is taken into account. The number of blocks that can fit on a 20 000-byte track can be expressed as

$$\frac{20\,000}{1300} = 15.38 = 15$$

So fifteen blocks, or 150 records, can be stored per track. (Note that we have to take the *floor* of the result because a block cannot span two tracks.)

2. If there are sixty 100-byte records per block, each block holds 6000 bytes of data and uses 6300 bytes of track space. The number of blocks per track can be expressed as

$$\frac{20\,000}{6300} = 3$$

So three blocks, or 180 records, can be stored per track.

Clearly, the larger blocking factor can lead to more efficient use of storage. When blocks are larger, fewer blocks are required to hold a file, so there is less space consumed by the 300 bytes of overhead that accompany each block.

Can we conclude from this example that larger blocking factors always lead to more efficient storage? Not necessarily. Since we can put only an integral number of blocks on a track and since tracks are fixed in length, we almost always lose some space at the end of a track. Here we have the internal fragmentation problem again, but this time it applies to fragmentation within a *track*. The greater the block size, the greater potential amount of internal track fragmentation. What would have happened if we had chosen a blocking factor of 98 in the preceding example? What about 97?

The flexibility introduced by the use of blocks, rather than sectors, can save time, since it lets the programmer determine to a large extent how data is to be organized physically on a disk. On the negative side, blocking schemes *require* the programmer and/or operating system to do the extra work of determining the data organization. Also, the very flexibility introduced by the use of blocking schemes precludes the synchronization of I/O operations with the physical movement of the disk, which sectoring permits. This means that strategies such as sector interleaving cannot be used to improve performance.

3.1.6 The Cost of a Disk Access

To give you a feel for the factors contributing to the total amount of time needed to access a file on a fixed disk, we calculate some access times. A disk access can be divided into three distinct physical operations, each with its own cost: *seek time, rotational delay,* and *transfer time.*

Seek Time

Seek time is the time required to move the access arm to the correct cylinder. The amount of time spent seeking during a disk access depends, of course, on how far the arm has to move. If we are accessing a file sequentially and the file is packed into several consecutive cylinders, seeking needs to be done only after all the tracks on a cylinder have been processed, and then the read/write head needs to move the width of only one track. At the other extreme, if we are alternately accessing sectors from two files that are stored at opposite extremes on a disk (one at the innermost cylinder, one at the outermost cylinder), seeking is very expensive.

Seeking is likely to be more costly in a multiuser environment, where several processes are contending for use of the disk at one time, than in a single-user environment, where disk usage is dedicated to one process.

Since seeking can be very costly, system designers often go to great extremes to minimize seeking. In an application that merges three files, for example, it is not unusual to see the three input files stored on three different drives and the output file stored on a fourth drive, so no seeking need be done as I/O operations jump from file to file.

Since it is usually impossible to know exactly how many tracks will be traversed in every seek, we usually try to determine the *average seek time* required for a particular file operation. If the starting and ending positions for each access are random, it turns out that the average seek traverses one-third of the total number of cylinders that the read/write head ranges over.[3] Manufacturers' specifications for disk drives often list this figure as the average seek time for the drives. Most hard disks available today have average seek times of less than 10 milliseconds (msec), and high-performance disks have average seek times as low as 7.5 msec.

Rotational Delay

Rotational delay refers to the time it takes for the disk to rotate so the sector we want is under the read/write head. Hard disks usually rotate at about 5000 rpm, which is one revolution per 12 msec. On average, the rotational delay is half a revolution, or about 6 msec. On floppy disks, which often rotate at only 360 rpm, average rotational delay is a sluggish 83.3 msec.

3. Derivations of this result, as well as more detailed and refined models, can be found in Wiederhold (1983), Knuth (1998), Teory and Fry (1982), and Salzberg (1988).

As in the case of seeking, these averages apply only when the read/write head moves from some random place on the disk surface to the target track. In many circumstances, rotational delay can be much less than the average. For example, suppose that you have a file that requires two or more tracks, that there are plenty of available tracks on one cylinder, and that you write the file to disk sequentially, with one write call. When the first track is filled, the disk can immediately begin writing to the second track, without any rotational delay. The "beginning" of the second track is effectively staggered by just the amount of time it takes to switch from the read/write head on the first track to the read/write head on the second. Rotational delay, as it were, is virtually nonexistent. Furthermore, when you read the file back, the position of data on the second track ensures that there is no rotational delay in switching from one track to another. Figure 3.10 illustrates this staggered arrangement.

Transfer Time

Once the data we want is under the read/write head, it can be transferred. The transfer time is given by the formula

$$\text{Transfer time} = \frac{\text{number of bytes transferred}}{\text{number of bytes on a track}} \times \text{rotation time}$$

If a drive is sectored, the transfer time for one sector depends on the number of sectors on a track. For example, if there are sixty-three sectors per track, the time required to transfer one sector would be 1/63 of a revo-

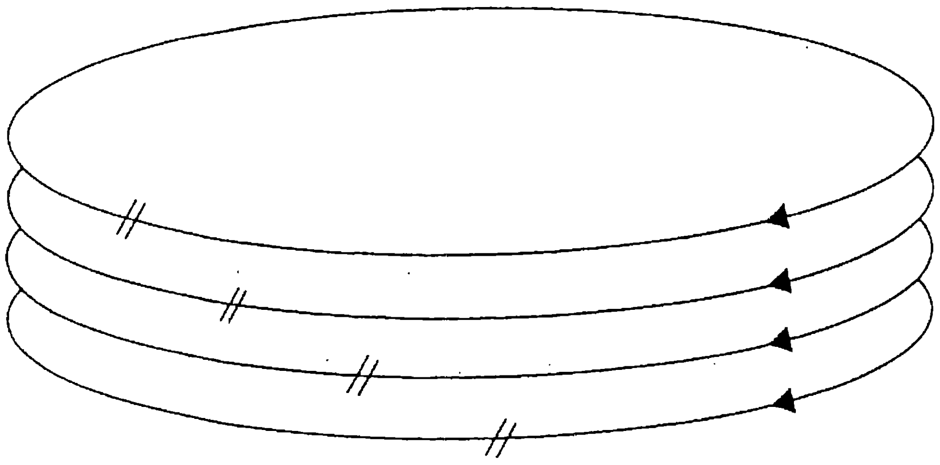


Figure 3.10 When a single file can span several tracks on a cylinder, we can stagger the beginnings of the tracks to avoid rotational delay when moving from track to track during sequential access.

lution, or 0.19 msec. The Seagate Cheetah rotates at 10 000 rpm. The transfer time for a single sector (170 sectors per track) is 0.036 msec. This results in a peak transfer rate of more than 14 megabytes per second.

Some Timing Computations

Let's look at two different file processing situations that show how different types of file access can affect access times. We will compare the time it takes to access a file *in sequence* with the time it takes to access all of the records in the file *randomly*. In the former case, we use as much of the file as we can whenever we access it. In the random-access case, we are able to use only one record on each access.

The basis for our calculations is the high performance Seagate Cheetah 9-gigabyte fixed disk described in Table 3.1. Although it is typical only of a certain class of fixed disk, the observations we draw as we perform these calculations are quite general. The disks used with personal computers are smaller and slower than this disk, but the nature and relative costs of the factors contributing to total access times are essentially the same.

The highest performance for data transfer is achieved when files are in one-track units. Sectors are interleaved with an interleave factor of 1, so data on a given track can be transferred at the stated transfer rate.

Let's suppose that we wish to know how long it will take, using this drive, to read an 8 704 000-byte file that is divided into thirty-four thousand 256-byte records. First we need to know how the file is distributed on the disk. Since the 4096-byte cluster holds sixteen records, the file will be stored as a sequence of 2125 4096-byte sectors occupying one hundred tracks.

This means that the disk needs one hundred tracks to hold the entire 8704 kilobytes that we want to read. We assume a situation in which the one hundred tracks are randomly dispersed over the surface of the disk. (This is an extreme situation chosen to dramatize the point we want to make. Still, it is not so extreme that it could not easily occur on a typical overloaded disk that has a large number of small files.)

Now we are ready to calculate the time it would take to read the 8704-kilobyte file from the disk. We first estimate the time it takes to read the file sector by sector *in sequence*. This process involves the following operations for each track:

Average seek	8 msec
Rotational delay	3 msec
Read one track	6 msec
Total	17 msec

We want to find and read one hundred tracks, so

Total time = 100 × 17 msec = 1700 msec = 1.7 seconds

Now let's calculate the time it would take to read in the same thirty-four thousand records using *random access* instead of sequential access. In other words, rather than being able to read one sector right after another, we assume that we have to access the records in an order that requires jumping from track to track every time we read a new sector. This process involves the following operations for each record:

Average seek	8 msec
Rotational delay	3 msec
Read one cluster (1/21.5 × 6 msec)	0.28 msec
Total	11.28 msec

Total time = 34 000 × 11.28 msec = 9250 msec = 9.25 seconds

This difference in performance between sequential access and random access is very important. If we can get to the right location on the disk and read a lot of information sequentially, we are clearly much better off than if we have to jump around, *seeking* every time we need a new record. Remember that seek time is very expensive; when we are performing disk operations, we should try to minimize seeking.

3.1.7 Effect of Block Size on Performance: A Unix Example

In deciding how best to organize disk storage allocation for several versions of BSD Unix, the Computer Systems Research Group (CSRG) in Berkeley investigated the trade-offs between block size and performance in a Unix environment (Leffler et al., 1989). The results of the research provide an interesting case study involving trade-offs between block size, fragmentation, and access time.

The CSRG research indicated that a minimum block size of 512 bytes, standard at the time on Unix systems, was not very efficient in a typical Unix environment. Files that were several blocks long often were scattered over many cylinders, resulting in frequent seeks and thereby significantly decreasing throughput. The researchers found that doubling the block size to 1024 bytes improved performance by more than a factor of 2. But even with 1024-byte blocks, they found that throughput was only about 4 percent of the theoretical maximum. Eventually, they found that 4096-byte blocks provided the fastest throughput, but this led to large amounts of wasted space due to internal fragmentation. These results are summarized in Table 3.2.

Table 3.2 The amount of wasted space as a function of block size.

Space Used (MB)	Percent Waste	Organization
775.2	0.0	Data only, no separation between files
807.8	4.2	Data only, each file starts on 512-byte boundary
828.7	6.9	Data + inodes, 512-byte block Unix file system
866.5	11.8	Data + inodes, 1024-byte block Unix file system
948.5	22.4	Data + inodes, 2048-byte block Unix file system
1128.3	45.6	Data + inodes, 4096-byte block Unix file system

From *The Design and Implementation of the 4.3BSD Unix Operating System*, Leffler et al., p. 198.

To gain the advantages of both the 4096-byte and the 512-byte systems, the Berkeley group implemented a variation of the cluster concept (see Section 3.1.3). In the new implementation, the researchers allocate 4096-byte blocks for files that are big enough to need them; but for smaller files, they allow the large blocks to be divided into one or more fragments. With a fragment size of 512 bytes, as many as eight small files can be stored in one block, greatly reducing internal fragmentation. With the 4096/512 system, wasted space was found to decline to about 12 percent.

3.1.8 Disk as Bottleneck

Disk performance is increasing steadily, even dramatically, but disk speeds still lag far behind local network speeds. A high-performance disk drive with 50 kilobytes per track can transmit at a peak rate of about 5 megabytes per second, and only a fraction of that under normal conditions. High-performance networks, in contrast, can transmit at rates of as much as 100 megabytes per second. The result can often mean that a process is *disk bound*—the network and the computer's central processing unit (CPU) have to wait inordinate lengths of time for the disk to transmit data.

A number of techniques are used to solve this problem. One is multiprogramming, in which the CPU works on other jobs while waiting for the data to arrive. But if multiprogramming is not available or if the process simply cannot afford to lose so much time waiting for the disk, methods must be found to speed up disk I/O.

One technique now offered on many high-performance systems is called *striping*. Disk striping involves splitting the parts of a file on several different drives, then letting the separate drives deliver parts of the file to the network simultaneously. Disk striping can be used to put different blocks of the file on different drives or to spread individual blocks onto different drives.

Disk striping exemplifies an important concept that we see more and more in system configurations—*parallelism*. Whenever there is a bottleneck at some point in the system, consider duplicating the source of the bottleneck and configure the system so several of them operate in parallel.

If we put different blocks on different drives, independent processes accessing the same file will not necessarily interfere with each other. This improves the throughput of the system by improving the speed of multiple jobs, but it does not necessarily improve the speed of a single drive. There is a significant possibility of a reduction in seek time, but there is no guarantee.

The speed of single jobs that do large amounts of I/O can be significantly improved by spreading each block onto many drives. This is commonly implemented in *RAID* (redundant array of independent disks) systems which are commercially available for most computer systems. For an eight-drive RAID, for example, the controller receives a single block to write and breaks it into eight pieces, each with enough data for a full track. The first piece is written to a particular track of the first disk, the second piece to the same track of the second disk, and so on. The write occurs at a sustained rate of eight times the rate of a single drive. The read operation is similar, the same track is read from each drive, the block in reassembled in cache, and the cache contents are transmitted back through the I/O channel. RAID systems are supported by a large memory cache on the disk controller to support very large blocks.

Another approach to solving the disk bottleneck is to avoid accessing the disk at all. As the cost of memory steadily decreases, more and more programmers are using memory to hold data that a few years ago had to be kept on a disk. Two effective ways in which memory can be used to replace secondary storage are memory disks and disk caches.

A *RAM disk* is a large part of memory configured to simulate the behavior of a mechanical disk in every respect except speed and volatility. Since data can be located in memory without a seek or rotational delay, RAM disks can provide much faster access than mechanical disks. Since memory is normally volatile, the contents of a RAM disk are lost when the computer is turned off. RAM disks are often used in place of floppy disks because they are much faster than floppies and because relatively little memory is needed to simulate a typical floppy disk.

A *disk cache*[4] is a large block of memory configured to contain *pages* of data from a disk. A typical disk-caching scheme might use a 256-kilobyte cache with a disk. When data is requested from secondary memory, the file manager first looks into the disk cache to see if it contains the page with the requested data. If it does, the data can be processed immediately. Otherwise, the file manager reads the page containing the data from disk, replacing some page already in the disk cache.

Cache memory can provide substantial improvements in performance, especially when a program's data access patterns exhibit a high degree of *locality*. Locality exists in a file when blocks that are accessed in close temporal sequence are stored close to one another on the disk. When a disk cache is used, blocks that are close to one another on the disk are much more likely to belong to the page or pages that are read in with a single read, diminishing the likelihood that extra reads are needed for extra accesses.

RAM disks and cache memory are examples of *buffering*, a very important and frequently used family of I/O techniques. We take a closer look at buffering in Section 3.9.

In these three techniques we see once again examples of the need to make trade-offs in file processing. With RAM disks and disk caches, there is tension between the cost/capacity advantages of disk over memory on the one hand, and the speed of memory on the other. Striping provides opportunities to increase throughput enormously, but at the cost of a more complex and sophisticated disk management system. Good file design balances these tensions and costs creatively.

3.2 Magnetic Tape

Magnetic tape units belong to a class of devices that provide no direct accessing facility but can provide very rapid sequential access to data. Tapes are compact, stand up well under different environmental conditions, are easy to store and transport, and are less expensive than disks. Many years ago tape systems were widely used to store application data. An application that needed data from a specific tape would issue a request

4. The term *cache* (as opposed to *disk cache*) generally refers to a very high-speed block of primary memory that performs the same types of performance-enhancing operations with respect to memory that a disk cache does with respect to secondary memory.

for the tape, which would be mounted by an operator onto a tape drive. The application could then directly read and write on the tape. The tremendous reduction in the cost of disk systems has changed the way tapes are used. At present, tapes are primarily used as archival storage. That is, data is written to tape to provide low cost storage and then copied to disk whenever it is needed. Tapes are very common as backup devices for PC systems. In high performance and high volume applications, tapes are commonly stored in racks and supported by a robot system that is capable of moving tapes between storage racks and tape drives.

3.2.1 Types of Tape Systems

There has been tremendous improvement in tape technology in the past few years. There are now a variety of tape formats with prices ranging from $150 to $150,000 per tape drive. For $150, a PC owner can add a tape backup system, with sophisticated backup software, that is capable of storing 4 gigabytes of data on a single $30 tape. For larger systems, a high performance tape system could easily store hundreds of terabytes in a tape robot system costing millions of dollars. Table 3.3 shows a comparison of some current tape systems.

In the past, most computer installations had a number of reel-to-reel tape drives and large numbers of racks or cabinets holding tapes. The primary media was one-half inch magnetic tape on 10.5-inch reels with 3600 feet of tape. In the next section we look at the format and data transfer capabilities of these tape systems which use nine linear tracks and are usually referred to as *nine-track tapes.*

Table 3.3 Comparison of some current tape systems

Tape Model	Media Format	Loading	Capacity	Tracks	Transfer Rate
9-track	one-half inch reel	autoload	200 MB	9 linear	1 MB/sec
Digital linear tape	DLT cartridge	robot	35 GB	36 linear	5 MB/sec
HP Colorado T3000	one-quarter inch cartridge	manual	1.6 GB	helical	0.5 MB/sec
StorageTek Redwood	one-half inch cartridge	robot silo	50 GB	helical	10 MB/sec

Newer tape systems are usually based on a *tape cartridge* medium where the tape and its reels are contained in a box. The tape media formats that are available include 4 mm, 8 mm, VHS, 1/2 inch, and 1/4 inch.

3.2.2 An Example of a High-Performance Tape System

The StorageTek Redwood SD3 is one of the highest-performance tape systems available in 1997. It is usually configured in a *silo* that contains storage racks, a tape robot, and multiple tape drives. The tapes are 4-by-4-inch cartridges with one-half inch tape. The tapes are formatted with helical tracks. That is, the tracks are at an angle to the linear direction of the tape. The number of individual tracks is related to the length of the tape rather than the width of the tape as in linear tapes. The expected reliable storage time is more than twenty years, and average durability is 1 million head passes.

The performance of the SD3 is achieved with tape capacities of up to 50 gigabytes and a sustained transfer rate of 11 megabytes per second. This transfer rate is necessary to store and retrieve data produced by the newest generation of scientific experimental equipment, including the Hubbell telescope, the Earth Observing System (a collection of weather satellites), seismographic instruments, and a variety of particle accelerators.

An important characteristic of a tape silo system is the speed of seeking, rewinding, and loading tapes. The SD3 silo using 50-gigabyte tapes has an average seek time of 53 seconds and can rewind in a maximum of 89 seconds. The load time is only 17 seconds. The time to read or write a full tape is about 75 minutes. Hence, the overhead to rewind, unload, and load is only 3 percent. Another way to look at this is that any tape in the silo can be mounted in under 2 minutes with no operator intervention.

3.2.3 Organization of Data on Nine-Track Tapes

Since tapes are accessed sequentially, there is no need for addresses to identify the locations of data on a tape. On a tape, the logical position of a byte within a file corresponds directly to its physical position relative to the start of the file. We may envision the surface of a typical tape as a set of parallel tracks, each of which is a sequence of bits. If there are nine tracks (see Fig. 3.11), the nine bits that are at corresponding positions in the nine respective tracks are taken to constitute 1 byte, plus a *parity bit.* So a byte can be thought of as a one-bit-wide slice of tape. Such a slice is called a *frame.*

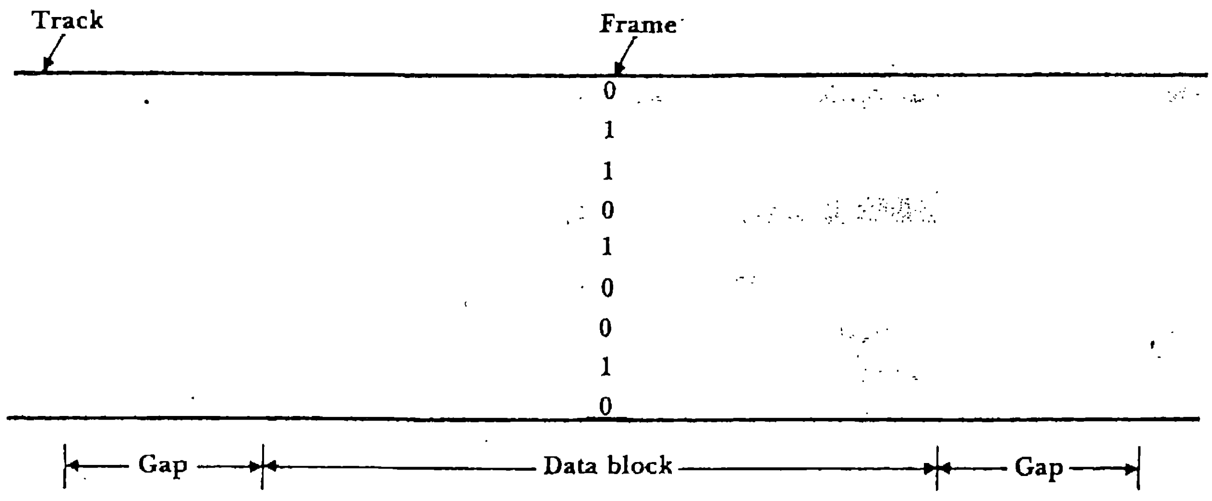


Figure 3.11 Nine-track tape.

The parity bit is not part of the data but is used to check the validity of the data. If *odd parity* is in effect, this bit is set to make the number of 1 bits in the frame *odd*. Even parity works similarly but is rarely used with tapes.

Frames (bytes) are grouped into data blocks whose size can vary from a few bytes to many kilobytes, depending on the needs of the user. Since tapes are often read one block at a time and since tapes cannot stop or start instantaneously, blocks are separated by *interblock gaps*, which contain no information and are long enough to permit stopping and starting. When tapes use odd parity, no valid frame can contain all 0 bits, so a large number of consecutive 0 frames is used to fill the interrecord gap.

Tape drives come in many shapes, sizes, and speeds. Performance differences among drives can usually be measured in terms of three quantities:

Tape density—commonly 800, 1600, or 6250 bits per inch (bpi) per track, but recently as much as 30 000 bpi;

Tape speed—commonly 30 to 200 inches per second (ips); and

Size of interblock gap—commonly between 0.3 inch and 0.75 inch.

Note that a 6250-bpi nine-track tape contains 6250 bits per inch per track, and 6250 *bytes* per inch when the full nine tracks are taken together. Thus in the computations that follow, 6250 bpi is usually taken to mean 6250 bytes of data per inch.

3.2.4 Estimating Tape Length Requirements

Suppose we want to store a backup copy of a large mailing-list file with one million 100-byte records. If we want to store the file on a 6250-bpi tape that has an interblock gap of 0.3 inches, how much tape is needed?

To answer this question we first need to determine what takes up space on the tape. There are two primary contributors: interblock gaps and data blocks. For every data block there is an interblock gap. If we let

b = the physical length of a data block,
g = the length of an interblock gap, and
n = the number of data blocks

then the space requirement *s* for storing the file is

$$s = n \times (b + g)$$

We know that *g* is 0.3 inch, but we do not know what *b* and *n* are. In fact, *b* is whatever we want it to be, and *n* depends on our choice of *b*. Suppose we choose each data block to contain one 100-byte record. Then *b*, the length of each block, is given by

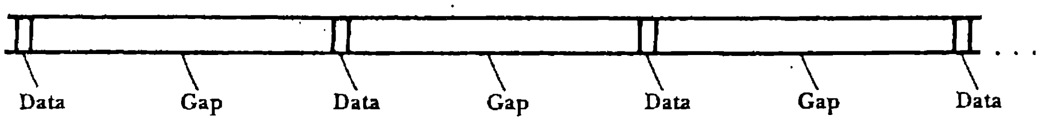
$$b = \frac{\text{block size (bytes per block)}}{\text{tape density (bytes per inch)}} - \frac{100}{6250} = 0.016 \text{ inch}$$

and *n*, the number of blocks, is 1 million (one per record).

The number of records stored in a physical block is called the *blocking factor*. It has the same meaning it had when it was applied to the use of blocks for disk storage. The blocking factor we have chosen here is 1 because each block has only one record. Hence, the space requirement for the file is

$$\begin{aligned} s &= 1\,000\,000 \times (0.016 + 0.3) \text{ inch} \\ &= 1\,000\,000 \times 0.316 \text{ inch} \\ &= 316\,000 \text{ inches} \\ &= 26\,333 \text{ feet} \end{aligned}$$

Magnetic tapes range in length from 300 feet to 3600 feet, with 2400 feet being the most common length. Clearly, we need quite a few 2400-foot tapes to store the file. Or do we? You may have noticed that our choice of block size was not a very smart one from the standpoint of space usage. The interblock gaps in the physical representation of the file take up about *nineteen times* as much space as the data blocks do. If we were to take a snapshot of our tape, it would look something like this:

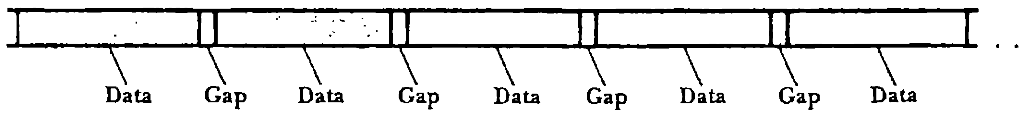


Most of the space on the tape is not used!

Clearly, we should consider increasing the relative amount of space used for actual data if we want to try to squeeze the file onto one 2400-foot tape. If we increase the blocking factor, we can *decrease* the number of blocks, which decreases the number of interblock gaps, which in turn decreases the amount of space consumed by interblock gaps. For example, if we increase the blocking factor from 1 to 50, the number of blocks becomes

$$n = \frac{1\ 000\ 000}{50} = 20\ 000$$

and the space requirement for interblock gaps decreases from 300 000 inches to 6000 inches. The space requirement for the data is of course the same as was previously. What has changed is the *relative* amount of space occupied by the gaps, as compared to the data. Now a snapshot of the tape would look much different:



We leave it to you to show that the file can fit easily on one 2400-foot tape when a blocking factor of 50 is used.

When we compute the space requirements for our file, we produce numbers that are quite specific to our file. A more general measure of the effect of choosing different block sizes is *effective recording density.* The effective recording density is supposed to reflect the amount of actual data that can be stored per inch of tape. Since this depends exclusively on the relative sizes of the interblock gap and the data block, it can be defined as

$$\frac{\text{number of bytes per block}}{\text{number of inches required to store a block}}$$

When a blocking factor of 1 is used in our example, the number of bytes per block is 100, and the number of inches required to store a block is 0.316. Hence, the effective recording density is

$$\frac{100 \text{ bytes}}{0.316 \text{ inches}} = 316.4 \text{ bpi}$$

which is a far cry from the *nominal* recording density of 6250 bpi.

Either way you look at it, space utilization is sensitive to the relative sizes of data blocks and interblock gaps. Let us now see how they affect the amount of *time* it takes to transmit tape data.

3.2.5 Estimating Data Transmission Times

If you understand the role of interblock gaps and data block sizes in determining effective recording density, you can probably see immediately that these two factors also affect the rate of data transmission. Two other factors that affect the rate of data transmission to or from tape are the nominal recording density and the speed with which the tape passes the read/write head. If we know these two values, we can compute the *nominal data transmission rate:*

$$\text{Nominal rate} = \text{tape density (bpi)} \times \text{tape speed (ips)}$$

Hence, our 6250-bpi, 200-ips tape has a nominal transmission rate of

$$\begin{aligned} 6250 \times 200 &= 1\,250\,000 \text{ bytes/sec} \\ &= 1250 \text{ kilobytes/sec} \end{aligned}$$

This rate is competitive with most disk drives.

But what about those interblock gaps? Once our data gets dispersed by interblock gaps, the *effective transmission rate* certainly suffers. Suppose, for example, that we use our blocking factor of 1 with the same file and tape drive discussed in the preceding section (one million 100-byte records, 0.3-inch gap). We saw that the effective recording density for this tape organization is 316.4 bpi. If the tape is moving at a rate of 200 ips, then its effective transmission rate is

$$\begin{aligned} 316.4 \times 200 &= 63\,280 \text{ bytes/sec} \\ &= 63.3 \text{ kilobytes/sec} \end{aligned}$$

a rate that is about *one-twentieth* the nominal rate!

It should be clear that a blocking factor larger than 1 improves on this result and that a substantially larger blocking factor improves on it substantially.

Although there are other factors that can influence performance, block size is generally considered to be the one variable with the greatest influence on space utilization and data transmission rate. The other factors we have included—gap size, tape speed, and recording density—are often beyond the control of the user. Another factor that can sometimes be important is the time it takes to start and stop the tape. We consider start/stop time in the exercises at the end of this chapter.

3.3 Disk versus Tape

In the past, magnetic tape and magnetic disk accounted for the lion's share of all secondary storage applications. Disk was excellent for random access and storage of files for which immediate access was desired; tape was ideal for processing data sequentially and for long-term storage of files. Over time, these roles have changed somewhat in favor of disk.

The major reason that tape was preferable to disk for sequential processing is that tapes are dedicated to one process, while disk generally serves several processes. This means that between accesses a disk read/write head tends to move away from the location where the next sequential access will occur, resulting in an expensive seek; the tape drive, being dedicated to one process, pays no such price in seek time.

This problem of excessive seeking has gradually diminished, and disk has taken over much of the secondary storage niche previously occupied by tape. This change is largely because of the continued dramatic decreases in the cost of disk and memory storage. To understand this change fully, we need to understand the role of memory buffer space in performing I/O.[5] Briefly, it is that performance depends largely on how big a chunk of file we can transmit at any time; as more memory space becomes available for I/O buffers, the number of accesses decreases correspondingly, which means that the number of seeks required goes down as well. Most systems now available, even small systems, have enough memory to decrease the number of accesses required to process most files that disk becomes quite competitive with tape for sequential processing. This change, along with the superior versatility and decreasing costs of disks, has resulted in use of disk for most sequential processing, which in the past was primarily the domain of tape.

This is not to say that tapes should not be used for sequential processing. If a file is kept on tape and there are enough drives available to use them for sequential processing, it may be more efficient to process the file directly from tape than to stream it to disk and process it sequentially.

Although it has lost ground to disk in sequential processing applications, tape remains important as a medium for long-term archival storage. Tape is still far less expensive than magnetic disk, and it is very easy and fast to stream large files or sets of files between tape and disk. In this context, tape has emerged as one of our most important media (along with CD-ROM) for *tertiary* storage.

5. Techniques for memory buffering are covered in Section 3.9.

3.4 Introduction to CD-ROM

CD-ROM is an acronym for Compact Disc, Read-Only Memory.[6] It is a CD audio disc that contains digital data rather than digital sound. CD-ROM is commercially interesting because it can hold a lot of data and can be reproduced cheaply. A single disc can hold more than 600 megabytes of data. That is approximately two hundred thousand printed pages, enough storage to hold almost four hundred books the size of this one. Replicates can be stamped from a master disc for about only a dollar a copy.

CD-ROM is read-only (or write-once) in the same sense as a CD audio disc: once it has been recorded, it cannot be changed. It is a publishing medium, used for distributing information to many users, rather than a data storage and retrieval medium like magnetic disks. CD-ROM has become the preferred medium for distribution of all types of software and for publication of database information such as telephone directories, zip codes, and demographic information. There are also many CD-ROM products that deliver textual data, such as bibliographic indexes, abstracts, dictionaries, and encyclopedias, often in association with digitized images stored on the disc. They are also used to publish video information and, of course, digital audio.

3.4.1 A Short History of CD-ROM

CD-ROM is the offspring of videodisc technology developed in the late 1960s and early 1970s, before the advent of the home VCR. The goal was to store movies on disc. Different companies developed a number of methods for storing video signals, including the use of a needle to respond mechanically to grooves in a disc, much like a vinyl LP record does. The consumer products industry spent a great deal of money developing the different technologies, including several approaches to optical storage, then spent years fighting over which approach should become standard. The surviving format is one called LaserVision. By the time LaserVision emerged as the winner, the competing developers had not only spent enormous sums of money but had also lost important market opportunities. These hard lessons were put to use in the subsequent development of CD audio and CD-ROM.

6. Usually we spell disk with a *k*, but the convention among optical disc manufacturers is to spell it with a *c*.

From the outset, there was an interest in using LaserVision discs to do more than just record movies. The LaserVision format supports recording in both a constant linear velocity (CLV) format that maximizes storage capacity and a constant angular velocity (CAV) format that enables fast seek performance. By using the CAV format to access individual video frames quickly, a number of organizations, including the MIT Media Lab, produced prototype interactive video discs that could be used to teach and entertain.

In the early 1980s, a number of firms began looking at the possibility of storing digital, textual information on LaserVision discs. LaserVision stores data in an analog form; it is, after all, storing an analog video signal. Different firms came up with different ways of encoding digital information in analog form so it could be stored on the disc. The capabilities demonstrated in the prototypes and early, narrowly distributed products were impressive. The videodisc has a number of performance characteristics that make it a technically more desirable medium than the CD-ROM; in particular, one can build drives that seek quickly and deliver information from the disc at a high rate of speed. But, reminiscent of the earlier disputes over the physical format of the videodisc, each of these pioneers in the use of LaserVision discs as computer peripherals had incompatible encoding schemes and error correction techniques. There was no standard format, and none of the firms was large enough to impose its format over the others through sheer marketing muscle. Potential buyers were frightened by the lack of a standard; consequently, the market never grew.

During this same period Philips and Sony began work on a way to store music on optical discs. Rather than storing the music in the kind of analog form used on videodiscs, they developed a digital data format. Philips and Sony had learned hard lessons from the expensive standards battles over videodiscs. This time they worked with other players in the consumer products industry to develop a licensing system that resulted in the emergence of CD audio as a broadly accepted, standard format as soon as the first discs and players were introduced. CD audio appeared in the United States in early 1984. CD-ROM, which is a digital data format built on top of the CD audio standard, emerged shortly thereafter. The first commercially available CD-ROM drives appeared in 1985.

Not surprisingly, the firms that were delivering digital data on LaserVision discs saw CD-ROM as a threat to their existence. They also recognized, however, that CD-ROM promised to provide what had always eluded them in the past: a standard physical format. Anyone with a CD-ROM drive was guaranteed that he or she could find and read a sector off of any disc manufactured by any firm. For a storage medium to be used in publishing, standardization at such a fundamental level is essential.

What happened next is remarkable considering the history of standards and cooperation within an industry. The firms that had been working on products to deliver computer data from videodiscs recognized that a standard physical format, such as that provided by CD-ROM, was not enough. A standard physical format meant that everyone would be able to read sectors off of any disc. But computer applications do not work in terms of sectors; they store data in files. Having an agreement about finding sectors, without further agreement about how to organize the sectors into files, is like everyone agreeing on an alphabet without having settled on how letters are to be organized into words on a page. In late 1985 the firms emerging from the videodisc/digital data industry, all of which were relatively small, called together many of the much larger firms moving into the CD-ROM industry to begin work on a standard file system that would be built on top of the CD-ROM format. In a rare display of cooperation, the different firms, large and small, worked out the main features of a file system standard by early summer of 1986; that work has become an official international standard for organizing files on CD-ROM.

The CD-ROM industry is still young, though in the past years it has begun to show signs of maturity: it is moving away from concentration on such matters as disc formats to a concern with CD-ROM applications. Rather than focusing on the new medium in isolation, vendors are seeing it as an enabling mechanism for new systems. As it finds more uses in a broader array of applications, CD-ROM looks like an optical publishing technology that will be with us over the long term.

Recordable CD drives make it possible for users to store information on CD. The price of the drives and the price of the blank recordable CDs make this technology very appealing for backup. Unfortunately, while the speed of CD readers has increased substantially, with 12X (twelve times CD audio speed) as the current standard, CD recorders work no faster than 2X, or about 300 kilobytes per second.

The latest new technology for CDs is the DVD, which stands for Digital Video Disc, or Digital Versatile Disk. The Sony Corporation has developed DVD for the video market, especially for the new high definition TVs, but DVD is also available for storing files. The density of both tracks and bits has been increased to yield a sevenfold increase in storage capacity. DVD is also available in a two-sided medium that yields 10 gigabytes per disc.

3.4.2 CD-ROM as a File Structure Problem

CD-ROM presents interesting file structure problems because it is a medium with great strengths and weaknesses. The strengths of CD-ROM include its high storage capacity, its inexpensive price, and its durability. The key weakness is that seek performance on a CD-ROM is very slow, often taking from a half second to a second per seek. In the introduction to this textbook we compared memory access and magnetic disk access and showed that if memory access is analogous to your taking twenty seconds to look up something in the index to this textbook, the equivalent disk access would take fifty-eight days, or almost 2 months. With a CD-ROM the analogy stretches the disc access to more than *two and a half years!* This kind of performance, or lack of it, makes intelligent file structure design a critical concern for CD-ROM applications. CD-ROM provides an excellent test of our ability to integrate and adapt the principles we have developed in the preceding chapters of this book.

3.5 Physical Organization of CD-ROM

CD-ROM is the child of CD audio. In this instance, the impact of heredity is strong, with both positive and negative aspects. Commercially, the CD audio parentage is probably wholly responsible for CD-ROM's viability. It is because of the enormous CD audio market that it is possible to make these discs so inexpensively. Similarly, advances in the design and decreases in the costs of making CD audio players affect the performance and price of CD-ROM drives. Other optical disc media without the benefits of this parentage have not experienced the commercial success of CD-ROM.

On the other hand, making use of the manufacturing capacity associated with CD audio means adhering to the fundamental physical organization of the CD audio disc. Audio discs are designed to play music, not to provide fast, random access to data. This biases CD toward having high storage capacity and moderate data transfer rates and against decent seek performance. If an application requires good random-access performance, that performance has to emerge from our file structure design efforts; it won't come from anything inherent in the medium.

3.5.1 Reading Pits and Lands

CD-ROMs are stamped from a master disc. The master is formed by using the digital data we want to encode to turn a powerful laser on and off very quickly. The master disc, which is made of glass, has a coating that is changed by the laser beam. When the coating is developed, the areas hit by the laser beam turn into pits along the track followed by the beam. The smooth, unchanged areas between the pits are called *lands.* The copies formed from the master retain this pattern of pits and lands.

When we read the stamped copy of the disc, we focus a beam of laser light on the track as it moves under the optical pickup. The pits scatter the light, but the lands reflect most of it back to the pickup. This alternating pattern of high- and low-intensity reflected light is the signal used to reconstruct the original digital information. The encoding scheme used for this signal is not simply a matter of calling a pit a 1 and a land a 0. Instead, the 1s are represented by the transitions from pit to land and back again. Every time the light intensity changes, we get a 1. The 0s are represented by the amount of time between transitions; the longer between transitions, the more 0s we have.

If you think about this encoding scheme, you realize that it is not possible to have two adjacent 1s—1s are always separated by 0s. In fact, due to the limits of the resolution of the optical pickup, there must be at least two 0s between any pair of 1s. This means that the raw pattern of 1s and 0s has to be translated to get the 8-bit patterns of 1s and 0s that form the bytes of the original data. This translation scheme, which is done through a lookup table, turns the original 8 bits of data into 14 expanded bits that can be represented in the pits and lands on the disc; the reading process reverses this translation. Figure 3.12 shows a portion of the lookup table values. Readers who have looked closely at the specifications for CD players may have encountered the term *EFM* encoding. *EFM* stands for "eight to fourteen modulation" and refers to this translation scheme.

Figure 3.12 A portion of the EFM encoding table.

Decimal value	Original bits	Translated bits
0	00000000	01001000100000
1	00000001	10000100000000
2	00000010	10010000100000
3	00000011	10001000100000
4	00000100	01000100000000
5	00000101	00000100010000
6	00000110	00010000100000
7	00000111	00100100000000
8	00001000	01001001000000

It is important to realize that since we represent the 0s in the EFM-encoded data by the *length of time* between transitions, our ability to read the data is dependent on moving the pits and lands under the optical pick-up at a precise and constant speed. As we will see, this affects the CD-ROM drive's ability to seek quickly.

3.5.2 CLV Instead of CAV

Data on a CD-ROM is stored in a single, spiral track that winds for almost 3 miles from the center to the outer edge of the disc. This spiral pattern is part of the CD-ROM's heritage from CD audio. For audio data, which requires a lot of storage space, we want to pack the data on the disc as tightly as possible. Since we "play" audio data, often from start to finish without interruption, seeking is not important. As Fig. 3.13 shows, a spiral pattern serves these needs well. A sector toward the outer edge of the disc takes the same amount of space as a sector toward the center of the disc. This means that we can write all of the sectors at the maximum density permitted by the storage medium. Since reading the data requires that it pass under the optical pickup device at a constant rate, the constant data density implies that the disc has to spin more slowly when we are reading at the outer edges than when we are reading toward the center. This is why the spiral is a Constant Linear Velocity (CLV) format: as we seek from the center to the edge, we change the rate of rotation of the disc so the linear speed of the spiral past the pickup device stays the same.

By contrast, the familiar Constant Angular Velocity (CAV) arrangement shown in Fig. 3.13, with its concentric tracks and pie-shaped sectors, writes data less densely in the outer tracks than in the tracks toward the center. We are wasting storage capacity in the outer tracks but have the advantage of being able to spin the disc at the same speed for all positions of the read head. Given the sector arrangement shown in the figure, one rotation reads eight sectors, no matter where we are on the disc. Furthermore, a timing mark placed on the disk makes it easy to find the start of a sector.

The CLV format is responsible, in large part, for the poor seeking performance of CD-ROM drives. The CAV format provides definite track boundaries and a timing mark to find the start of a sector. But the CLV format provides no straightforward way to jump to a specific location. Part of the problem is associated with the need to change rotational speed as we seek across the disc. To read the address information that is stored on the disc along with the user's data, we need to be moving the data under

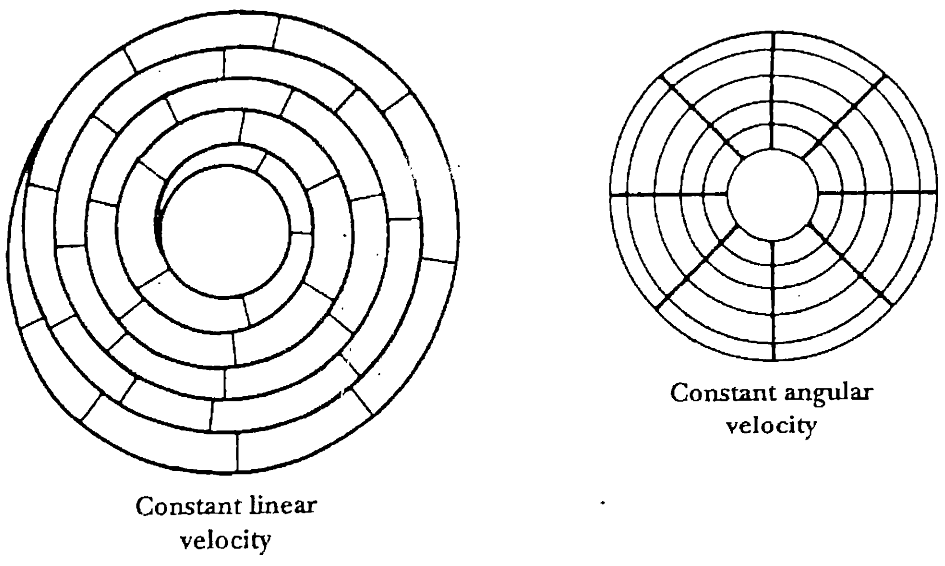


Figure 3.13 CLV and CAV recording.

the optical pickup at the correct speed. But to know how to adjust the speed, we need to be able to read the address information so we know where we are. How does the drive's control mechanism break out of this loop? In practice, the answer often involves making guesses and finding the correct speed through trial and error. This takes time and slows down seek performance.

On the positive side, the CLV sector arrangement contributes to the CD-ROM's large storage capacity. Given a CAV arrangement, the CD-ROM would have only a little better than half its present capacity.

3.5.3 Addressing

The use of the CLV organization means that the familiar cylinder, track, sector method of identifying a sector address will not work on a CD-ROM. Instead, we use a sector-addressing scheme that is related to the CD-ROM's roots as an audio playback device. Each second of playing time on a CD is divided into seventy-five sectors, each of which holds 2 kilobytes of data. According to the original Philips/Sony standard, a CD, whether used for audio or CD-ROM, contains at least one hour of playing time. That means that the disc is capable of holding at least 540 000 kilobytes of data:

60 minutes × 60 seconds/minute × 75 sectors/second = 270 000 sectors

In fact, since it is possible to put more than seventy minutes of playing time on a CD, the capacity of the disk is over 600 megabytes.

We address a given sector by referring to the minute, second, and sector of play. So, the thirty-fourth sector in the twenty-second second in the sixteenth minute of play would be addressed with the three numbers 16:22:34.

3.5.4 Structure of a Sector

It is interesting to see how the fundamental design of the CD disc, initially intended to deliver digital audio information, has been adapted for computer data storage. This investigation will also help answer the question: If the disc is capable of storing a quarter of a million printed pages, why does it hold only an hour's worth of Roy Orbison?

When we want to store sound, we need to convert a wave pattern into digital form. Figure 3.14 shows a wave. At any given point in time, the wave has a specific amplitude. We digitize the wave by measuring the amplitude at very frequent intervals and storing the measurements. So, the question of how much storage space we need to represent a wave digitally turns into two other questions: How much space does it take to store each amplitude sample? How often do we take samples?

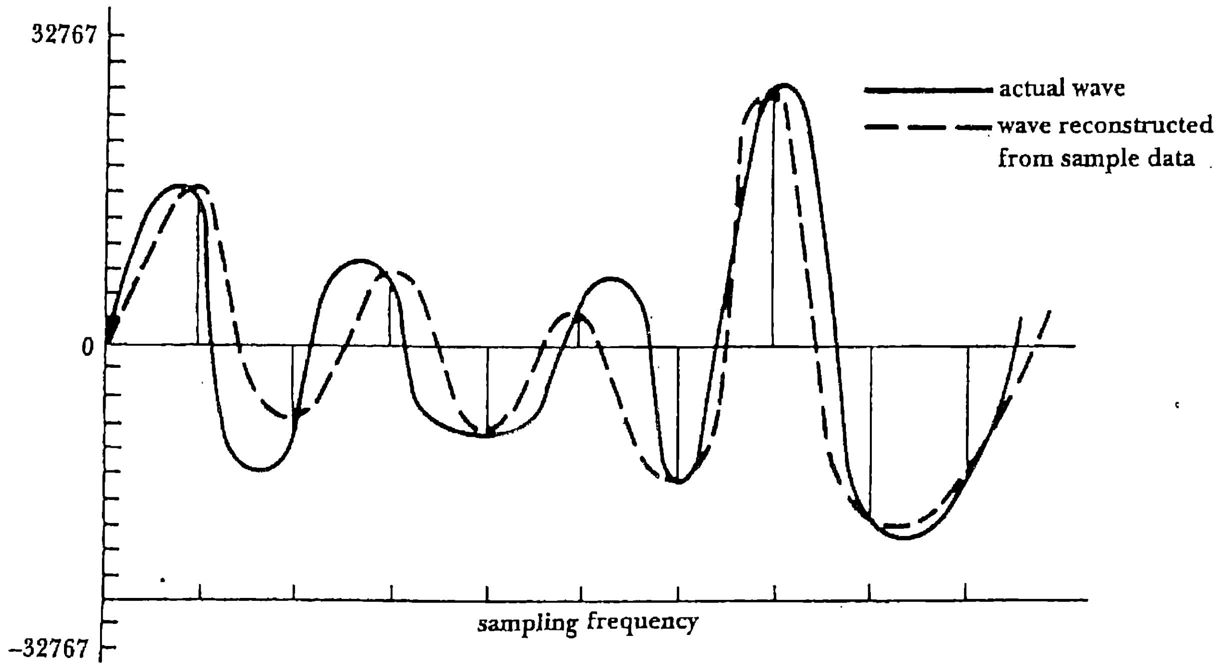


Figure 3.14 Digital sampling of a wave.

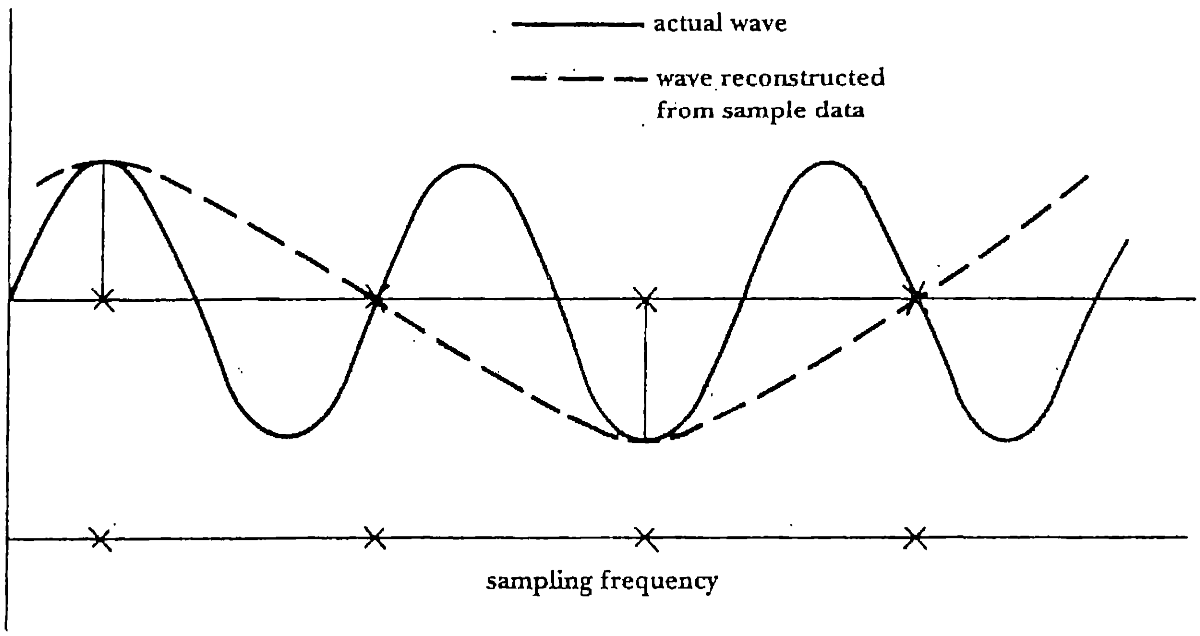


Figure 3.15 The effect of sampling at less than twice the frequency of the wave.

CD audio uses 16 bits to store each amplitude measurement; that means that the "ruler" we use to measure the height of the wave has 65 536 different gradations. To approximate a wave accurately through digital sampling, we need to take the samples at a rate that is more than twice as frequent as the highest frequency we want to capture. This makes sense if you look at the wave in Fig. 3.15. You can see that if we sample at less than twice the frequency of the wave, we lose information about the variation in the wave pattern. The designers of CD audio selected a sampling frequency of 44.1 kilohertz, or 44 100 times per second, so they could record sounds with frequencies ranging up to 20 kilohertz (20 000 cycles per second), which is toward the upper bound of what people can hear.

So, if we are taking a 16-bit, or 2-byte, sample 44 100 times per second, we need to store 88 200 bytes per second. Since we want to store stereo sound, we need double this and store 176 400 bytes per second. You can see why storing an hour of Roy Orbison takes so much space.

If you divide the 176 400-byte-per-second storage capacity of the CD into seventy-five sectors per second, you have 2352 bytes per sector. CD-ROM divides up this "raw" sector storage as shown in Fig. 3.16 to provide 2 kilobytes of user data storage, along with addressing information, error detection, and error correction information. The error correction information is necessary because, although CD audio contains redundancy for error correction, it is not adequate to meet computer data storage needs.

12 bytes synch	4 bytes sector ID	2,048 bytes user data	4 bytes error detection	8 bytes null	276 bytes error correction

Figure 3.16 Structure of a CD-ROM sector.

The audio error correction would result in an average of one incorrect byte for every two discs. The additional error correction information stored within the 2352-byte sector decreases this error rate to 1 uncorrectable byte in every twenty thousand discs.

3.6 CD-ROM Strengths and Weaknesses

As we say throughout this book, good file design is responsive to the nature of the medium, making use of strengths and minimizing weaknesses. We begin, then, by cataloging the strengths and weaknesses of CD-ROM.

3.6.1 Seek Performance

The chief weakness of CD-ROM is the random-access performance. Current magnetic disk technology is such that the average time for a random data access, combining seek time and rotational delay, is about 30 msec. On a CD-ROM, this average access takes 500 msec and can take up to a second or more. Clearly, our file design strategies must avoid seeks to an even greater extent than on magnetic disks.

3.6.2 Data Transfer Rate

A CD-ROM drive reads seventy sectors, or 150 kilobytes of data per second. This data transfer rate is part of the fundamental definition of CD-ROM; it can't be changed without leaving behind the commercial advantages of adhering to the CD audio standard. It is a modest transfer rate, about five times faster than the transfer rate for floppy disks, and an order of magnitude slower than the rate for good Winchester disks. The inadequacy of the transfer rate makes itself felt when we are loading large files, such as those associated with digitized images. On the other hand, the transfer rate is fast enough relative to the CD-ROM's seek performance that we have a design incentive to organize data into blocks, reading more data with each seek in the hope that we can avoid as much seeking as possible.

3.6.3 Storage Capacity

A CD-ROM holds more than 600 megabytes of data. Although it is possible to use up this storage area very quickly, particularly if you are storing raster images, 600 megabytes is big when it comes to text applications. If you decide to download 600 megabytes of text with a 2400-baud modem, it will take about three days of constant data transmission, assuming errorless transmission conditions. Many typical text databases and document collections published on CD-ROM use only a fraction of the disc's capacity.

The design benefit arising from such large capacity is that it enables us to build indexes and other support structures that can help overcome some of the limitations associated with CD-ROM's poor seek performance.

3.6.4 Read-Only Access

From a design standpoint, the fact that CD-ROM is a publishing medium, a storage device that cannot be changed after manufacture, provides significant advantages. We never have to worry about updating. This not only simplifies some of the file structures but also means that it is worthwhile to optimize our index structures and other aspects of file organization. We know that our efforts to optimize access will not be lost through later additions or deletions.

3.6.5 Asymmetric Writing and Reading

For most media, files are written and read using the same computer system. Often, reading and writing are both interactive and are therefore constrained by the need to provide quick response to the user. CD-ROM is different. We create the files to be placed on the disc once; then we distribute the disc, and it is accessed thousands, even millions, of times. We are in a position to bring substantial computing power to the task of file organization and creation, even when the disc will be used on systems with much less capability. In fact, we can use extensive batch-mode processing on large computers to try to provide systems that will perform well on small machines. We make the investment in intelligent, carefully designed file structures only once; users can enjoy the benefits of this investment again and again.

3.7 Storage as a Hierarchy

Although the best mixture of devices for a computing system depends on the needs of the system's users, we can imagine any computing system as a hierarchy of storage devices of different speed, capacity, and cost. Figure 3.17 summarizes the different types of storage found at different levels in such hierarchies and shows approximately how they compare in terms of access time, capacity, and cost.

Types of memory	Devices and media	Access times (sec)	Capacities (bytes)	Cost (Cents/bit)
Primary				
Registers Memory RAM disk and disk cache	Semiconductors	$10^{-9} - 10^{-5}$	$10^{0} - 10^{9}$	$10^{0} - 10^{-3}$
Secondary				
Direct-access	Magnetic disks	$10^{-3} - 10^{-1}$	$10^{4} - 10^{9}$	$10^{-2} - 10^{-5}$
Serial	Tape and mass storage	$10^{1} - 10^{2}$	$10^{0} - 10^{11}$	$10^{-5} - 10^{-7}$
Offline				
Archival and backup	Removable magnetic disks, optical discs, and tapes	$10^{0} - 10^{2}$	$10^{4} - 10^{12}$	$10^{-5} - 10^{-7}$

Figure 3.17 Approximate comparisons of types of storage.

3.8 A Journey of a Byte

What happens when a program writes a byte to a file on a disk? We know what the program does (it makes a call to a write function), and we now know something about how the byte is stored on a disk. But we haven't looked at what happens *between* the program and the disk. The whole story of what happens to data between program and disk is not one we can tell here, but we can give you an idea of the many different pieces of hardware and software involved and the many jobs that have to be done by looking at an example of a journey of 1 byte.

Suppose we want to append a byte representing the character *P* stored in a character variable `ch` to a file named in the variable `textfile` stored somewhere on a disk. From the program's point of view, the entire journey that the byte will take might be represented by the statement

```
write(textfile, ch, 1)
```

but the journey is much longer than this simple statement suggests.

The `write` statement results in a call to the computer's operating system, which has the task of seeing that the rest of the journey is completed successfully (Fig. 3.18). Often our program can provide the operating system with information that helps it carry out this task more effectively, but once the operating system has taken over, the job of overseeing the rest of the journey is largely beyond our program's control.

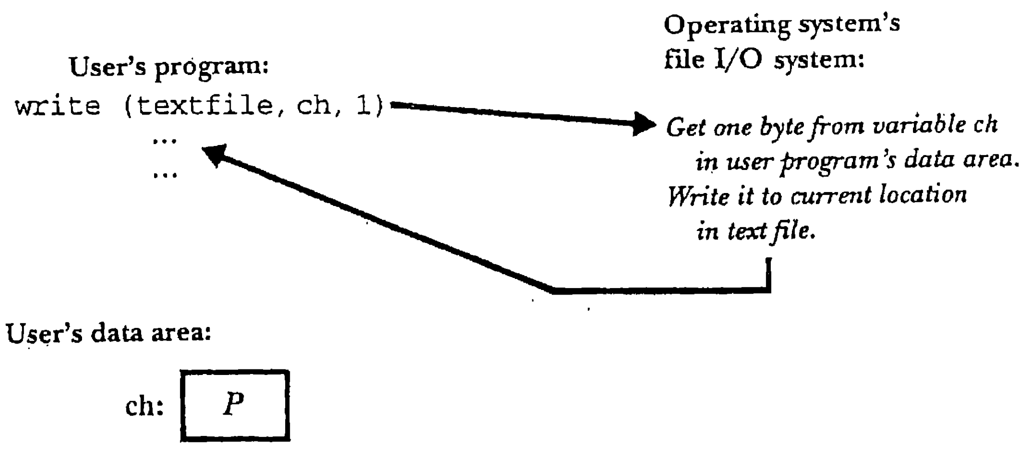
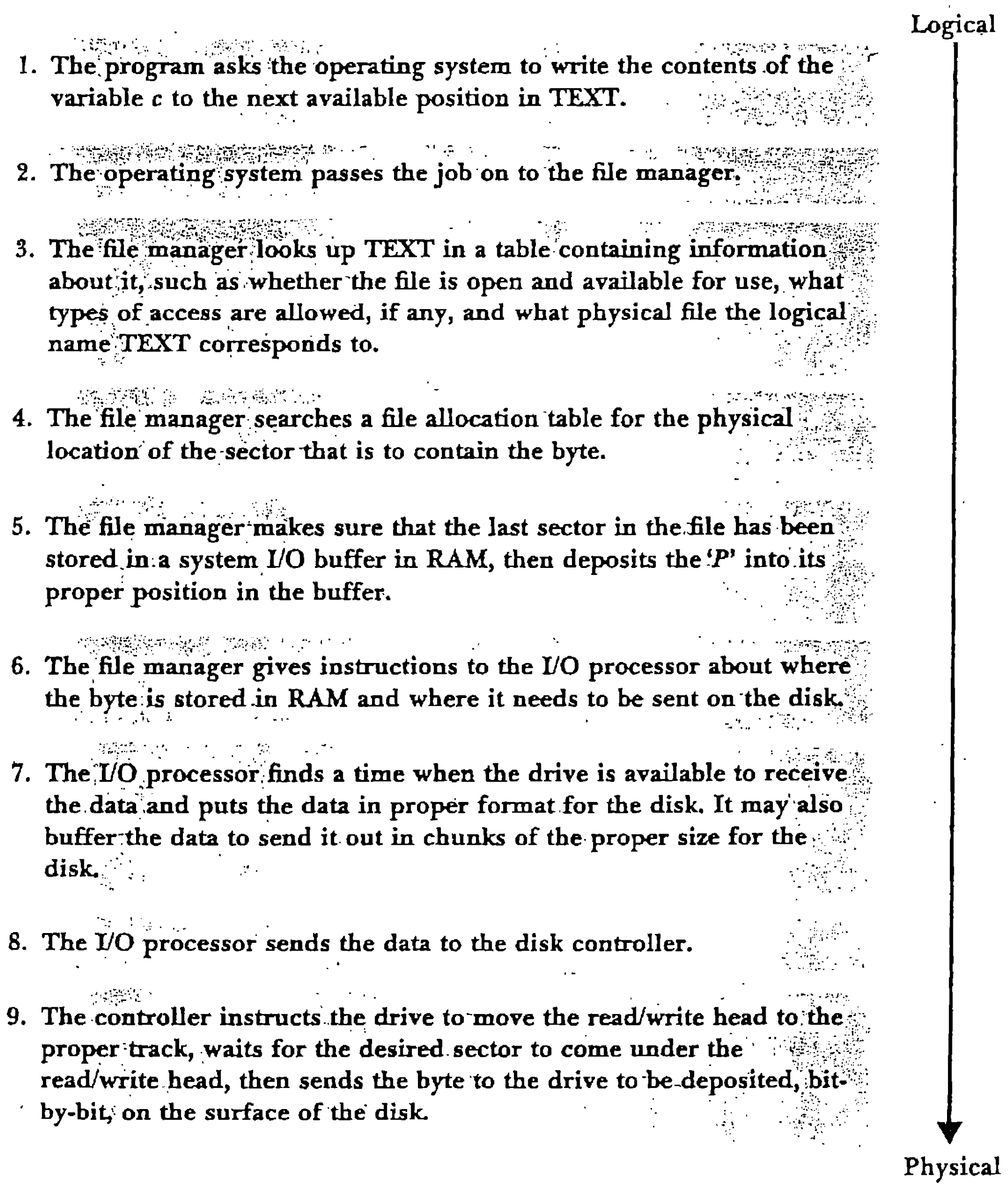

User's program:
write (textfile, ch, 1)
...
...
Operating system's file I/O system:
Get one byte from variable ch in user program's data area.
Write it to current location in textfile.
User's data area:
ch:
P

Figure 3.18 The write statement tells the operating system to send one character to disk and gives the operating system the location of the character. The operating system takes over the job of writing, and then returns control to the calling program.

3.8.1 The File Manager

An operating system is not a single program but a collection of programs, each one designed to manage a different part of the computer's resources. Among these programs are ones that deal with file-related matters and I/O devices. We call this subset of programs the operating system's *file manager*. The file manager may be thought of as several layers of procedures (Fig. 3.19), with the upper layers dealing mostly with symbolic, or *logical*, aspects of file management, and the lower layers dealing more with the



1. The program asks the operating system to write the contents of the variable *c* to the next available position in TEXT.
2. The operating system passes the job on to the file manager.
3. The file manager looks up TEXT in a table containing information about it, such as whether the file is open and available for use, what types of access are allowed, if any, and what physical file the logical name TEXT corresponds to.
4. The file manager searches a file allocation table for the physical location of the sector that is to contain the byte.
5. The file manager makes sure that the last sector in the file has been stored in a system I/O buffer in RAM, then deposits the '*P*' into its proper position in the buffer.
6. The file manager gives instructions to the I/O processor about where the byte is stored in RAM and where it needs to be sent on the disk.
7. The I/O processor finds a time when the drive is available to receive the data and puts the data in proper format for the disk. It may also buffer the data to send it out in chunks of the proper size for the disk.
8. The I/O processor sends the data to the disk controller.
9. The controller instructs the drive to move the read/write head to the proper track, waits for the desired sector to come under the read/write head, then sends the byte to the drive to be deposited, bit-by-bit, on the surface of the disk.

Figure 3.19 Layers of procedures involved in transmitting a byte from a program's data area to a file called *textfile* on disk.

physical aspects. Each layer calls the one below it, until, at the lowest level, the byte is written to the disk.

. The file manager begins by finding out whether the logical characteristics of the file are consistent with what we are asking it to do with the file. It may look up the requested file in a table, where it finds out such things as whether the file has been opened, what type of file the byte is being sent to (a binary file, a text file, or some other organization), who the file's owner is, and whether write access is allowed for this particular user of the file.

The file manager must also determine where in the file `textfile` the *P* is to be deposited. Since the *P* is to be appended to the file, the file manager needs to know where the end of the file is—the physical location of the last sector in the file. This information is obtained from the file allocation table (FAT) described earlier. From the FAT, the file manager locates the drive, cylinder, track, and sector where the byte is to be stored.

3.8.2 The I/O Buffer

Next, the file manager determines whether the sector that is to contain the *P* is already in memory or needs to be loaded into memory. If the sector needs to be loaded, the file manager must find an available *system I/O buffer* space for it and then read it from the disk. Once it has the sector in a buffer in memory, the file manager can deposit the *P* into its proper position in the buffer (Fig. 3.20). The system I/O buffer allows the file manager to read and write data in sector-sized or block-sized units. In other words, it enables the file manager to ensure that the organization of data in memory conforms to the organization it will have on the disk.

Instead of sending the sector immediately to the disk, the file manager usually waits to see if it can accumulate more bytes going to the same sector before transmitting anything. Even though the statement `write(textfile,ch,1)` seems to imply that our character is being sent immediately to the disk, it may in fact be kept in memory for some time before it is sent. (There are many situations in which the file manager cannot wait until a buffer is filled before transmitting it. For instance, if `textfile` were closed, it would have to flush all output buffers holding data waiting to be written to `textfile` so the data would not be lost.)

3.8.3 The Byte Leaves Memory: The I/O Processor and Disk Controller

So far, all of our byte's activities have occurred within the computer's primary memory and have probably been carried out by the computer's

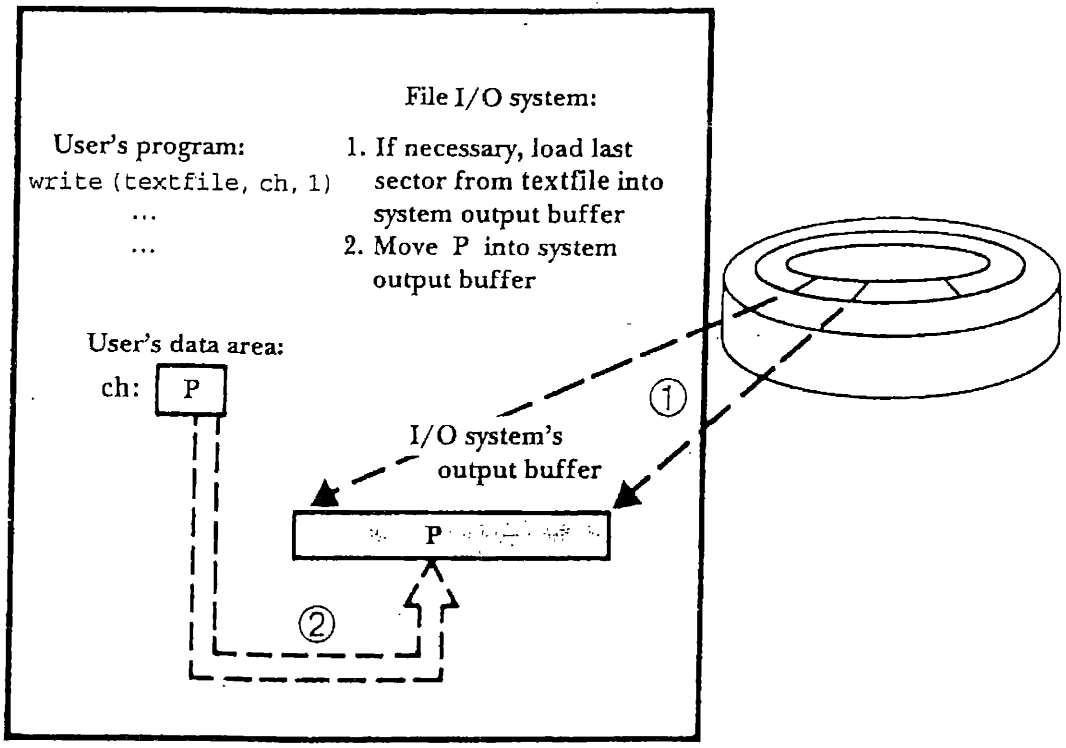

File I/O system:
1. If necessary, load last sector from textfile into system output buffer
2. Move P into system output buffer
User's program:
write (textfile, ch, 1)
...
...
User's data area:
ch: P
①
I/O system's output buffer
P
②

Figure 3.20 The file manager moves *P* from the program's data area to a system output buffer where it may join other bytes headed for the same place on the disk. If necessary, the file manager may have to load the corresponding sector from the disk into the system output buffer.

central processing unit. The byte has traveled along data paths that are designed to be very fast and are relatively expensive. Now it is time for the byte to travel along a data path that is likely to be slower and narrower than the one in primary memory. (A typical computer might have an internal data-path width of 4 bytes, whereas the width of the path leading to the disk might be only 2 bytes.)

Because of bottlenecks created by these differences in speed and data-path widths, our byte and its companions might have to wait for an external data path to become available. This also means that the CPU has extra time on its hands as it deals out information in small enough chunks and at slow enough speeds that the world outside can handle them. In fact, the differences between the internal and external speeds for transmitting data are often so great that the CPU can transmit to several external devices simultaneously.

The processes of disassembling and assembling groups of bytes for transmission to and from external devices are so specialized that it is unreasonable to ask an expensive, general-purpose CPU to spend its valuable time doing I/O when a simpler device could do the job and free the CPU to do the work that it is most suited for. Such a special-purpose device is called an *I/O processor.*

An I/O processor may be anything from a simple chip capable of taking a byte and passing it along one cue, to a powerful, small computer capable of executing very sophisticated programs and communicating with many devices simultaneously. The I/O processor takes its instructions from the operating system, but once it begins processing I/O, it runs independently, relieving the operating system (and the CPU) of the task of communicating with secondary storage devices. This allows I/O processes and internal computing to overlap.[7]

In a typical computer, the file manager might now tell the I/O processor that there is data in the buffer to be transmitted to the disk, how much data there is, and where it is to go on the disk. This information might come in the form of a little program that the operating system constructs and the I/O processor executes (Fig. 3.21).

The job of controlling the operation of the disk is done by a device called a *disk controller.* The I/O processor asks the disk controller if the disk drive is available for writing. If there is much I/O processing, there is a good chance that the drive will not be available and that our byte will have to wait in its buffer until the drive becomes available.

What happens next often makes the time spent so far seem insignificant in comparison: the disk drive is instructed to move its read/write head to the track and sector on the drive where our byte and its companions are to be stored. For the first time, a device is being asked to do something mechanical! The read/write head must seek to the proper track (unless it is already there) and then wait until the disk has spun around so the desired sector is under the head. Once the track and sector are located, the I/O processor (or perhaps the controller) can send out bytes, one at a time, to the drive. Our byte waits until its turn comes; then it travels alone to the drive, where it probably is stored in a little 1-byte buffer while it waits to be deposited on the disk.

Finally, as the disk spins under the read/write head, the 8 bits of our byte are deposited, one at a time, on the surface of the disk (Fig. 3.21). There the *P* remains, at the end of its journey, spinning at a leisurely 50 to 100 miles per hour.

7. On many systems the I/O processor can take data directly from memory, without further involvement from the CPU. This process is called *direct memory access* (DMA). On other systems, the CPU must place the data in special I/O registers before the I/O processor can have access to it.

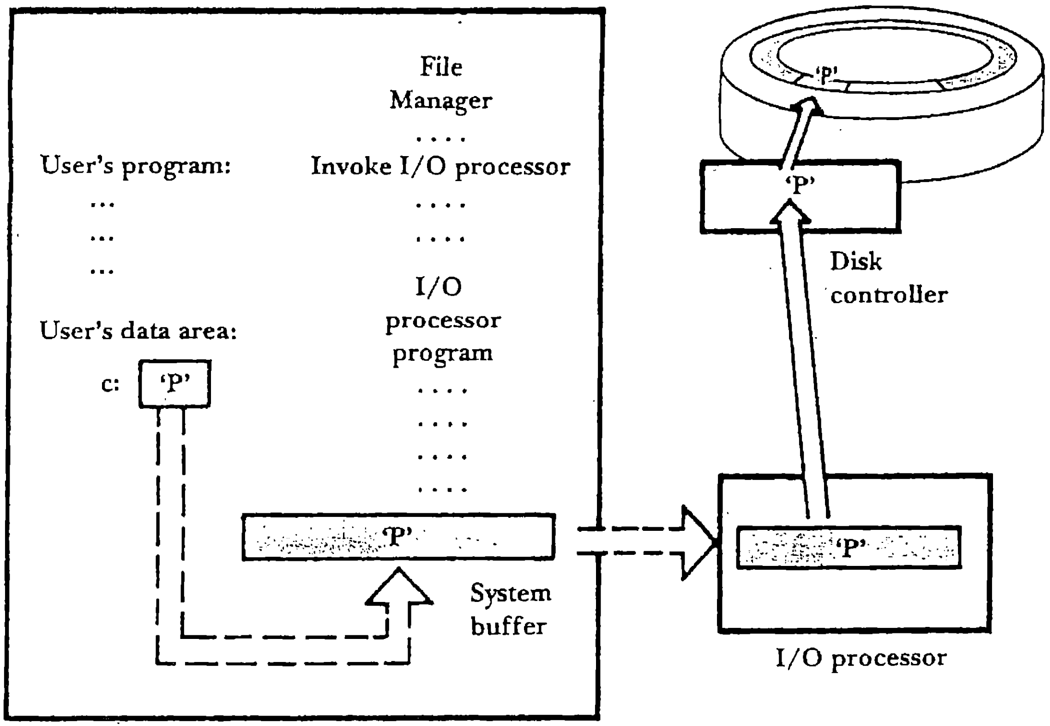


Figure 3.21 The file manager sends the I/O processor instructions in the form of an I/O processor program. The I/O processor gets the data from the system buffer, prepares it for storing on the disk, then sends it to the disk controller, which deposits it on the surface of the disk.

3.9 Buffer Management

Any user of files can benefit from some knowledge of what happens to data traveling between a program's data area and secondary storage. One aspect of this process that is particularly important is the use of buffers. Buffering involves working with large chunks of data in memory so the number of accesses to secondary storage can be reduced. We concentrate on the operation of *system* I/O buffers; but be aware that the use of buffers within programs can also substantially affect performance.

3.9.1 Buffer Bottlenecks

We know that a file manager allocates I/O buffers that are big enough to hold incoming data, but we have said nothing so far about *how many* buffers are used. In fact, it is common for file managers to allocate several buffers for performing I/O.

To understand the need for several system buffers, consider what happens if a program is performing both input and output on one character at a time and only one I/O buffer is available. When the program asks for its first character, the I/O buffer is loaded with the sector containing the character, and the character is transmitted to the program. If the program then decides to output a character, the I/O buffer is filled with the sector into which the output character needs to go, destroying its original contents. Then when the next input character is needed, the buffer contents have to be written to disk to make room for the (original) sector containing the second input character, and so on.

Fortunately, there is a simple and generally effective solution to this ridiculous state of affairs, and that is to use more than one system buffer. For this reason, I/O systems almost always use at least two buffers—one for input and one for output.

Even if a program transmits data in only one direction, the use of a single system I/O buffer can slow it down considerably. We know, for instance, that the operation of reading a sector from a disk is extremely slow compared with the amount of time it takes to move data in memory, so we can guess that a program that reads many sectors from a file might have to spend much of its time waiting for the I/O system to fill its buffer every time a read operation is performed before it can begin processing. When this happens, the program that is running is said to be *I/O bound*—the CPU spends much of its time just waiting for I/O to be performed. The solution to this problem is to use more than one buffer and to have the I/O system filling the next sector or block of data while the CPU is processing the current one.

3.9.2 Buffering Strategies

Multiple Buffering

Suppose that a program is only writing to a disk and that it is I/O bound. The CPU wants to be filling a buffer at the same time that I/O is being performed. If *two* buffers are used and I/O-CPU overlapping is permitted, the CPU can be filling one buffer while the contents of the other are being transmitted to disk. When both tasks are finished, the roles of the buffers can be exchanged. This method of swapping the roles of two buffers after each output (or input) operation is called *double buffering*. Double buffering allows the operating system to operate on one buffer while the other buffer is being loaded or emptied (Fig. 3.22).

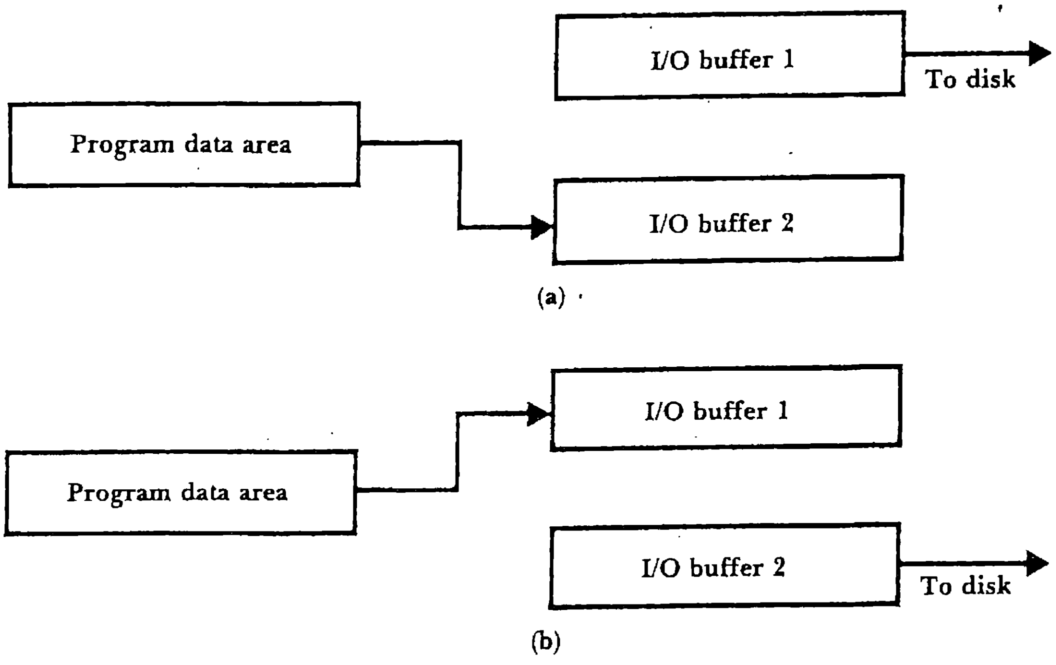


Figure 3.22 Double buffering: (a) the contents of system I/O buffer 1 are sent to disk while I/O buffer 2 is being filled; and (b) the contents of buffer 2 are sent to disk while I/O buffer 1 is being filled.

This technique of swapping system buffers to allow processing and I/O to overlap need not be restricted to two buffers. In theory, any number of buffers can be used, and they can be organized in a variety of ways. The actual management of system buffers is usually done by the operating system and can rarely be controlled by programmers who do not work at the systems level. It is common, however, for programmers to be able to control the *number* of system buffers assigned to jobs.

Some file systems use a buffering scheme called *buffer pooling:* when a system buffer is needed, it is taken from a pool of available buffers and used. When the system receives a request to read a certain sector or block, it looks to see if one of its buffers already contains that sector or block. If no buffer contains it, the system finds from its pool of buffers one that is not currently in use and loads the sector or block into it.

Several different schemes are used to decide which buffer to take from a buffer pool. One generally effective strategy is to take the buffer that is *least recently used.* When a buffer is accessed, it is put on a least-recently-used queue so it is allowed to retain its data until all other less-recently-used buffers have been accessed. The least-recently-used (LRU) strategy for replacing old data with new data has many applications in computing.

It is based on the assumption that a block of data that has been used recently is more likely to be needed in the near future than one that has been used less recently. (We encounter LRU again in later chapters.)

It is difficult to predict the point at which the addition of extra buffers ceases to contribute to improved performance. As the cost of memory continues to decrease, so does the cost of using more and bigger buffers. On the other hand, the more buffers there are, the more time it takes for the file system to manage them. When in doubt, consider experimenting with different numbers of buffers.

Move Mode and Locate Mode

Sometimes it is not necessary to distinguish between a program's data area and system buffers. When data must always be copied from a system buffer to a program buffer (or vice versa), the amount of time taken to perform the move can be substantial. This way of handling buffered data is called *move mode*, as it involves moving chunks of data from one place in memory to another before they can be accessed.

There are two ways that move mode can be avoided. If the file manager can perform I/O directly between secondary storage and the program's data area, no extra move is necessary. Alternatively, the file manager could use system buffers to handle all I/O but provide the program with the *locations*, using pointer variables, of the system buffers. Both techniques are examples of a general approach to buffering called *locate mode*. When locate mode is used, a program is able to operate directly on data in the I/O buffer, eliminating the need to transfer data between an I/O buffer and a program buffer.

Scatter/Gather I/O

Suppose you are reading in a file with many blocks, and each block consists of a header followed by data. You would like to put the headers in one buffer and the data in a different buffer so the data can be processed as a single entity. The obvious way to do this is to read the whole block into a single big buffer; then move the different parts to their own buffers. Sometimes we can avoid this two-step process using a technique called *scatter input*. With scatter input, a single read call identifies not one, but a collection of buffers into which data from a single block is to be scattered.

The converse of scatter input is *gather output*. With gather output, several buffers can be gathered and written with a single write call; this avoids the need to copy them to a single output buffer. When the cost of copying several buffers into a single output buffer is high, scatter/gather can have a significant effect on the running time of a program.

It is not always obvious when features like scatter/gather, locate mode, and buffer pooling are available in an operating system. You often have to go looking for them. Sometimes you can invoke them by communicating with your operating system, and sometimes you can cause them to be invoked by organizing your program in ways that are compatible with the way the operating system does I/O. Throughout this text we return many times to the issue of how to enhance performance by thinking about how buffers work and adapting programs and file structures accordingly.

3.10 I/O in Unix

We see in the journey of a byte that we can view I/O as proceeding through several layers. Unix provides a good example of how these layers occur in a real operating system, so we conclude this chapter with a look at Unix. It is of course beyond the scope of this text to describe the Unix I/O layers in detail. Rather, our objective here is just to pick a few features of Unix that illustrate points made in the text. A secondary objective is to familiarize you with some of the important terminology used in describing Unix systems. For a comprehensive, detailed look at how Unix works, plus a thorough discussion of the design decisions involved in creating and improving Unix, see Leffler et al. (1989).

3.10.1 The Kernel

In Figure 3.19 we see how the process of transmitting data from a program to an external device can be described as proceeding through a series of layers. The topmost layer deals with data in *logical*, structural terms. We store in a file a name, a body of text, an image, an array of numbers, or some other logical entity. This reflects the view that an application has of what goes into a file. The layers that follow collectively carry out the task of turning the logical object into a collection of bits on a *physical* device.

Likewise, the topmost I/O layer in Unix deals with data primarily in logical terms. This layer in Unix consists of *processes* that impose certain logical views on files. Processes are associated with solving some problem, such as counting the words in the file or searching for somebody's address. Processes include *shell routines* like `cat` and `tail`, *user programs* that operate on files, and *library routines* like `scanf` and `fread` that are called from programs to read strings, numbers, and so on.

Below this layer is the Unix *kernel*, which incorporates all the rest of the layers.[8] The components of the kernel that do I/O are illustrated in Figure 3.23. The kernel views all I/O as operating on a sequence of bytes, so once we pass control to the kernel all assumptions about the logical view of a file are gone. The decision to design Unix this way—to make all operations below the top layer independent of an application's logical view of a file—is unusual. It is also one of the main attractions in choosing Unix as a focus for this text, for Unix lets us make all of the decisions about the logical structure of a file, imposing no restrictions on how we think about the file beyond the fact that it must be built from a sequence of bytes.

8. It is beyond the scope of this text to describe the Unix kernel in detail. For a full description of the Unix kernel, including the I/O system, see Leffler et al. (1989).

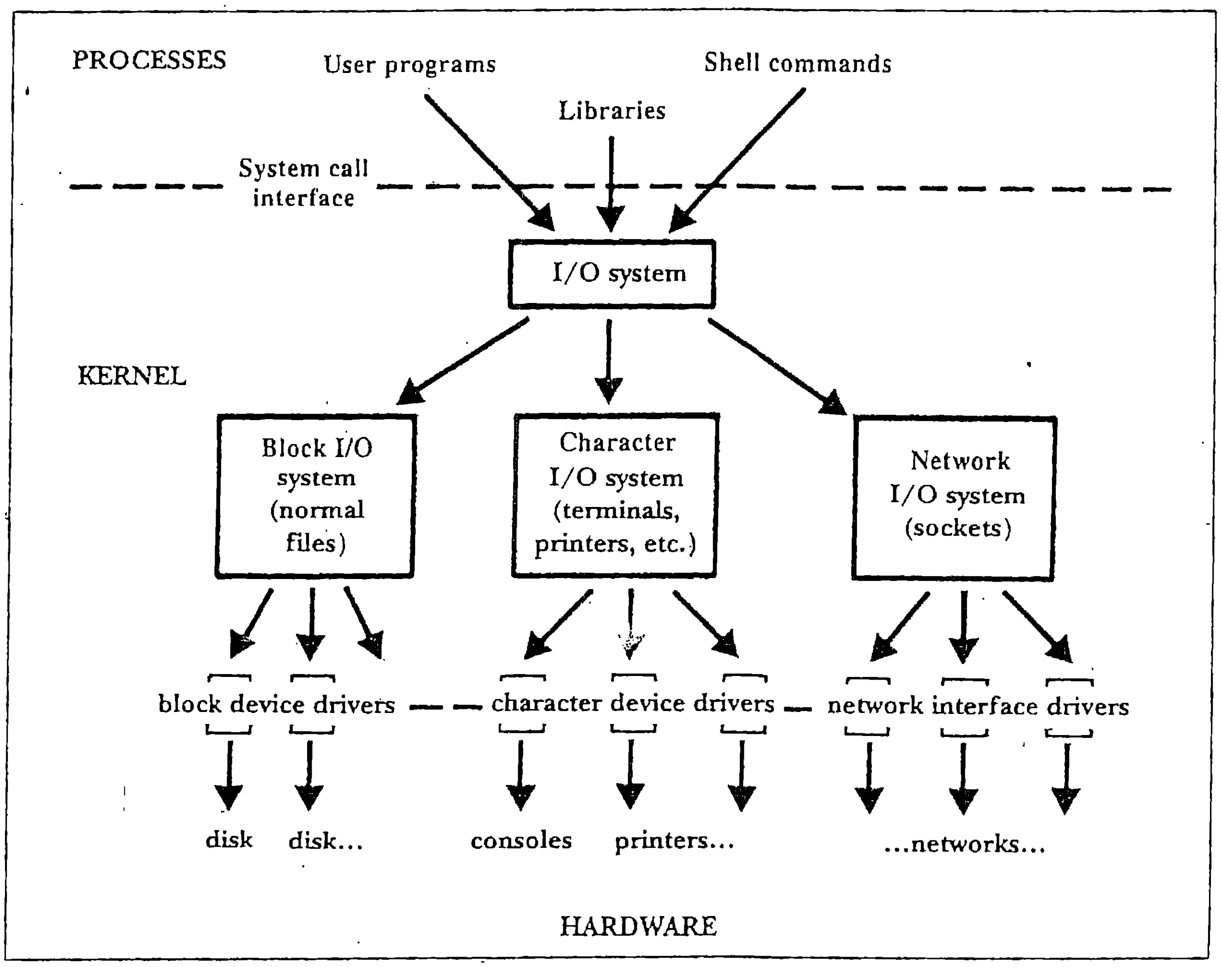


Figure 3.23 Kernel I/O structure.

Let's illustrate the journey of a byte through the kernel, as we did earlier in this chapter by tracing the results of an I/O statement. We assume in this example that we are writing a character to disk. This corresponds to the left branch of the I/O system in Fig. 3.23.

When your program executes a system call such as

```
write (fd, &ch, 1);
```

the kernel is invoked immediately.[9] The routines that let processes communicate directly with the kernel make up the *system call interface.* In this case, the system call instructs the kernel to write a character to a file.

The kernel I/O system begins by connecting the file descriptor (`fd`) in your program to some file or device in the file system. It does this by proceeding through a series of four tables that enable the kernel to find its way from a process to the places on the disk that will hold the file they refer to. The four tables are

- a file descriptor table;
- an open file table, with information about open files;
- a file allocation table, which is part of a structure called an index node; and
- a table of index nodes, with one entry for each file in use.

Although these tables are managed by the kernel's I/O system, they are, in a sense, "owned" by different parts of the system:

- The file descriptor table is owned by the process (your program).
- The open file table and index node table are owned by the kernel.
- The index node is part of the file system.

The four tables are invoked in turn by the kernel to get the information it needs to write to your file on disk. Let's see how this works by looking at the functions of the tables.

The *file descriptor table* (Fig. 3.24a) is a simple table that associates each of the file descriptors used by a process with an entry in another table, the open file table. Every process has its own descriptor table, which includes entries for all files it has opened, including the "files" `stdin`, `stdout`, and `stderr`.

9. This should not be confused with a *library* call, such as *fprintf,* which invokes the standard library to perform some additional operations on the data, such as converting it to an ASCII format, and *then* makes a corresponding system call.

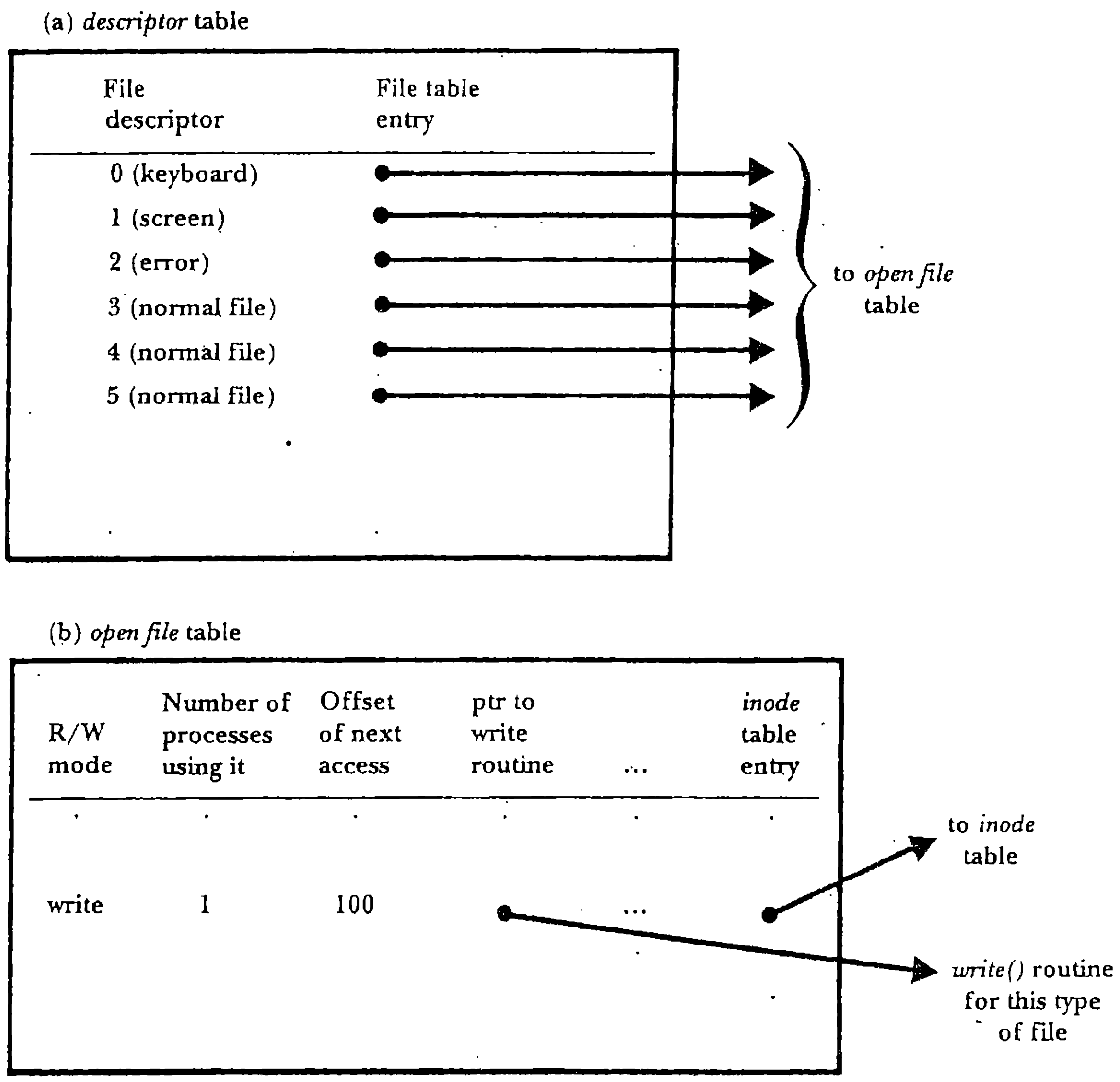


Figure 3.24 Descriptor table and open file table.

The *open file* table (Fig. 3.24b) contains entries for every open file. Every time a file is opened or created, a new entry is added to the open file table. These entries are called *file structures,* and they contain important information about how the corresponding file is to be used, such as the read/write mode used when it was opened, the number of processes currently using it, and the offset within the file to be used for the next read or write. The open file table also contains an array of pointers to generic functions that can be used to operate on the file. These functions will differ depending on the type of file.

It is possible for several different processes to refer to the same open file table entry so one process could read part of a file, another process could read the next part, and so forth, with each process taking over where the previous one stopped. On the other hand, if the same file is opened by two separate open statements, two separate entries are made in the table, and the two processes operate on the file quite independently.[10]

The information in the open file table is transitory. It tells the kernel what it can do with a file that has been opened in a certain way and provides information on how it can operate on the file. The kernel still needs more information about the file, such as where the file is stored on disk, how big the file is, and who owns it. This information is found in an *index node*, more commonly referred to as an *inode* (Fig. 3.25).

An inode is a more permanent structure than an open file table's file structure. A file structure exists only while a file is open for access, but an inode exists as long as its corresponding file exists. For this reason, a file's inode is kept on disk *with* the file (though not physically adjacent to the file). When a file is opened, a copy of its inode is usually loaded into memory where it is added to the aforementioned *inode table* for rapid access.

For the purposes of our discussion, the most important component of the inode is a list (index) of the disk blocks that make up the file. This list is the Unix counterpart to the file allocation table that we described earlier in this chapter.[11] Once the kernel's I/O system has the inode information, it knows all it needs to know about the file. It then invokes an I/O processor program that is appropriate for the type of data, the type of operation, and the type of device that is to be written. In Unix, this program is called a *device driver.*

The device driver sees that your data is moved from its buffer to its proper place on disk. Before we look at the role of device drivers in Unix, it is instructive to look at how the kernel distinguishes among the different kinds of file data it must deal with.

3.10.2 Linking File Names to Files

It is instructive to look a little more closely at how a file name is linked to the corresponding file. All references to files begin with a directory, for it is

10. Of course, there are risks in letting this happen. If you are writing to a file with one process at the same time that you are independently reading from the file with another, the meaning of these may be difficult to determine.

11. This might not be a simple linear array. To accommodate both large and small files, this table often has a dynamic, tree-like structure.

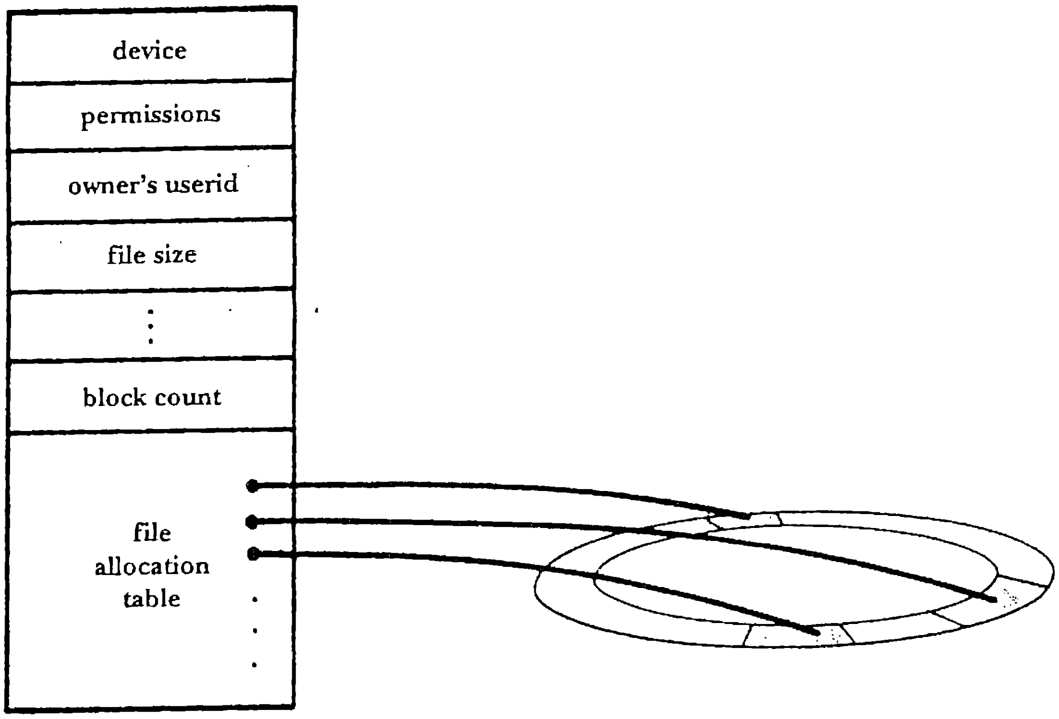


Figure 3.25 An inode. The inode is the data structure used by Unix to describe the file. It includes the device containing the file, permissions, owner and group IDs, and file allocation table, among other things.

in directories that file names are kept. In fact, a directory is just a small file that contains, for each file, a file name together with a pointer to the file's inode on disk.[12] This pointer from a directory to the inode of a file is called a *hard link*. It provides a direct reference from the file name to all other information about the file. When a file is opened, this hard link is used to bring the inode into memory and to set up the corresponding entry in the open file table.

It is possible for several file names to point to the same inode, so one file can have several different names. A field in the inode tells how many hard links there are to the inode. This means that if a file name is deleted and there are other file names for the same file, the file itself is not deleted; its inode's hard-link count is just decremented by one.

There is another kind of link, called a *soft link*, or *symbolic link*. A symbolic link links a file name to another file name rather than to an actu-

12. The actual structure of a directory is a little more complex than this, but these are the essential parts. See Leffler, et al. (1989) for details.

al file. Instead of being a pointer to an inode, a soft link is a pathname of some file. Since a symbolic link does not point to an actual file, it can refer to a directory or even to a file in a different file system. Symbolic links are not supported on all Unix systems. Unix System 4.3BSD supports symbolic links, but System V does not.

3.10.3 Normal Files, Special Files, and Sockets

The "everything is a file" concept in Unix works only when we recognize that some files are quite a bit different from others. We see in Fig. 3.23 that the kernel distinguishes among three different types of files. *Normal files* are the files that this text is about. *Special files* almost always represent a stream of characters and control signals that drive some device, such as a line printer or a graphics device. The first three file descriptors in the descriptor table (Fig. 3.24a) are special files. *Sockets* are abstractions that serve as endpoints for interprocess communication.

At a certain conceptual level, these three different types of Unix files are very similar, and many of the same routines can be used to access any of them. For instance, you can establish access to all three types by opening them, and you can write to them with the write system call.

3.10.4 Block I/O

In Fig. 3.23, we see that the three different types of files access their respective devices via three different I/O systems: the *block I/O system*, the *character I/O system*, and the *network I/O system*. Henceforth we ignore the second and third categories, since it is normal file I/O that we are most concerned with in this text.[13]

The block I/O system is the Unix counterpart of the file manager in the journey of a byte. It concerns itself with how to transmit normal file data, viewed by the user as a sequence of bytes, onto a block-oriented device like a disk or tape. Given a byte to store on a disk, for example, it arranges to read in the sector containing the byte to be replaced, to replace the byte, and to write the sector back to the disk.

The Unix view of a block device most closely resembles that of a disk. It is a randomly addressable array of fixed blocks. Originally, all blocks

13. This is not entirely true. Sockets, for example, can be used to move normal files from place to place. In fact, high-performance network systems bypass the normal file system in favor of sockets to squeeze every bit of performance out of the network.

were 512 bytes, which was the common sector size on most disks. No other organization (such as clusters) was imposed on the placement of files on disk. (In Section 3.1.7 we saw how the design of later Unix systems dealt with this convention.)

3.10.5 Device Drivers

For each peripheral device there is a separate set of routines, called a *device driver*, that performs the I/O between the I/O buffer and the device. A device driver is roughly equivalent to the I/O processor program described in the journey of a byte.

Since the block I/O system views a peripheral device as an array of physical blocks, addressed as block 0, block 1, and so on, a block I/O device driver's job is to take a block from a buffer, destined for one of these physical blocks, and see that it gets deposited in the proper physical place on the device. This saves the block I/O part of the kernel from having to know anything about the specific device it is writing to, other than its identity and that it is a block device. A thorough discussion of device drivers for block, character, and network I/O can be found in Leffler et al. (1989).

3.10.6 The Kernel and File Systems

In Chapter 2 we described the Unix concept of a *file system*. A Unix file system is a collection of files, together with secondary information about the files in the system. A file system includes the directory structure, the directories, ordinary files, and the inodes that describe the files.

In our discussions we talk about the file system as if it is part of the kernel's I/O system, which it is, but it is also in a sense separate from it. All parts of a file system reside on disk, rather than in memory where the kernel does its work. These parts are brought into memory by the kernel as needed. This separation of the file system from the kernel has many advantages. One important advantage is that we can tune a file system to a particular device or usage pattern independently of how the kernel views files. The discussions of BSD Unix block organization in Section 3.1.7 are file-system concerns, for example, and need not have any effect on how the kernel works.

Another advantage of keeping the file system and I/O system distinct is that we can have separate file systems that are organized differently, perhaps on different devices, but are accessible by the same kernel. In Appendix A, for instance, we describe the design of a file system on CD-ROM that is organized quite differently from a typical disk-based file system yet looks just like any other file system to the user and to the I/O system.

3.10.7 Magnetic Tape and Unix

Important as it is to computing, magnetic tape is somewhat of an orphan in the Unix view of I/O. A magnetic tape unit has characteristics similar to both block I/O devices (block oriented) and character devices (primarily used for sequential access) but does not fit nicely into either category. Character devices read and write streams of data, not blocks, and block devices in general access blocks randomly, not sequentially.

Since block I/O is generally the less inappropriate of the two inappropriate paradigms for tape, a tape device is normally considered in Unix to be a block I/O device and hence is accessed through the block I/O interface. But because the block I/O interface is most often used to write to random-access devices, disks, it does not require blocks to be written in sequence, as they must be written to a tape. This problem is solved by allowing only one write request at a time per tape drive. When high-performance I/O is required, the character device interface can be used in a raw mode to stream data to tapes, bypassing the stage that requires the data to be collected into relatively small blocks before or after transmission.

SUMMARY

In this chapter we look at the software environment in which file processing programs must operate and at some of the hardware devices on which files are commonly stored, hoping to understand how they influence the ways we design and process files. We begin by looking at the two most common storage media: magnetic disks and tapes.

A disk drive consists of a set of read/write heads that are interspersed among one or more platters. Each platter contributes one or two surfaces, each surface contains a set of concentric tracks, and each track is divided into sectors or blocks. The set of tracks that can be read without moving the read/write heads is called a cylinder.

There are two basic ways to address data on disks: by sector and by block. Used in this context, the term *block* refers to a group of records that are stored together on a disk and treated as a unit for I/O purposes. When blocks are used, the user is better able to make the physical organization of data correspond to its logical organization, and hence can sometimes improve performance. Block-organized drives also sometimes make it possible for the disk drive to search among blocks on a track for a record with a certain key without first having to transmit the unwanted blocks into memory.

Three possible disadvantages of block-organized devices are the danger of internal track fragmentation, the burden of dealing with the extra complexity that the user has to bear, and the loss of opportunities to do some of the kinds of synchronization (such as sector interleaving) that sector-addressing devices provide.

The cost of a disk access can be measured in terms of the time it takes for seeking, rotational delay, and transfer time. If sector interleaving is used, it is possible to access logically adjacent sectors by separating them physically by one or more sectors. Although it takes much less time to access a single record directly than sequentially, the extra seek time required for doing direct accesses makes it much slower than sequential access when a series of records is to be accessed.

Despite increasing disk performance, network speeds have improved to the point that disk access is often a significant bottleneck in an overall I/O system. A number of techniques are available to address this problem, including striping, the use of RAM disks, and disk caching.

Research done in connection with BSD Unix shows that block size can have a major effect on performance. By increasing the default block size from 512 bytes to 4096 bytes, throughput was improved enormously, especially for large files, because eight times as much data could be transferred in a single access. A negative consequence of this reorganization was that wasted storage increased from 6.9 percent for 512-byte blocks to 45.6 percent for 4096-byte blocks. It turned out that this problem of wasted space could be dealt with by treating the 4096-byte blocks as clusters of 512-byte blocks, which could be allocated to different files.

Though not as important as disks, magnetic tape has an important niche in file processing. Tapes are inexpensive, reasonably fast for sequential processing, compact, robust, and easy to store and transport. Data is usually organized on tapes in 1-bit-wide parallel tracks, with a bit-wide cross section of tracks interpreted as 1 or more bytes. When estimating processing speed and space utilization, it is important to recognize the role played by the interblock gap. Effective recording density and effective transmission rate are useful measurements of the performance one can expect to achieve for a given physical file organization.

In comparing disk and tape as secondary storage media, we see that disks are replacing tape in more and more cases. This is largely because memory is becoming less expensive relative to secondary storage, which means that one of the earlier advantages of tape over disk, the ability to do sequential access without seeking, has diminished significantly.

CD-ROM is an electronic publishing medium that allows us to replicate and distribute large amounts of information very inexpensively. The primary disadvantage of CD-ROM is that seek performance is relatively slow. This is not a problem that can be solved simply by building better drives; the limits in seek performance grow directly from the fact that CD-ROM is built on top of the CD audio standard. Adherence to this standard, even given its limitations, is the basis for CD-ROM's success as a publishing medium. Consequently, CD-ROM application developers must look to careful file structure design to build fast, responsive retrieval software.

This chapter follows a journey of a byte as it is sent from memory to disk. The journey involves the participation of many different programs and devices, including

- a user's program, which makes the initial call to the operating system;
- the operating system's file manager, which maintains tables of information that it uses to translate between the program's logical view of the file and the physical file where the byte is to be stored;
- an I/O processor and its software, which transmit the byte, synchronizing the transmission of the byte between an I/O buffer in memory and the disk;
- the disk controller and its software, which instruct the drive about how to find the proper track and sector and then send the byte; and
- the disk drive, which accepts the byte and deposits it on the disk surface.

Next, we take a closer look at buffering, focusing mainly on techniques for managing buffers to improve performance. Some techniques include double buffering, buffer pooling, locate-mode buffering, and scatter/gather buffering.

We conclude with a second look at I/O layers, this time concentrating on Unix. We see that every I/O system call begins with a call to the Unix kernel, which knows nothing about the logical structure of a file, treating all data essentially the same—as a sequence of bytes to be transmitted to some external device. In doing its work the I/O system in the kernel invokes four tables: a file descriptor table, an open file table, an inode table, and a file access table in the file's inode. Once the kernel has determined which device to use and how to access it, it calls on a device driver to carry out the accessing.

Although it treats every file as a sequence of bytes, the kernel I/O system deals differently with three different types of I/O: block I/O, character I/O, and network I/O. In this text we concentrate on block I/O. We look briefly at the special role of the file system within the kernel, describing how it uses links to connect file names in directories to their corresponding inodes. Finally, we remark on the reasons that magnetic tape does not fit well into the Unix paradigm for I/O.

KEY TERMS

Block. Unit of data organization corresponding to the amount of data transferred in a single access. *Block* often refers to a collection of records, but it may be a collection of sectors (see *cluster*) whose size has no correspondence to the organization of the data. A block is sometimes called a physical record; a sector is sometimes called a block.

Block device. In Unix, a device such as a disk drive that is organized in blocks and accessed accordingly.

Block I/O. I/O between a computer and a block device.

Block organization. Disk drive organization that allows the user to define the size and organization of blocks and then access a block by giving its block address or the key of one of its records. (See *sector organization.*)

Blocking factor. The number of records stored in one block.

bpi. Bits per inch per track. On a disk, data is recorded serially on tracks. On a tape, data is recorded in parallel on several tracks, so a 6250-bpi nine-track tape contains 6250 bytes per inch, when all nine tracks are taken into account (one track used for parity).

Cartridge tape. Tape systems in which the media are stored in a container, rather than on independent tape reels.

Character device. In Unix, a device such as a keyboard or printer (or tape drive when stream I/O is used) that sends or receives data in the form of a stream of characters.

Character I/O. I/O between a computer and a character device.

Cluster. Minimum unit of space allocation on a sectored disk, consisting of one or more contiguous sectors. The use of large clusters can improve sequential access times by guaranteeing the ability to read longer spans of data without seeking. Small clusters tend to decrease internal fragmentation.

Controller. Device that directly controls the operation of one or more secondary storage devices, such as disk drives and magnetic tape units.

Count subblock. On block-organized drives, a small block that precedes each data block and contains information about the data block, such as its byte count and its address.

Cylinder. The set of tracks on a disk that are directly above and below each other. All of the tracks in a given cylinder can be accessed without having to move the access arm—they can be accessed without the expense of seek time.

Descriptor table. In Unix, a table associated with a single process that links all of the file descriptors generated by that process to corresponding entries in an open file table.

Device driver. In Unix, an I/O processor program invoked by the kernel that performs I/O for a particular device.

Direct access storage device (DASD). Disk or other secondary storage device that permits access to a specific sector or block of data without first requiring the reading of the blocks that precede it.

Direct memory access (DMA). Transfer of data directly between memory and peripheral devices, without significant involvement by the CPU.

Disk cache. A segment of memory configured to contain pages of data from a disk. Disk caches can lead to substantial improvements in access time when access requests exhibit a high degree of locality.

Disk drive. An assemblage of magnetic disks mounted on the same vertical shaft. A disk drive is treated as a single unit consisting of a number of cylinders equivalent to the number of tracks per surface.

Disk striping. Storing information on multiple disk drives by splitting up the information and accessing all of the drives in parallel.

Effective recording density. Recording density after taking into account the space used by interblock gaps, nondata subblocks, and other space-consuming items that accompany data.

Effective transmission rate. Transmission rate after taking into account the time used to locate and transmit the block of data in which a desired record occurs.

Extent. One or more adjacent clusters allocated as part (or all) of a file. The number of extents in a file reflects how dispersed the file is over the disk. The more dispersed a file, the more seeking must be done in moving from one part of the file to another.

File allocation table (FAT). A table that contains mappings to the physical locations of all the clusters in all files on disk storage.

File manager. The part of an operating system that is responsible for managing files, including a collection of programs whose responsibilities range from keeping track of files to invoking I/O processes that transmit information between primary and secondary storage.

File structure. In connection with the open file table in a Unix kernel, the term *file structure* refers to a structure that holds information the kernel needs about an open file. File structure information includes such things as the file's read/write mode, the number of processes currently using it, and the offset within the file to be used for the next read or write.

File system. In Unix, a hierarchical collection of files, usually kept on a single secondary device, such as a hard disk or CD-ROM.

Fixed disk. A disk drive with platters that may not be removed.

Formatting. The process of preparing a disk for data storage, involving such things as laying out sectors, setting up the disk's file allocation table, and checking for damage to the recording medium.

Fragmentation. Space that goes unused within a cluster, block, track, or other unit of physical storage. For instance, track fragmentation occurs when space on a track goes unused because there is not enough space left to accommodate a complete block.

Frame. A 1-bit-wide slice of tape, usually representing a single byte.

Hard link. In Unix, an entry in a directory that connects a file name to the inode of the corresponding file. There can be several hard links to a single file; hence a file can have several names. A file is not deleted until all hard links to the file are deleted.

Index node. In Unix, a data structure associated with a file that describes the file. An index node includes such information as a file's type, its owner and group IDs, and a list of the disk blocks that comprise the file. A more common name for index node is *inode*.

Inode. *See* index node.

Interblock gap. An interval of blank space that separates sectors, blocks, or subblocks on tape or disk. In the case of tape, the gap provides sufficient space for the tape to accelerate or decelerate when starting or stopping. On both tapes and disks the gaps enable the read/write heads to tell accurately when one sector (or block or subblock) ends and another begins.

Interleaving factor. Since it is often not possible to read physically adjacent sectors of a disk, logically adjacent sectors are sometimes arranged so they are not physically adjacent. This is called interleaving. The interleaving factor refers to the number of physical sectors the next logically adjacent sector is located from the current sector being read or written.

I/O processor. A device that carries out I/O tasks, allowing the CPU to work on non-I/O tasks.

Kernel. The central part of the Unix operating system.

Key subblock. On block-addressable drives, a block that contains the key of the last record in the data block that follows it, allowing the drive to search among the blocks on a track for a block containing a certain key, without having to load the blocks into primary memory.

Mass storage system. General term applied to storage units with large capacity. Also applied to very high-capacity secondary storage systems that are capable of transmitting data between a disk and any of several thousand tape cartridges within a few seconds.

Nominal recording density. Recording density on a disk track or magnetic tape without taking into account the effects of gaps or nondata subblocks.

Nominal transmission rate. Transmission rate of a disk or tape unit without taking into account the effects of such extra operations as seek time for disks and interblock gap traversal time for tapes.

Open file table. In Unix, a table owned by the kernel with an entry, called a file structure, for each open file. See *file structure.*

Parity. An error-checking technique in which an extra parity bit accompanies each byte and is set in such a way that the total number of 1 bits is even (even parity) or odd (odd parity).

Platter. One disk in the stack of disks on a disk drive.

Process. An executing program. In Unix, several instances of the same program can be executing at the same time, as separate processes. The kernel keeps a separate file descriptor table for each process.

RAID disk system. An array of disk drives that provide access to the disks in parallel. Storage of files on RAID systems often involves disk striping.

RAM disk. Block of memory configured to simulate a disk.

Rotational delay. The time it takes for the disk to rotate so the desired sector is under the read/write head.

Scatter/gather I/O. Buffering techniques that involve, on input, scattering incoming data into more than one buffer and, on output, gathering data from several buffers to be output as a single chunk of data.

Sector. The fixed-sized data blocks that together make up the tracks on certain disk drives. Sectors are the smallest addressable unit on a disk whose tracks are made up of sectors.

Sector organization. Disk drive organization that uses sectors.

Seek time. The time required to move the access arm to the correct cylinder on a disk drive.

Sequential access device. A device, such as a magnetic tape unit or card reader, in which the medium (for example, tape) must be accessed from the beginning. Sometimes called a serial device.

Socket. In Unix, a socket is an abstraction that serves as an endpoint of communication within some domain. For example, a socket can be used to provide direct communication between two computers. Although in some ways the kernel treats sockets like files, we do not deal with sockets in this text.

Soft link. See *symbolic link.*

Special file. In Unix, the term *special file* refers to a stream of characters and control signals that drive some device, such as a line printer or a graphics device.

Streaming tape drive. A tape drive whose primary purpose is to dump large amounts of data from disk to tape or from tape to disk.

Subblock. When blocking is used, there are often separate groupings of information concerned with each individual block. For example, a count subblock, a key subblock, and a data subblock might all be present.

Symbolic link. In Unix, an entry in a directory that gives the pathname of a file. Since a symbolic link is an indirect pointer to a file, it is not as closely associated with the file as a hard link. Symbolic links can point to directories or even to files in other file systems.

Track. The set of bytes on a single surface of a disk that can be accessed without seeking (without moving the access arm). The surface of a disk can be thought of as a series of concentric circles with each circle corresponding to a particular position of the access arm and read/write heads. Each of these circles is a track.

Transfer time. Once the data we want is under the read/write head, we have to wait for it to pass under the head as we read it. The amount of time required for this motion and reading is the transfer time.

FURTHER READINGS

Many textbooks contain more detailed information on the material covered in this chapter. In the area of operating systems and file management systems, we have found the operating system texts by Deitel (1989), Silberschatz and Galvin (1998), and Tannenbaum, et al. (1997) useful. Hanson (1982) has a great deal of material on blocking and buffering, secondary storage devices, and performance.

Ritchie and Thompson (1974), Kernighan and Ritchie (1978), and McKusick et al. (1984) provide information on how file I/O is handled in the Unix operating system. The latter provides a good case study of ways in which a file system can be altered to provide substantially faster throughput for certain applications. A comprehensive coverage of Unix I/O from the design perspective can be found in Leffler et al. (1989). Information about I/O devices and file system services for Windows 95 and Windows NT is covered in Hart (1997).

Information on specific systems and devices can often be found in manuals and documentation published by manufacturers and in web sites. Information on specific disks, tapes, and CDs that is presented in this chapter comes from web sites for Seagate, Western Digital, StorageTek, and Sony, among others.

EXERCISES

1. Determine as well as you can what the journey of a byte would be like on your system. You may have to consult technical reference manuals that describe your computer's file management system, operating system, and peripheral devices. You may also want to talk to local gurus who have experience using your system.

2. Suppose you are writing a list of names to a text file, one name per write statement. Why is it not a good idea to close the file after every write and then reopen it before the next write?
3. Find out what utility routines for monitoring I/O performance and disk utilization are available on your computer system. If you have a large computing system, there are different routines available for different kinds of users, depending on what privileges and responsibilities they have.
4. When you create or open a file in C++, you must provide certain information to your computer's file manager so it can handle your file properly. Compared to certain languages, such as Cobol, the amount of information you must provide in C++ is very small. Find a text or manual on PL/I or Cobol and look up the ENVIRONMENT file description attribute, which can be used to tell the file manager a great deal about how you expect a file to be organized and used. Compare PL/I or Cobol with C++ in terms of the types of file specifications available to the programmer.
5. Much is said in section 3.1 about how disk space is organized physically to store files. Assume that no such complex organization is used and that every file must occupy a single contiguous piece of a disk, somewhat the way a file is stored on tape. How does this simplify disk storage? What problems does it create?
6. A disk drive uses 512-byte sectors. If a program requests that a 128-byte record be written to disk, the file manager may have to read a sector from the disk before it can write the record. Why? What could you do to decrease the number of times such an extra read is likely to occur?
7. Use the Internet to determine the detailed characteristics of current disk drives. Reproduce the information in Table 3.1 for three new disk drives.
8. In early Unix systems, inodes were kept together on one part of a disk, while the corresponding data was scattered elsewhere on the disk. Later editions divided disk drives into groups of adjacent cylinders called cylinder groups, in which each cylinder group contains inodes and their corresponding data. How does this new organization improve performance?
9. In early Unix systems, the minimum block size was 512 bytes, with a cluster size of one. The block size was increased to 1024 bytes in 4.0BSD, more than doubling its throughput. Explain how this could occur.

10. Draw pictures that illustrate the role of fragmentation in determining the numbers in Table 3.2, section 3.1.7.

11. The IBM 3350 disk drive uses block addressing. The two subblock organizations described in the text are available:

    Count-data, where the extra space used by count subblock and interblock gaps is equivalent to 185 bytes; and

    Count-key-data, where the extra space used by the count and key subblocks and accompanying gaps is equivalent to 267 bytes, plus the key size.

    An IBM 3350 has 19 069 usable bytes available per track, 30 tracks per cylinder, and 555 cylinders per drive. Suppose you have a file with 350 000 80-byte records that you want to store on a 3350 drive. Answer the following questions. Unless otherwise directed, assume that the blocking factor is 10 and that the count-data subblock organization is used.

    a. How many blocks can be stored on one track? How many records?
    b. How many blocks can be stored on one track if the count-key-data subblock organization is used and key size is 13 bytes?
    c. Make a graph that shows the effect of block size on storage utilization, assuming count-data subblocks. Use the graph to help predict the best and worst possible blocking factor in terms of storage utilization.
    d. Assuming that access to the file is always sequential, use the graph from 11c to predict the best and worst blocking factor. Justify your answer in terms of efficiency of storage utilization and processing time.
    e. How many cylinders are required to hold the file (blocking factor 10 and count-data format)? How much space will go unused due to internal track fragmentation?
    f. If the file were stored on contiguous cylinders and if there were no interference from other processes using the disk drive, the average seek time for a random access of the file would be about 12 msec. Use this rate to compute the average time needed to access one record randomly.
    g. Explain how retrieval time for random accesses of records is affected by increasing block size. Discuss trade-offs between storage efficiency and retrieval when different block sizes are used. Make a table with different block sizes to illustrate your explanations.
    h. Suppose the file is to be sorted, and a shell sort is to be used to sort the file. Since the file is too large to read into memory, it will be

sorted in place, on the disk. It is estimated (Knuth, 1973b, p. 380) that this requires about 15 $N^{1.25}$ moves of records, where N represents the total number of records in the file. Each move requires a random access. If all of the preceding is true, how long does it take to sort the file? (As you will see, this is not a very good solution. We provide much better ones in Chapter 7, which deals with cosequential processing.)

12. A sectored disk drive differs from one with a block organization in that there is less of a correspondence between the logical and physical organization of data records or blocks.

    For example, consider the Seagate Cheetah 9 disk drive, described in Table 3.1. From the drive's (and drive controller's) point of view, a file is just a vector of bytes divided into 512-byte sectors. Since the drive knows nothing about where one record ends and another begins, a record can span two or more sectors, tracks, or cylinders.

    One common way that records are formatted is to place a two-byte field at the beginning of each block, giving the number of bytes of data, followed by the data itself. There is no extra gap and no other overhead. Assuming that this organization is used, and that you want to store a file with 350 000 80-byte records, answer the following questions:

    a. How many records can be stored on one track if one record is stored per block?
    b. How many cylinders are required to hold the file?
    c. How might you block records so each physical record access results in 10 actual records being accessed? What are the benefits of doing this?

13. Suppose you have a collection of 500 large images stored in files, one image per file, and you wish to "animate" these images by displaying them in sequence on a workstation at a rate of at least 15 images per second over a high-speed network. Your secondary storage consists of a disk farm with 30 disk drives, and your disk manager permits striping over as many as 30 drives, if you request it. Your drives are guaranteed to perform I/O at a steady rate of 2 megabytes per second. Each image is 3 megabytes in size. Network transmission speeds are not a problem.

    a. Describe in broad terms the steps involved in doing such an animation in real time from disk.
    b. Describe the performance issues that you have to consider in implementing the animation. Use numbers.

c. How might you configure your I/O system to achieve the desired performance?

14. Consider the 1 000 000-record mailing-list file discussed in the text. The file is to be backed up on 2400-foot reels of 6250-bpi tape with 0.3-inch interblock gaps. Tape speed is 200 inches per second.

    a. Show that only one tape would be required to back up the file if a blocking factor of 50 is used.
    b. If a blocking factor of 50 is used, how many extra records could be accommodated on a 2400-foot tape?
    c. What is the effective recording density when a blocking factor of 50 is used?
    d. How large does the blocking factor have to be to achieve the maximum effective recording density? What negative results can result from increasing the blocking factor? (*Note:* An I/O buffer large enough to hold a block must be allocated.)
    e. What would be the minimum blocking factor required to fit the file onto the tape?
    f. If a blocking factor of 50 is used, how long would it take to read one block, including the gap? What would the effective transmission rate be? How long would it take to read the entire file?
    g. How long would it take to perform a binary search for one record in the file, assuming that it is not possible to read backwards on the tape? (Assume that it takes 60 seconds to rewind the tape.) Compare this with the expected average time it would take for a sequential search for one record.
    h. We implicitly assume in our discussions of tape performance that the tape drive is always reading or writing at full speed, so no time is lost by starting and stopping. This is not necessarily the case. For example, some drives automatically stop after writing each block.

       Suppose that the extra time it takes to start before reading a block and to stop after reading the block totals 1 msec and that the drive must start before and stop after reading each block. How much will the effective transmission rate be decreased due to starting and stopping if the blocking factor is 1? What if it is 50?

15. Why are there interblock gaps on linear tapes? In other words, why do we not just jam all records into one block?

16. The use of large blocks can lead to severe internal fragmentation of tracks on disks. Does this occur when tapes are used? Explain.

17. Each MS-DOS file system (or drive) uses a FAT with 64K entries. For each disk in Table 3.1, give the minimum sector size if the disk is configured as a single MS-DOS drive. Each file uses a minimum of one sector.

18. Use the Internet to determine the characteristics of the second generation of Digital Versatile Disc (DVD). What are the plans to put four independent surfaces on a single disc? What are the density, sector size, and transfer rate for these new disc systems?

CHAPTER 4

Fundamental File Structure Concepts

CHAPTER OBJECTIVES

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

CHAPTER OUTLINE

4.1 Field and Record Organization
4.1.1 A Stream File
4.1.2 Field Structures
4.1.3 Reading a Stream of Fields
4.1.4 Record Structures
4.1.5 A Record Structure That Uses a Length Indicator
4.1.6 Mixing Numbers and Characters: Use of a File Dump
4.2 Using Classes to Manipulate Buffers
4.2.1 Buffer Class for Delimited Text Fields
4.2.2 Extending Class Person with Buffer Operations
4.2.3 Buffer Classes for Length-Based and Fixed-Length Fields
4.3 Using Inheritance for Record Buffer Classes
4.3.1 Inheritance in the C++ Stream Classes
4.3.2 A Class Hierarchy for Record Buffer Objects
4.4 Managing Fixed-Length, Fixed-Field Buffers
4.5 An Object-Oriented Class for Record Files

4.1 Field and Record Organization

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

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

4.1.1 A Stream File

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

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

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

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

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

```
AmesMary123 MapleStillwaterOK74075MasonAlan90 EastgateAdaOK74820
```

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

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

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

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

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

4.1.2 Field Structures

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

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

Method 1: Fix the Length of Fields

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

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

In C:

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

In C++:

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

Figure 4.2 Definition of record to hold person information.

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

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

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

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

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

(a)

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

(b)

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

(c)

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

(d)

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

Method 2: Begin Each Field with a Length Indicator

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

Method 3: Separate the Fields with Delimiters

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

The choice of a delimiter character can be very important as it must be a character that does not get in the way of processing. In many instances *white-space characters* (blank, new line, tab) make excellent delimiters because they provide a clean separation between fields when we list them on the console. Also, most programming languages include I/O statements which, by default, assume that fields are separated by white space.

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

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

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

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

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

4.1.3 Reading a Stream of Fields

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

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

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

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

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

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

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

4.1.4 Record Structures

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

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

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

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

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

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

A *fixed-length record file* is one in which each record contains the same number of bytes. This method of recognizing records is analogous to the first method we discussed for making fields recognizable. As we will see in the chapters that follow, fixed-length record structures are among the most commonly used methods for organizing files.

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

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

Method 2: Make Records a Predictable Number of Fields

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

Method 3: Begin Each Record with a Length Indicator

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

Method 4: Use an Index to Keep Track of Addresses

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

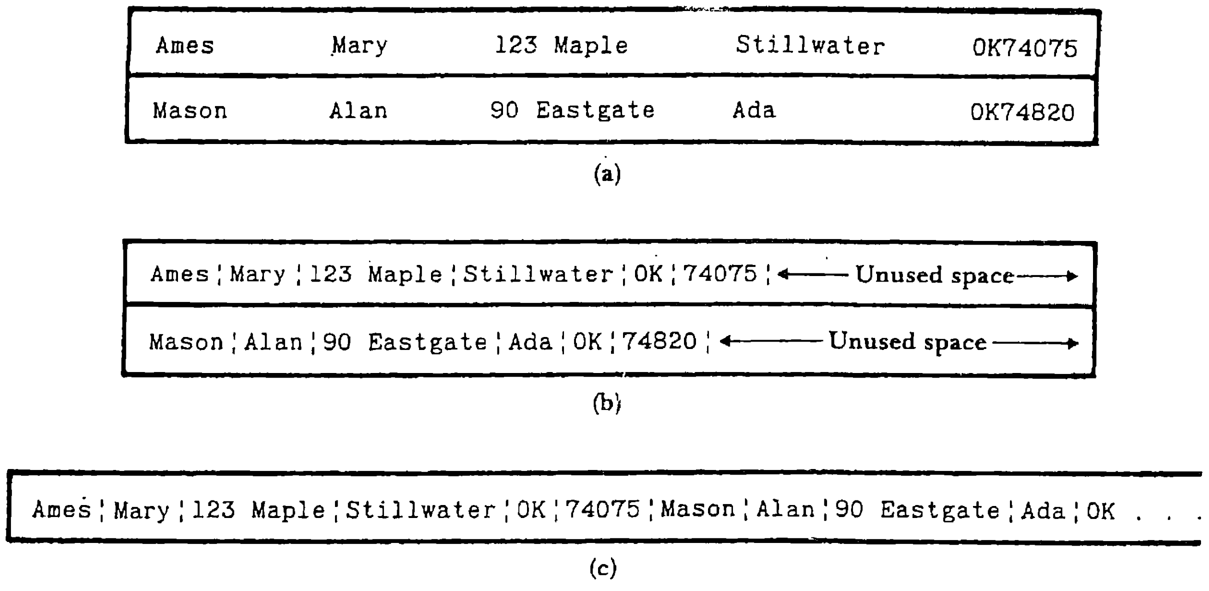

Ames Mary 123 Maple Stillwater OK74075
Mason Alan 90 Eastgate Ada OK74820
(a)
Ames¦Mary¦123 Maple¦Stillwater¦OK¦74075¦ Unused space
Mason¦Alan¦90 Eastgate¦Ada¦OK¦74820¦ Unused space
(b)
Ames¦Mary¦123 Maple¦Stillwater¦OK¦74075¦Mason¦Alan¦90 Eastgate¦Ada¦OK . . .
(c)

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

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

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

4.1.5 A Record Structure That Uses a Length Indicator

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

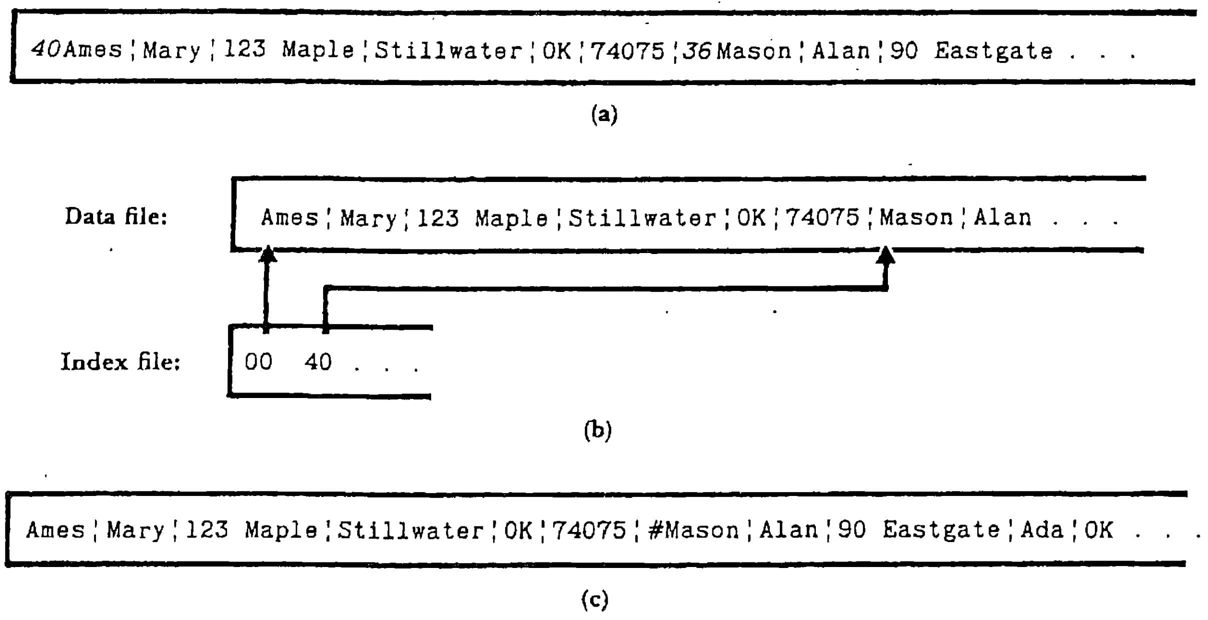

40Ames¦Mary¦123 Maple¦Stillwater¦OK¦74075¦36Mason¦Alan¦90 Eastgate . . .
(a)
Data file:
Ames¦Mary¦123 Maple¦Stillwater¦OK¦74075¦Mason¦Alan . . .
Index file:
00 40 . . .
(b)
Ames¦Mary¦123 Maple¦Stillwater¦OK¦74075¦#Mason¦Alan¦90 Eastgate¦Ada¦OK . . .
(c)

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

Writing the Variable-Length Records to the File

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

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

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

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

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

Representing the Record Length

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

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

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

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

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

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

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

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

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

Reading the Variable-Length Records from the File

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

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

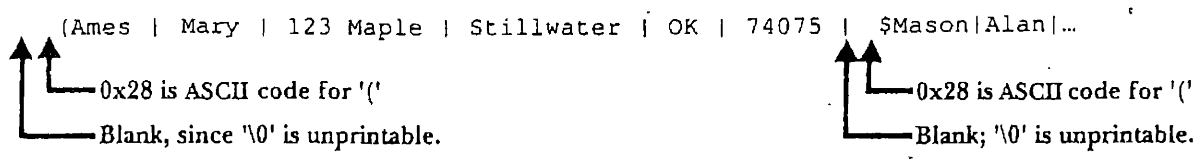
Figure 4.9 Function `ReadVariablePerson` that reads a variable-sized `Person` record.

4.1.6 Mixing Numbers and Characters: Use of a File Dump

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

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

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



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

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

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

```
od -xc filename
```

produces the following:

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

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

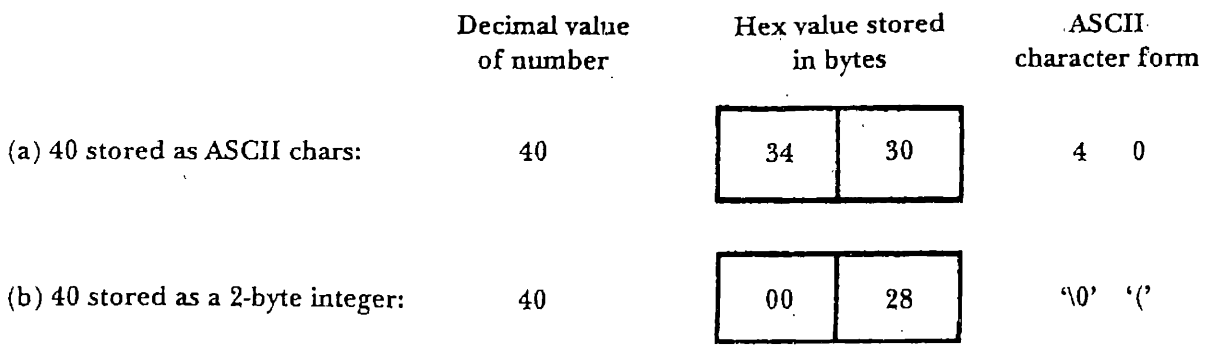

Decimal value of number
Hex value stored in bytes
ASCII character form
(a) 40 stored as ASCII chars:
40
34
30
4 0
(b) 40 stored as a 2-byte integer:
40
00
28
'\0' '('

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

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

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

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

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

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

A Note about Byte Order

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

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

4.2 Using Classes to Manipulate Buffers

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

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

4.2.1 Buffer Class for Delimited Text Fields

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

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

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

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

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

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

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

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

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

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

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

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

```
int DelimTextBuffer :: Read (istream & stream)
{
   Clear ();
   stream . read ((char *)&BufferSize, sizeof(BufferSize));
   if (stream.fail()) return FALSE;
   if (BufferSize > MaxBytes) return FALSE; // buffer overflow
   stream . read (Buffer, BufferSize);
   return stream . good ();
}
```

4.2.2 Extending Class Person with Buffer Operations

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

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

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

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

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

Figure 4.12 Main methods and members of class LengthTextBuffer.

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

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

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

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

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

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

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

4.3 Using Inheritance for Record Buffer Classes

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

4.3.1 Inheritance in the C++ Stream Classes

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

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

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

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

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

4.3.2 A Class Hierarchy for Record Buffer Objects

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

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

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

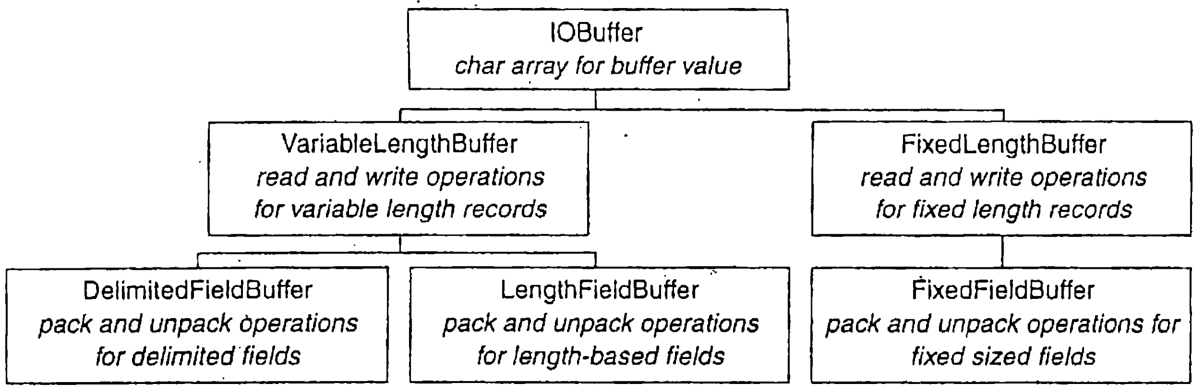

IOBuffer
char array for buffer value
VariableLengthBuffer
read and write operations
for variable length records
FixedLengthBuffer
read and write operations
for fixed length records
DelimitedFieldBuffer
pack and unpack operations
for delimited fields
LengthFieldBuffer
pack and unpack operations
for length-based fields
FixedFieldBuffer
pack and unpack operations for
fixed sized fields

Figure 4.14 Buffer class hierarchy

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

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

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

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

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

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

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

```
int VariableLengthBuffer :: Write (ostream & stream) const
// read the length and buffer from the stream
{
   int recaddr = stream . tellp ();
   unsigned short bufferSize = BufferSize;
   stream . write ((char *)&bufferSize, sizeof(bufferSize));
   if (!stream) return -1;
   stream . write (Buffer, BufferSize);
   if (!stream.good ()) return -1;
   return recaddr;
}
```

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

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

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

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

Figure 4.16 Classes `VariableLengthBuffer` and `DelimFieldBuffer`.

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

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

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

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

4.4 Managing Fixed-Length, Fixed-Field Buffers

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

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

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

Figure 4.17 Class `FixedFieldBuffer`.

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

```
int Person::InitBuffer (FixedFieldBuffer & Buffer)
// initialize a FixedFieldBuffer to be used for Persons
{
   int result;
   result = Buffer . AddField (10); // LastName [11];
   result = result && Buffer . AddField (10); // FirstName [11];
   result = result && Buffer . AddField (15); // Address [16];
   result = result && Buffer . AddField (15); // City [16];
   result = result && Buffer . AddField (2);  // State [3];
   result = result && Buffer . AddField (9); // ZipCode [10];
   return result;
}
```

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

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

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

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

4.5 An Object-Oriented Class for Record Files

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

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

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

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

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

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

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

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

SUMMARY

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

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

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

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

In this chapter we use the record structure with a length indicator at the beginning of each record to develop programs for writing and reading a simple file of variable-length records containing names and addresses of individuals. We use buffering to accumulate the data in an individual record before we know its length to write it to the file. Buffers are also useful in allowing us to read in a complete record at one time. We represent the length field of each record as a binary number or as a sequence of ASCII digits. In the former case, it is useful to use a *file dump* to examine the contents of our file.

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

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

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

KEY TERMS

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

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

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

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

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

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

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

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

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

FURTHER READINGS

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

EXERCISES

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

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

```
LNAmes     FNMary     AD123 Maple     CTStillwaterSTOKZP74075
```

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

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

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

PROGRAMMING EXERCISES

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

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

PROGRAMMING PROJECT

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

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

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

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

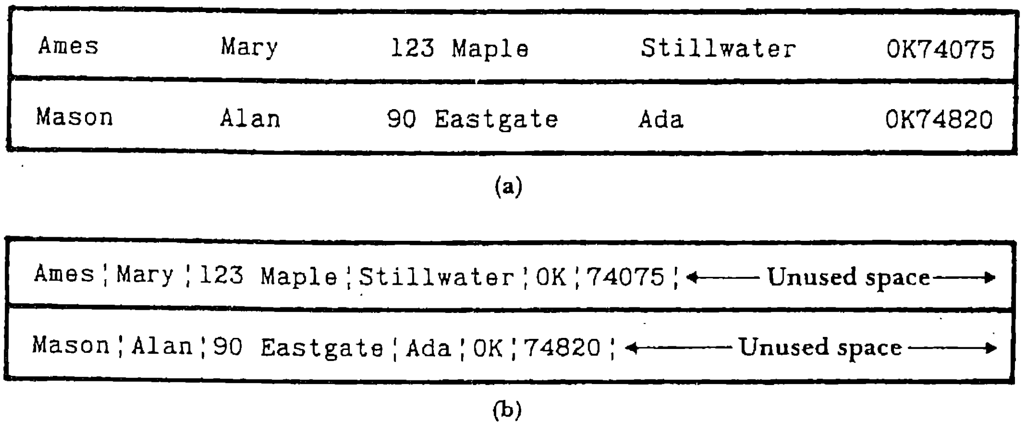

Ames Mary 123 Maple Stillwater OK74075
Mason Alan 90 Eastgate Ada OK74820
(a)
Ames | Mary | 123 Maple | Stillwater | OK | 74075 |
Unused space
Mason | Alan | 90 Eastgate | Ada | OK | 74820 |
Unused space
(b)

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:

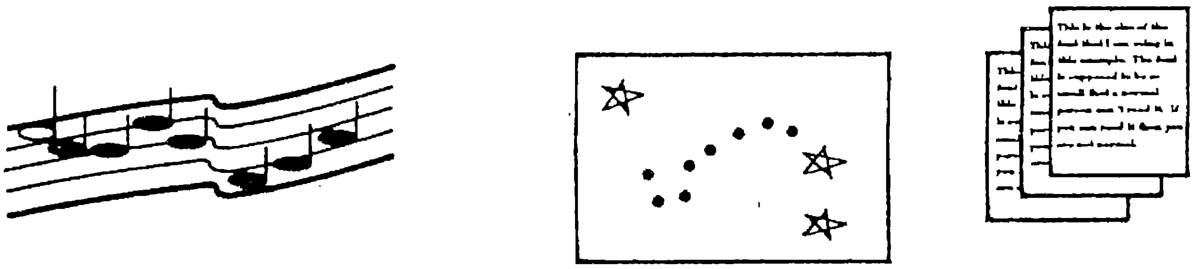


Figure 5.4 Data such as sound, images, and documents do not fit the traditional metaphor of data stored as sequences of records that are divided into fields.

- 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 hanale 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

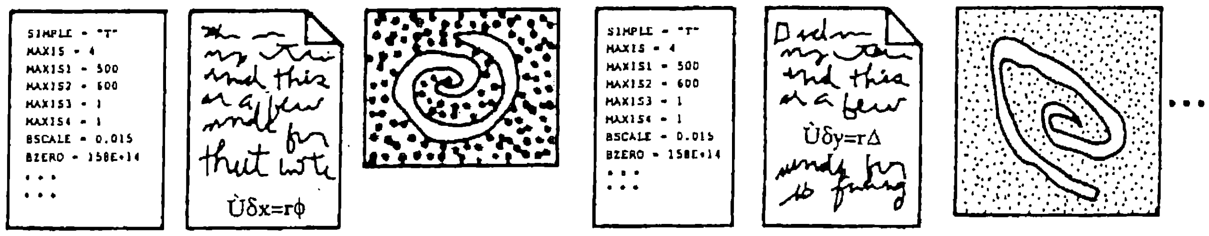


Figure 5.8 Information that an astronomer wants to include in a file.

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

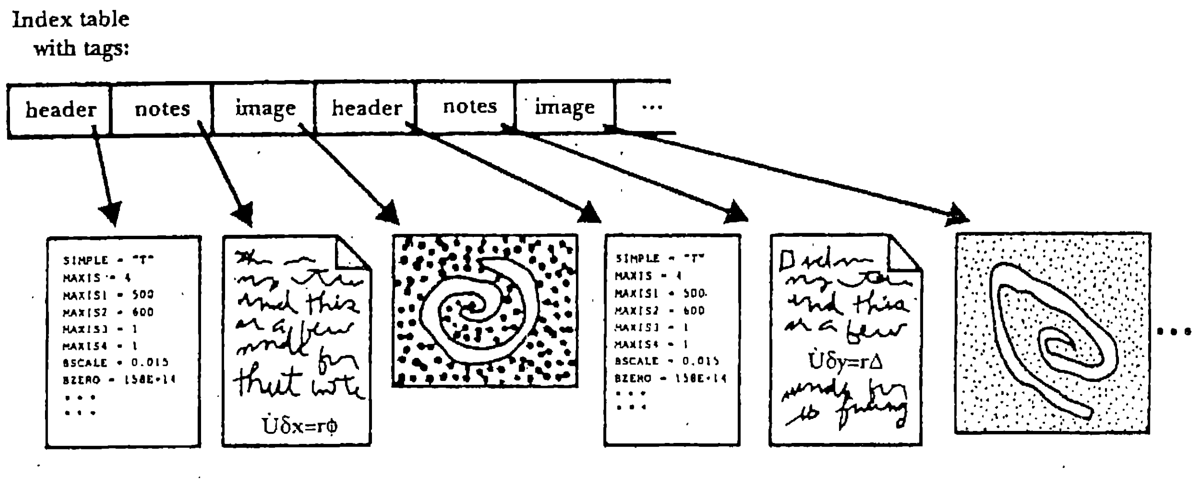


Figure 5.9 Same as Fig. 5.8, except with tags identifying the objects.

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.

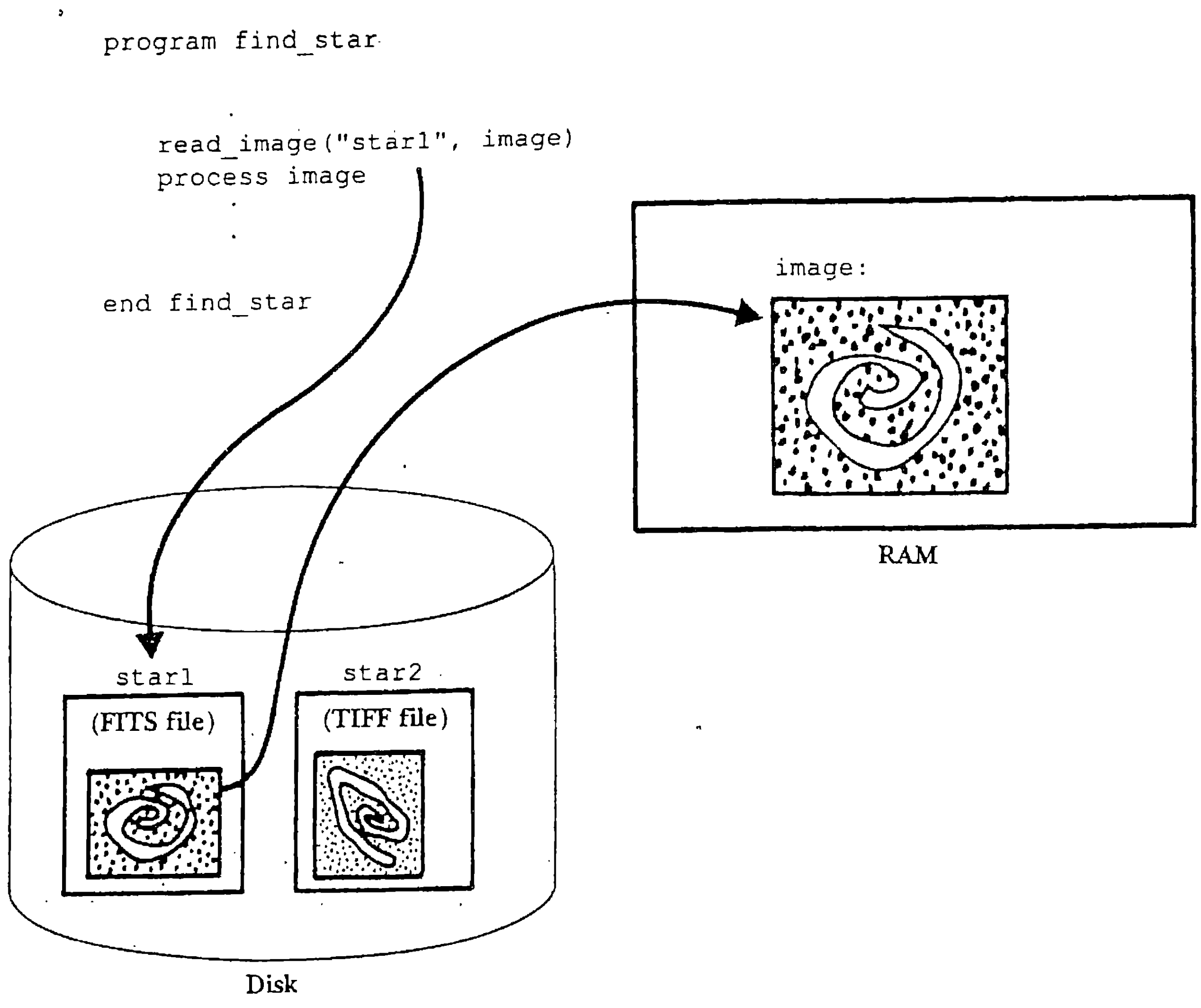


Figure 5.10 Example of object-oriented access. The program `find_star` knows nothing about the file format of the image that it wants to read. The routine `read_image` has methods to convert the image from whatever format it is stored in on disk into the 8-bit in-memory format required by `find_star`.

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 implement 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).

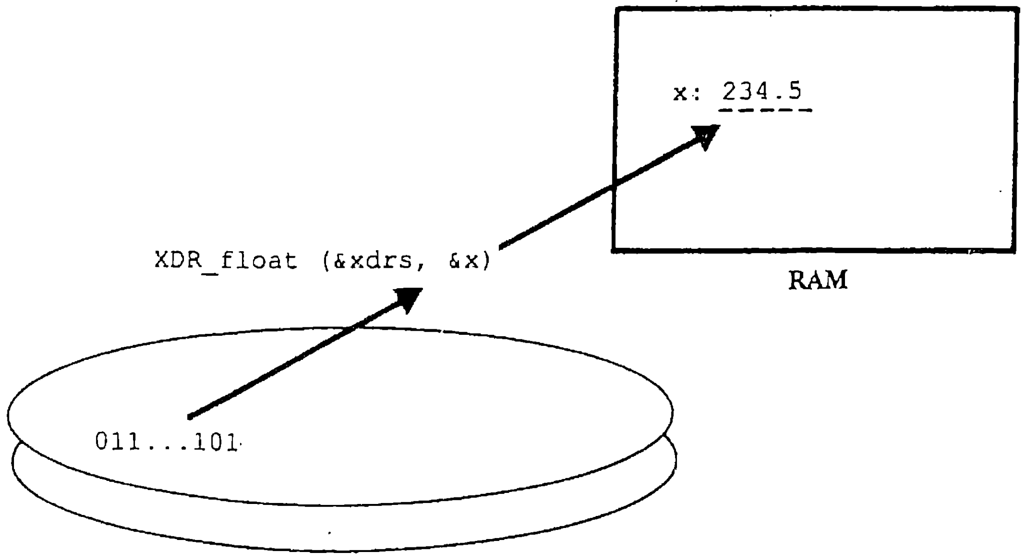


Figure 5.11 XDR specifies a standard external data representation for numbers stored in a file. XDR routines are provided for converting to and from the XDR representation to the encoding scheme used on the host machine. Here a routine called `XDR_float` translates a 32-bit floating point number from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

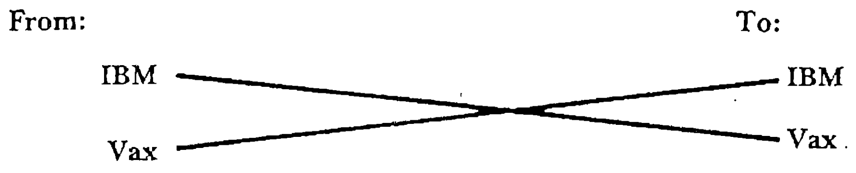
What happens, then, when you want to move your files back and forth between your IBM and a VAX, which uses a different native format for numbers and generally uses ASCII for text? You need a way to convert from the IBM format to the VAX format and back. One solution is to write (or borrow) a program 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.

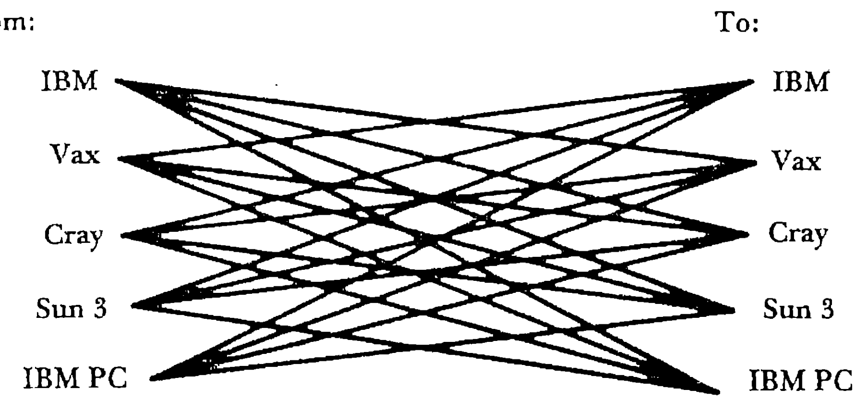
In this case, a better solution would probably be to agree on a standard intermediate format, such as XDR, and translate files into XDR whenever they are to be exported to a different platform. This solution is illustrated in Fig. 5.12(c). Not only does it reduce the number of translators from $n(n-1)$ 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

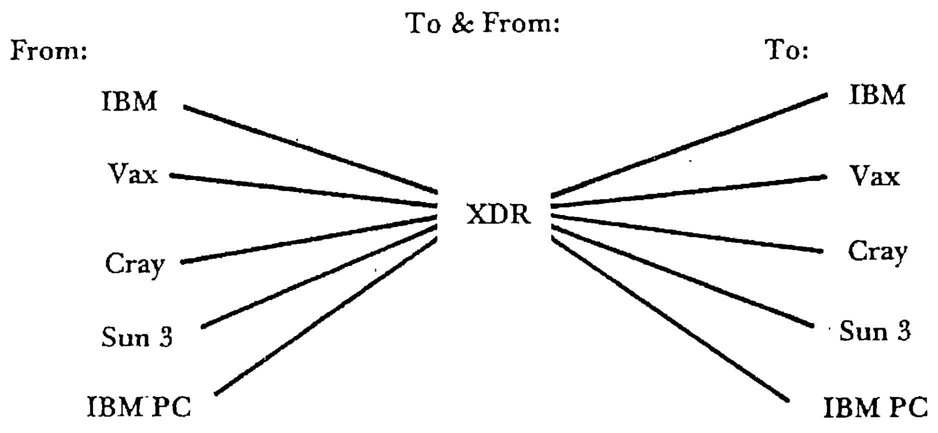
Suppose you are a doctor and you have X-ray raster images of a particular organ taken periodically over several minutes. You want to look at a certain image in the collection using a program that lets you zoom in and out and detect special features in the image. You have another program that lets you 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.