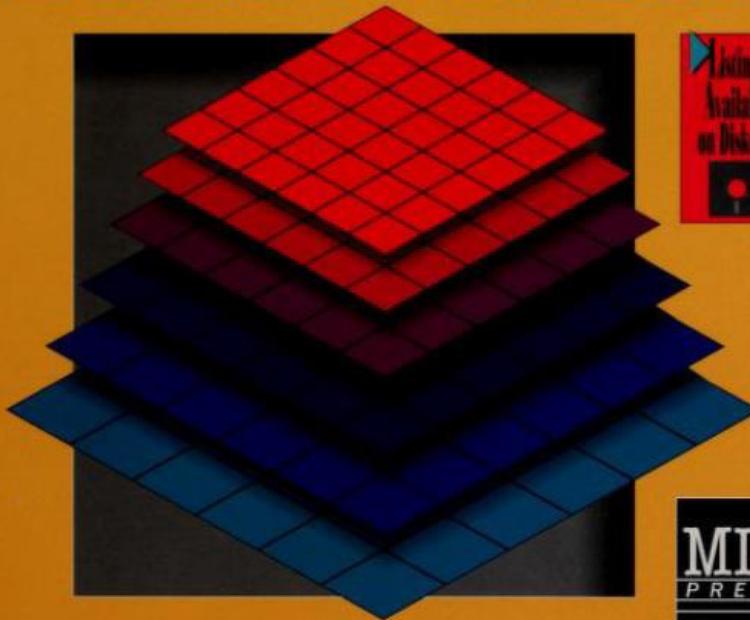


**2ND
EDITION
UPGRADED TO ANSI C**

C DATABASE DEVELOPMENT



**MIS:
PRESS**

AL STEVENS



**2ND
EDITION**

C **DATABASE DEVELOPMENT**

AL STEVENS

**ADVANCED
COMPUTER
BOOKS**



© 1991 by Management Information Source, Inc.
P.O. Box 5277
Portland, Oregon 97208-5277

All rights reserved. Reproduction or use of editorial or pictorial content in any manner is prohibited without express permission. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

Second Printing
ISBN 1-55828-135-5

Printed in the United States of America

TRADEMARKS

CP/M is a trademark of Digital Research
IBM PC is a trademark of IBM
Microsoft C is a trademark of Microsoft Corporation
TopSpeed C is a trademark of Jensen & Partners International, Inc.
Turbo C and Turbo C++ are trademarks of Borland International, Inc.
TurboDos is a trademark of Software 2000, Inc.
UNIX is a trademark of AT&T Bell Laboratories
WATCOM C is a trademark of WATCOM Products, Inc.

Dedication

To the memory of Chester Fortune

Acknowledgments

Thanks to Borland, Microsoft, WATCOM, and JPI & Associates for providing their fine compiler products.

And a salute to the members of the ANSI X3J11 committee.

Contents

PREFACE	xiii
CHAPTER 1. INTRODUCTION	1
Database Management	4
Chapter Overviews	4
Summary	6
 CHAPTER 2. DATABASE FUNDAMENTALS	7
The Database	8
The Data Element and Data Value	10
The Data Element Dictionary	11
Data Element Types	12
Data Element Names	13
Data Value Representation	13
Files	13
The Database Schema	14
The Specification of Data Files	15
The Data Definition Language	15
Key Data Elements	15
Interfile Relationships	17
Summary	23
 CHAPTER 3. DESIGNING A DATABASE	25
The Nine Steps of Database Design	26
Identify the Basis for the Database Requirements	27
Starting from an Existing Solution	29
Starting from Scratch	29
Write Down the Basis	29
Define the Database Functional and Performance Requirements	30
Functional Requirements	30
Performance Requirements	31

Contents

Identify the Data Items	31
Separate the Data Elements from the Files	32
Build the Data Element Dictionary	33
Gather the Data Elements into Files	34
Identify the Retrieval Characteristics of Each File	35
Specific Key Retrieval	35
Key Range Searches	37
Boolean Query Retrievals	37
File Scans: Select, Sort, and Display	37
Multiple-File Retrievals	38
Identify the Relationships between Files	39
Develop the Schema for the DBMS You Are Using	42
Summary	42
 CHAPTER 4. THE DATABASE MANAGEMENT SYSTEM	43
The DBMS	43
Commercial DBMS Packages	44
An Alternative to the DBMS Package	46
Database Management System Fundamentals	47
The Data Definition Language	47
Data Models	48
The Data Manipulation Language	52
DBMS Utility Programs	53
Summary	54
 CHAPTER 5. C AS A DATA DEFINITION LANGUAGE	55
Summary	60
 CHAPTER 6. CDATA: THE CHEAP DATABASE MANAGEMENT SYSTEM	61
The Consultant's Billing System	61
The Cdata Data Definition Language	62
The Cdata Data Element Dictionary	63
File Specifications	68
Index Specifications	74
The Cdata DDL Compiler	77
The Cdata DDL	79
Comments	79
DDL Directives	80
Data Element Dictionary	80
File Specifications	81

Index Specifications	81
The Cdata DBMS Header Source File	82
The Compiler, Schema.c	85
Application Software Architecture	110
Cdata File Formats	112
Data Files	113
Index Files	114
Cdata System Architecture	116
Cdata: The Data Manipulation Language Functions	118
Cdata Source Listings	128
Database Manger (cdata.c)	128
File Manager (datafile.h and datafile.c)	141
Index Manager (btree.c)	147
Summary	177
 CHAPTER 7 CDATA UTILITY PROGRAMS	179
Cdata and the Screen and Keyboard	180
Keyboard Header File (Keys.h)	180
System Subroutines (Sys.c)	182
The Database Size Calculator (Dbsize.c)	185
Database Initialization (Dbinit.c)	188
Data Entry and Query (Qd.c)	190
Database File Report Program (Ds.c)	200
Index Builder (Index.c)	203
Some Useful Utility Functions	205
Parse Command Line Data Element List (ellist.c)	205
Screen Manager (screen.c)	207
File Listing Utilities	223
Parse File Name	229
Sorting	231
The Sort Functions	232
The Sort Source Code	233
A Sorting Example: The Client Roster	243
Summary	244
 CHAPTER 8. AN APPLICATION: THE CONSULTANT'S BILLING SYSTEM	245
Consultant's Billing System Requirements	246
Consultant's Billing System Software Design	246
Consultant's Billing System Data Entry	247
Consultant's Billing System Reports	250
The Data Posting Programs	251
The Invoice Program	257

Summary	258
CHAPTER 9. BUILDING THE SOFTWARE	
Preparing Your System	259
ANSI SYS	260
File Handles	260
The Compilers	261
Microsoft C 5.1	262
TopSpeed C 1.04	266
Turbo C 2.0 and Turbo C++ 1.0	270
WATCOM C 8.0	275
Compiler Comparison	279
How May You Use this Software?	280
How Can You Get Help?	280
APPENDIX. CDATA REFERENCE GUIDE	
Cdata Functions	281
Datafile Functions	284
B-tree Functions	286
Console Functions	288
Screen Manager Functions	289
Data Record Display Functions	290
Utility Functions	290
Sort Functions	291
Data Structures	291
Data File Header	291
B-Tree Header	292
B-Tree Node	292
Screen Template Elements	292
Global Symbols	293
Cdata Utility Programs	293
Consultant's Billing System Example Programs	293
Cdata Schema Data Definition Language	294
Index	295
LIST of FIGURES:	
Figure 2-1. The Data Base Analogy.	9
Figure 2-2. The Three Relationships.	18
Figure 2-3. The PERSONNEL Database.	18

Figure 2-4. DEPARTMENTS and EMPLOYEES: One-to-Many.....	20
Figure 2-5. Forming the One-to-Many Relationship.....	20
Figure 2-6. EMPLOYEES and PROJECTS: Many-to-Many.....	22
Figure 2-7. Forming the Many-to-Many Relationship.....	23
Figure 3-1. Retrieval Paths.....	41
Figure 4-1. Managing the Database.....	44
Figure 4-2. The Data Definition Language.....	48
Figure 4-3. The Hierarchical Data Model.....	49
Figure 4-4. The Network Data Model.....	49
Figure 4-5. The Relational Data Model.....	50
Figure 4-6. The Data Manipulation Language.....	53
Figure 5-1. Building a Cdata Application.....	56
Figure 6-1. The Consultant's Billing System Data Base.....	68
Figure 6-2. Addition of the ASSIGNMENTS File.....	69
Figure 6-3. The B-Tree.....	115
Figure 6-4. The Cdata System Architecture.....	117
Figure 8-1. Consultant's Billing System Structure Chart.....	247

LIST of TABLES:

Table 2-1. Personnel System Database Data Element List.....	11
Table 6-1. CBS Data Element List.....	63
Table 7-1. Keyboard Functions for Qd.....	200
Table 9-1. Compiler Comparison.....	279

LIST of SCREENS:

Screen 7.1. Dbize Session for CBS Schema.....	188
Screen 7.2. Dbinit Session for CBS Data Base.....	190
Screen 7.3. A qd Session with the CLIENTS File.....	198
Screen 7.4. General Report Program (ds).....	203
Screen 8.1. CLIENTS File Data Entry Screen.....	248
Screen 8.2. PROJECTS File Data Entry Screen.....	249
Screen 8.3. CONSULTANTS File Data Entry Screen.....	249
Screen 8.4. ASSIGNMENTS File Data Entry Screen.....	249

LIST of LISTINGS:

Listing 6.1	64
Listing 6.2	65
Listing 6.3	66
Listing 6.4	67
Listing 6.5	70

Contents

Listing 6.6	71
Listing 6.7	73
Listing 6.8	75
Listing 6.9 cbs.sch	77
Listing 6.10 cdata.h	82
Listing 6.11 schema.c	86
Listing 6.12 cbs.h	101
Listing 6.13 cbs.c	103
Listing 6.14 invoice.c	110
Listing 6.15 cdata.c	129
Listing 6.16 datafile.h	141
Listing 6.17 datafile.c	142
Listing 6.18 btree.h	147
Listing 6.19 btree.c	149
Listing 7.1 keys.h	181
Listing 7.2 sys.h	182
Listing 7.3 sys.c	183
Listing 7.4 dbsize.c	186
Listing 7.5 dbinit.c	189
Listing 7.6 qd.c	191
Listing 7.7 ds.c	201
Listing 7.8 index.c	204
Listing 7.9 ellist.c	206
Listing 7.10 screen.h	207
Listing 7.11 screen.c	208
Listing 7.12 dblist.c	224
Listing 7.13 clist.c	225
Listing 7.14 filename.c	229
Listing 7.15 sort.h	231
Listing 7.16 sort.c	233
Listing 7.17 roster.c	244
Listing 8.1 Client Report	250
Listing 8.2 Project Report	250
Listing 8.3 Consultant Report	251
Listing 8.4 Assignments Report	251
Listing 8.5 posttime.c	253
Listing 8.6 payments.c	256
Listing 8.7 Invoice Statements	258
Listing 9.1 makefile.msc	262
Listing 9.2 cdata.bld	265
Listing 9.3 makets.bat	266

Listing 9.4	cdata.prj	267
Listing 9.5	schema.prj	267
Listing 9.6	dbinit.prj	267
Listing 9.7	index.prj	268
Listing 9.8	dbsize.prj	268
Listing 9.9	qd.prj	268
Listing 9.10	ds.prj	269
Listing 9.11	invoice.prj	269
Listing 9.12	roster.prj	269
Listing 9.13	posttime.prj	270
Listing 9.14	payments.prj	270
Listing 9.15	makefile.tc	271
Listing 9.16	makefile.wat	275
Listing 9.17	watcdata.bld	278

Preface

C *Database Development* defines a methodology for applying the ANSI Standard C language to designing a relational database and to implementing the programs needed to process the data contained in the database. This methodology depends heavily on the use of reusable C language functions, commonly called tools.

This book will help you build a library of C language database management functions. Such a library can be among the most important of programmer possessions, because Libraries are also software tools, and tools (and tool building) are subjects this book directly addresses.

The subject of tool building was given its first widely read treatment in *Software Tools* by Kernighan and Plauger (Addison-Wesley, 1976). Although Kernighan is co-

author of *The C Programming Language* (Kernighan & Ritchie, Prentice-Hall, 1978) and Plauger is an active participant of the ANSI X3J11 committee that defined the ANSI C Standard, *Software Tools* delivers its message not through C but through RATFOR, an effective language that was destined for quiet obscurity. When *Software Tools* was written, no one knew that the world would soon be stormed by the personal computer and that the programming community would navigate through assembly language, BASIC, and PASCAL to settle upon C as the language of choice. Some of the C programmers from those days knew that such a thing should happen, but no one guessed that it actually would happen. *Software Tools* was written with RATFOR, and a later edition (1981) was published with the tools written in Pascal, but the work has not been published with C as the toolset's language.

Software Tools emphasizes the stand-alone program as a tool. Examples of these programs are word-counting programs, file-sorting utility programs, and text-indexing programs. *C Database Development*, however, takes a different approach to software tool building by emphasizing the use of C language functions as reusable software tools that can be called from your custom applications programs. In particular, it emphasizes those functions that serve a common purpose—the management of databases.

The foundation of tool building is as follows: when you write a program or a function, ask if it could be applied to other applications. If so, write the code to be reusable, and store that code where you can find it when you need it. Later, call the code up and polish it some, perhaps making it even more reusable. Also, seize any opportunity to augment your collection with the tools created by other programmers (within the limits of copyright law, of course). Do all of these steps and before long you will have an impressive tool collection.

The software tool concept is based on the premise that most software systems are built from a set of common components, and those components can be programmed as reusable software modules. The reusable functions that support those common processes are software tools. The more complete your toolset, the fewer programs you will write for each new application. Consider for a moment that many applications systems can be described in terms so general that they cannot be distinguished from many other applications. If you interview programmers for a new job, you will find that the more generally they describe their last assignment, the quicker you will identify with it, and the less you will know about it. If a programmer tells you that he or she designed a system that used a screen driver to collect transactions that were posted against a relational database, you may not know

what the system does, but you might know that you have worked on a dozen other systems that can be similarly described.

To build a tool collection, you must know how to write a reusable function and how to enter it into a library of functions where it is available when you need it again. But, most of all, you must know a potential tool when you see one. Such wisdom should come with experience. After you have written the same function several times, the process will sink in; you should catalog that function to spare yourself the effort of writing it again. After you have had that experience several times, you will learn to look at the functions you are writing to see if they could be reusable tools. One way to learn to be a software tool builder is through the example of another programmer's collection and experiences. In my first book, *C Development Tools for the IBM PC* (Brady, 1986), I explained how the philosophies of programming are centered in top-down design, bottom-up development, structured programming, information hiding, and software tools, and I provided a library of software functions that set those examples. In this book, I try to teach you more of the fundamentals of software systems and give you software tools that support these advanced methods.

First came the idea. Then came the tools. Then came the book. The code in this book was not developed so that a book could be written. The code is for developing systems. All the programs were written to support specific programming assignments. You can survive without such programs, but with them, you can become a better programmer.

Programmers have few qualities that allow them to outperform their fellow programmers. Native ability is one. Special knowledge of an application or a technology is another. Techniques for improved programming productivity are still more. Assume you have superior native ability. But look around you: Genius abounds. You will be admired and hired for your achievements, not for your I.Q. "What can you do for me?" prospective employers and clients will ask. If you are a specialist in some functional application, you will work as long as programs are needed that match your specialty. If you have unique knowledge of a particular language, computer, or operating system, then you will work when someone needs your particular expert skill. But if your specialties are information processing, data structures, data management, user interfaces, and writing working programs, then you will always work because these capabilities are needed in every system development; moreover, if you can deliver a system in less time than it ought to take to build one, then you will be in demand. And that fact is true regardless of the language, the operating system, the computer, or the application.

To establish yourself as a programmer who is in demand, you must acquire the knowledge and skill that will make you the productive programmer that clients and employers want to hire. You need to learn how to build acceptable programs for the people who want those programs.

First, learn what passes for good software—not necessarily what constitutes good software, but what passes for it. If the software does what the user wants it to do, then it passes the most critical test. The quality of a new program has no better measure. The customer paying for the software does not usually read the code and cares little about the programming principles or elegant algorithms that the programmer uses. The customer wants the software to work.

Next, learn how a program stays productive. If it holds up—if it is reliable—then it has the first chance to remain a good program. The second chance comes on that infrequent occasion when the program fails. If you or some other programmer can find the problem and fix it without much extra time and agony, then the program is reasonably maintainable and is still a good program. The third chance comes when the user wants to change the function of the program. Remember that the solution to a problem usually modifies the program, and most programs are going to be changed. If a program is modifiable, then it is a good program because you can change it.

Those attributes just listed are marks of a good program. Another mark is its usefulness, which is measurably increased if the program is ready when it is supposed to be. Some programmers regard their programs as works of art, not to be released until the programs are perfect. Perfection, in the view of some software hackers, is in the internals of the program and not in its usefulness. A program in the hands of one of these birds might be aesthetically beautiful and elegantly coded yet totally worthless because it is not available when it is needed.

If you know how to get a reliable, maintainable, modifiable, and usable program into the hands of the user on schedule and within budget, there isn't much more you need to know.

This book addresses an area of knowledge that many programmers lack: the understanding of data structures and how to implement them.

In the early days of programming, data structures were constrained by the hardware itself. My first programming job in the late 1950s was with the IBM 650, and the

input and output were punched cards and printed listings. The user was the computer operator, and the user interface was a panel of lights and switches. (That old 650 was visited more by the person who replaced vacuum tubes than by the operator.) Our data structures consisted of 80-column punched cards and 120-column printed listings. When we moved up to tape and disk in the so-called "second generation," the operator had a hard-copy typewriter console and more lights and switches, but we programmers had tape and disk—two new kinds of mass storage to deal with—and we had to start thinking about more complex data structures. We sank to the challenge. We invented data models involving complex header and trailer records loaded with inter-record and inter-file pointers and all kinds of spaghetti logic. Variable-length records were common. Hashing algorithms used strange formulae to compute record addresses from various parts of the data values and provided the techniques for random access. It was usual to have many records hashing to the same address, so we invented collision strategies that involved overflow areas and more pointers and secondary hashing algorithms. Whenever some new requirement would surface, the whole programming staff went into a dither about how much software needed changing; the new wrinkle never seemed to fit into the old data model.

The main problem was that we didn't have a database management system. We didn't even have file servers. The specific tape and disk read, write, and error correction routines existed in code at the hardware level and were in every program. Disk volumes didn't have file directories; they had files positioned wherever we wanted them, and every program knew the physical sector addresses of the files. If the volume needed reorganizing, scores of programs were changed and retested.

If all that chaos seemed normal, it was because at the time it was. Everyone in the field was going about programming in the same way. Most of us were too busy building systems to stand back and look at the business of building systems. The scholars and researchers sought better ways, however, and the first ideas about general-purpose, reusable software were born. First came device macros that handled input, output, and error checking. Next came the file servers that would find a place for a file and let it grow without involving the applications programs. New ideas poured out. Among them was the notion of the logical device, where the program talked to a symbolic device, and the operator assigned the reel of tape or the disk cartridge to a physical device and then mounted it accordingly. That idea matured when operating systems began to translate the program's logical device to a physical device.

Then came the database management system (DBMS). The idea behind the DBMS was that you could describe an application's database in a format that was external to the application's programs, and a general purpose software system would tend the database by adding, deleting, and retrieving records at the request of the programs. From the DBMS came the need for standards for expressing the format of a database, and from these standards came the idea that most or all databases could be described from one of a small set of data models.

The development of a common set of data models was strongly influenced by all the wacky and convoluted data structures that programmers had been building for so long. From the disorder came a vision of order. It became apparent that discipline and structure must be applied. Without it, the DBMS would not be possible. The quest for order resulted in the development of three data models: the hierarchical model, the network model, and the relational model. Any data structure problem could be solved with one of these models. (You will find discussions of these data models in Chapter 4.)

At last programmers had some standards to follow. They had to follow them because they were being given these DBMSs to use, and each one would support only one of the models. Programmers could not repeat their past sins with contrived data structures because the tools at hand wouldn't support them.

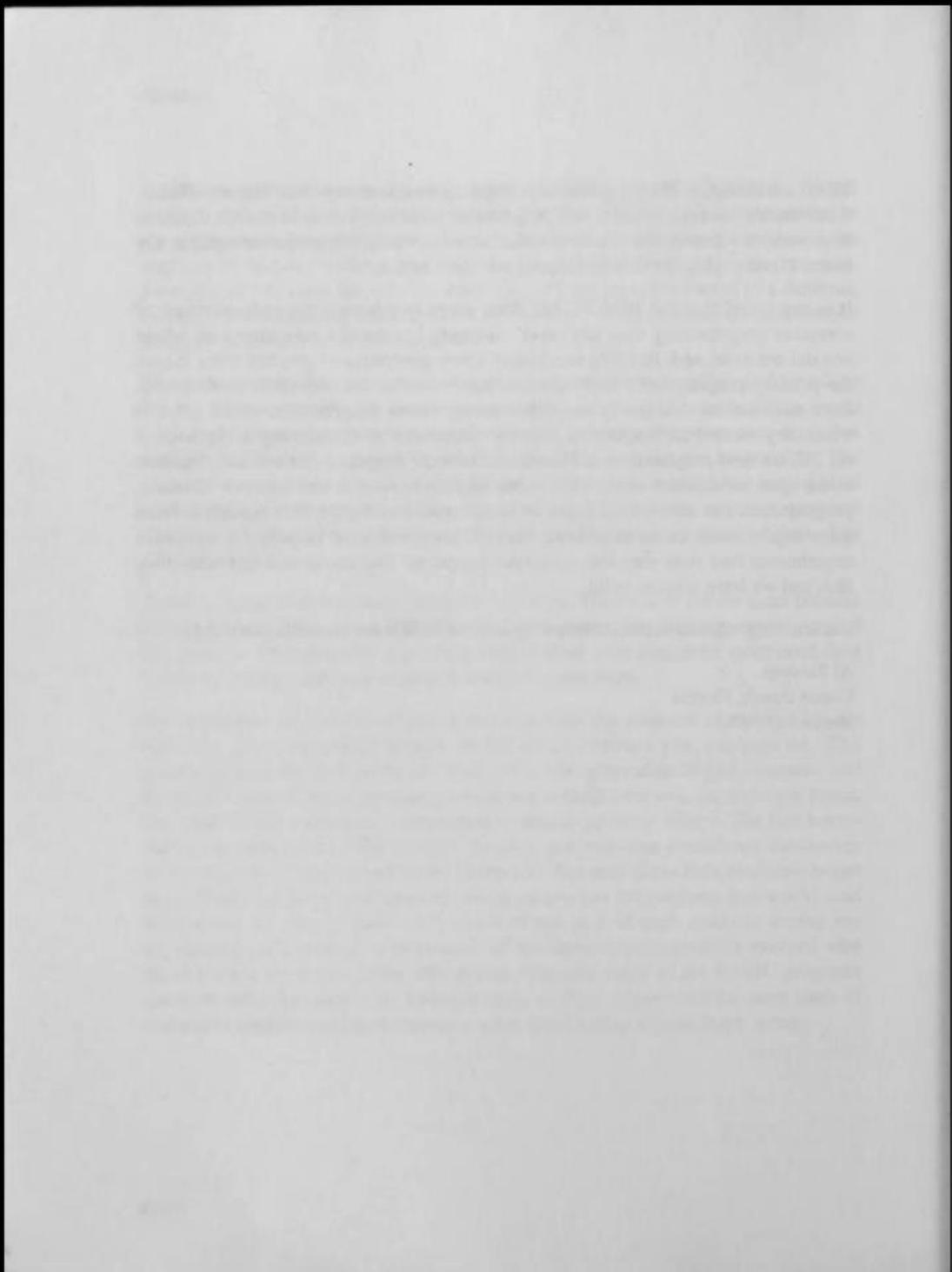
So where does all this development leave us with the personal computer? Do we have the same structural standards for data? Perhaps yes, perhaps no. The proliferation of the little computer spawned a new generation of programmers, and the cycle repeated. When programmers started to build their own computers at home, they didn't build a database management system to go along with it. The first home-built computers weren't big enough. Besides, programmers considered themselves lucky to get their hands on a BASIC interpreter. Not until these little machines began to get bigger (in power and memory) did programmers get products that would send them down the correct path. As a result of the lack of such products during the microcomputer's infancy, a generation of homegrown programmers evolved who never learned the lesson of the data model. Consider many of the BASIC programs out there today. Lurking in the hearts of many of these systems are the same kinds of weird data models that the old-timers used to build before anyone knew better.

Times are changing. If a programmer is to produce software systems that are reliable, maintainable, and modifiable, that programmer must know how to build a database from accepted data structure techniques. Hacked-out solutions are not acceptable any more. Those paying the bills have come to expect better.

It is my belief that the IBM PC has done more to advance the understanding of computer programming than any other computer. It created a consumer base where one did not exist, and, in doing so, created a new generation of programmers. It took the primary programmer's tool—the computer—out of the air-conditioned, raised-floor environment and put it on kitchen tables where programmers could get at it when they wanted to. Because of the new consumers' overwhelming acceptance of the PC, the new programmers are motivated to write programs that will sell. Because of the open architecture of the PC's video display, keyboard, and interrupt structure, programmers can allow their creative imaginations to explore the machine. From these explorations come excursions into different software techniques, and these experiments find their way into consumer programs. The consumers buy what they like, and we learn what to build.

But knowing what to build and knowing how to build it are two different things.

Al Stevens
Cocoa Beach, Florida
December 1990



Chapter 1

Introduction

C Database Development explains how to apply the ANSI Standard C language in defining a relational database and in implementing the programs needed to process the data within the database. *C Database Development* also contains an introduction to the fundamentals of database management, how to use Standard C to develop programs that manage information in a database, and a brief history of how today's database concepts have evolved.

The text is accompanied by a significant body of source code. This code constitutes a complete relational database management system, or DBMS, with which you can build systems that manage data in the orderly formats known as databases.

This is the second edition of C Database Development and the source code herein is written exclusively in Standard C. The only compiler-dependent expressions are

1 Introduction

found in the makefile compiler commands that build the executable software from the source code. The first edition, published in 1987 by MIS Press, used the conventions of the C language that existed before the ANSI Standard definition of C was approved in 1990. The book addressed a number of compilers that ran under DOS on the IBM PC. Because the compilers were different in their implementations of C and no standard existed, the code in the first edition employed a number of compiler-dependent conditional expressions.

The software in this book consists of functions that the programmer includes in a software system, which, in turn, are tools that become part of the total application once it is operational. By supporting requirements that are common among most systems, the functions reduce the amount of code you must write when you are developing a new application. These tools represent a layer of software that sits between the custom application code and hardware and the operating system. As such, they provide support to higher levels of common system requirements and reduce the quantity of code you develop. When you use proven and reliable utility code—or software tools—instead of writing one-of-a-kind custom functions, your software system gains a significant measure of reliability.

Programming should include the philosophy and disciplines of portable software, top-down design, bottom-up development, structured programming, information hiding, and, above all, reusable software tool building. These measures are sound, and they will serve you well if you remain faithful to their principles.

If you develop software tools that can be ported to different computers, you are, through your portable tool collection, a more productive programmer. Some programmers go from assignment to assignment without taking as much as a program listing. The work they do in each assignment remains behind, and they start a new project with only their skill and knowledge. Perhaps with the example of a useful tool collection, they could have learned to build reusable software to carry over to future applications. This book sets that example.

You will find plenty of books about C on the Computer Science shelves in book stores. But even though most of them serve useful purposes, they seem to fall short of the mark when it comes to providing usable C functions. Many don't shoot for that particular mark, but the ones that do seem to miss.

A typical book explains the C language and provides examples of how to use its features. Some books provide simple functions to illustrate algorithms and the language principles. Often, the functions are similar to those available in the libraries

of compilers. More often, they are textbook examples without much practical use. An author needs to deliver his message in as brief and concise a manner as possible. Not only are large software systems difficult to publish—because of the size of the source files—but they are difficult to explain. Complex algorithms need simple explanations, and comprehensible discussions of intricate subjects are difficult to develop. Even so, the objective of this book is to provide complete, useful software tools—easy to use and understand tools that you can put to work in the next software system you develop. You will be able to actually use the software published here, and you will be able to apply it and your new knowledge in developing database systems.

To use this book, you need a programmer's understanding of the IBM PC and the DOS software development environment. You need to understand the DOS hierarchical file structure and the associated paths and file directories and subdirectories. Several good texts are available that will teach you this information. *Teach Yourself DOS*, written by this author and published by MIS Press in 1989, is a user's introduction to DOS. Two programmers' books that are recommended are *The Peter Norton Programmer's Guide to the IBM PC* (1985) by Peter Norton and *Advanced MS-DOS* (1986) by Ray Duncan, both published by Microsoft Press. You should keep these two books handy whenever you are writing programs that reach beyond the standard C function library.

This book does not attempt to teach you how to program in Standard C; instead, it assumes that you are already a C programmer. Perhaps you could install this software without ever learning the basics of the C language, but you would never be able to use the functions for anything beyond the applications examples published here. Even though the examples are functional and useful software packages, they themselves are not the point of this book. Their purpose is to illustrate the use of the toolset functions; if you can use the examples as applications, then all the better. But you need more. Two elements essential to a productive programming career are an understanding of software tool building as a discipline and a collection of software tools to use. If this collection shows you by example that tool building is a good idea, the first purpose of this book has been successful. If you can use the tools in your work, the second purpose has been realized. Perhaps the functions published here will be a significant addition to your personal library of software tools. If so, the book has met its objective. If that addition turns out to be a starter set—if you have not yet begun tool building—then the book has served an even greater purpose; you are about to become a better programmer.

DATABASE MANAGEMENT

The database management system found in later chapters is written as generic software that you can move into many environments without undue concern for the hardware or operating system used. The package has worked on several varieties of computers, ranging from the Z80 to the 80386 microprocessors, under four operating systems: DOS, CP/M-80, CP/M-86, and TurboDos. This system has come together into a single package that operates on the IBM PC under DOS but is portable to other environments. The PC version is published in this book.

CHAPTER OVERVIEWS

Chapters 2 through 8 address the database and the database management system (DBMS) that support the software applications you develop. These chapters include a library of functions that, when called from your applications programs, provide most of the facilities of a DBMS. With this knowledge and the toolset, you can write software systems that process data organized in the relational model. The relational database theory is discussed in Chapter 4.

Chapter 2 explains database principles by addressing each of the components of data that constitutes a database. It then proceeds to show how you describe a database by using a language known as the database schema. You will learn to design databases as logically interrelated files, each containing data elements. You will also learn to index the records in the files by the values of chosen data elements called keys. You will then see how the data element dictionary is the basis for all related data definitions.

Chapter 3 discusses how to design a database from scratch. You can build a database design by collecting the requirements for data storage and by taking an inventory of the data items that must be stored and retrieved.

Chapter 4 addresses the DBMS itself—the system of software that supports your data by providing functions to describe the architecture of the database, to store and retrieve records in response to calls from your programs, and to maintain the integrity and continuity of the indices into the record contents. This chapter contains the essence of this book's DBMS approach and assumes that the reader has a moderate knowledge of C syntax.

Chapter 5 discusses the features of the Standard C programming language as you can apply them to the definition of a database. The technique described allows you to change the database format without concern for most of the software that processes it.

Chapter 6 introduces the "Cheap Database Management System," called Cdata. With Cdata you develop the specifications of a database by using the techniques addressed in Chapter 5. Chapter 6 includes a schema compiler program that reads a textual description of the database schema from an ASCII text file and generates the Cdata C language data definition language. This subject is followed by a discussion of Cdata's data manipulation language functions. You use these functions to store and retrieve records to and from the database.

Chapter 7 presents a series of utility functions to manipulate data in the database. These functions include programs to enter data into the file records, retrieve records by indexed key data elements, and generate reports based on data records retrieved from the database.

Chapter 8 presents an example of a complete—although small—software system that builds a consultant's billing system with the Cdata DBMS.

Chapter 9 explains how you can build the software published in this book by using one of the compilers. The procedure for each compiler accompanies an appropriate **makefile** to build the toolset functions and the example applications. You can compile the software in this book with these five Standard C compilers for the IBM PC:

- Microsoft C 5.1
- TopSpeed C 1.04
- Turbo C 2.0
- Turbo C++ 1.0
- WATCOM C 8.0

The Appendix is a reference guide to the programs, functions, global symbols, and the Cdata schema language.

SUMMARY

The functions in this book are useful, informative, and interesting programs. But, best of all, they are fun. Developing tools for fellow programmers is an ideal assignment for a programmer because it embraces a kindred clientele. A long programming career will know many different user communities, but most of us don't understand what our users do for a living; however, we do know the pains and pleasures of the programmer's lot. Some of us have known them through several generations of computers and have waded through many iterations of discovering and rediscovering the principles and practices of the craft. I hope that you will find as much enjoyment in studying and using these programs as I have found in developing and writing about them.

Chapter 2

Database Fundamentals

The next three chapters discuss the principles of database architecture and design and the database management system (DBMS). If you are an experienced database analyst, you might want to skip these chapters, but you are encouraged to read them all. Although none of the three is comprehensive, together Chapters 2 through 4 can help you understand the software presented in the remainder of the book because they reveal a perspective of database theory that may differ from your own. Many so-called standards of the software industry are defined and redefined by popular use rather than by studied application. When a technical phrase such as "relational database" becomes fashionable, its meaning can be lost when those who do not know its original definition use it extensively. Zealous advertising, uninformed media treatment, and layperson involvement can result in the unintended redefinition of a phrase.

The job of a system designer/developer is complex. His or her objective is to produce a useful automated system. To do this, the designer must provide for an orderly, disciplined definition of the system's components. An automated system contains many elements or subsystems including requirements analyses, hardware, software, user and operator documentation, maintenance, and training.

This book is about software. Software consists of programs and information; good programs need information to process, and information requires effective processing to be useful. To make information suitable for processing, you must organize it into a disciplined and consistent form called a database.

In this chapter, a database is defined, and a small personnel database is presented as an example. Chapter 3 explains the process of database design and uses the example personnel database to illustrate the steps taken in its design. Chapter 4 explains the database management system and how to build the database design into the software system that will manage it.

THE DATABASE

Definition:

A database is an integrated collection of automated data files related to one another in the support of a common purpose.

Each file in a database is made of data elements—account numbers, dates, amounts, quantities, names, addresses, hat sizes, and many other identifiable items of data. These data elements are to be organized by, stored in, and retrieved from the database. To design such a database, you must understand files, their data elements, and how they are used.

To a designer, data means automated information, i.e., information in a format acceptable for automatic processing by a computer. Information can be automated when it can be reduced into a machine-readable form.

The smallest component of data in a computer is the bit—a binary element with the values of 0 and 1. Programmers are sometimes concerned with bits but, as the database designer, you will view data from a higher perspective. Bits are used to build bytes (or characters), which are used to build data elements. Data files contain records that are made up of data elements, and a database consists of files. Take time

to learn the hierarchy of data in a database because it figures significantly in the rest of this book. Starting from the highest level, the hierarchy is as follows:

1. database
2. file
3. record
4. data element
5. character (byte)
6. bit

At this stage in the design, the first five levels in the hierarchy are of primary interest. The database contains files of records that contain data elements made up of characters which express the data values.

Figure 2-1 shows a popular analogy to the database—the office file drawer. The drawer represents the database, the folders represent the files, the documents in the folder represent the records, and the information on the documents represents the data elements. The analogy is so popular that some database systems use file cabinets, drawers, and folders as video icons (small graphic pictures) to represent components of the database.

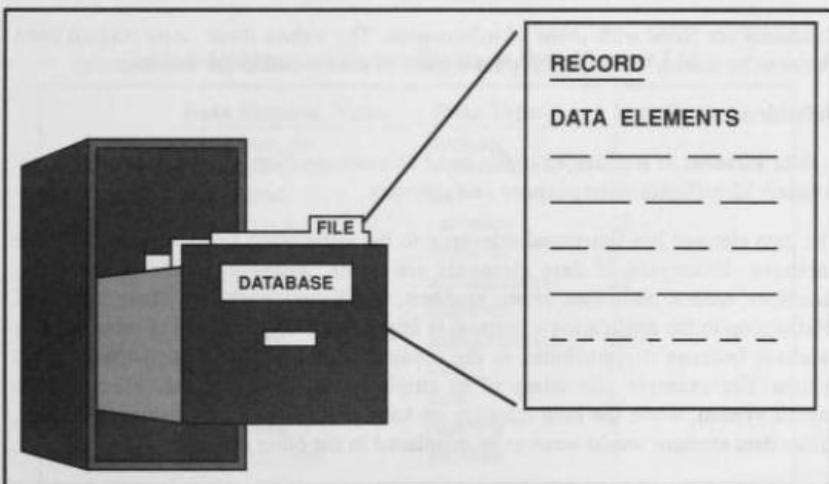


Figure 2-1. The Data Base Analogy.

The analogy is not perfect, however. The paper records in file drawers are usually stored without a strict discipline. Of course, an efficient file clerk will attempt to maintain order, but his or her attention provides the only control. In an automated database, the controls are designed into the database and the software. Each item of data serves a specific purpose, and information is not unnecessarily repeated. Everything needed is there, and anything that does not fit is not allowed. The records in a file all have the same format, with identical sets of data elements in all records in the same file.

To design a database, you must understand data structures and interrelationships. The document (record) in a folder (file) has form and structure, and the folder contains many copies of the same document with different information (data values) filled into the blank entry spaces (data elements) of the document. The many folders are collected in a drawer (database), representing the collection of information. The file cabinet itself is analogous to the total information resource of an organization.

You can begin to understand the database by examining the lowest addressable item of automated data—the data element.

THE DATA ELEMENT and DATA VALUE

Databases are filled with items of information. The values these items contain need places to be stored. The repository for a piece of data is called the data element.

Definition:

A data element is a place in a file used to store an item of information that is uniquely identifiable by its purpose and contents.

The data element has functional relevance to the application being supported by the database. Examples of data elements are dates, account numbers, amounts, quantities, names, addresses, phone numbers, and batting averages. Their functional relationship to the application's purpose is important: a data element is recorded in a database because it contributes to the context of the functions supported by the system. For example, the salary of an employee is a relevant data element in a payroll system, while the item quantity on hand is relevant in an inventory system. Either data element would seem to be misplaced in the other system.

The data element is the temporary home for a transient value of information. Do not confuse the data element (the place for the data to be stored) with the data value (the current contents of the data element). The employee record will always contain the data element for the employee's salary, but that data element will have different values for different employees, and the value for a given employee will change from time to time.

Definition:

A data value is the information stored in a data element.

The Data Element Dictionary

Definition:

A data element dictionary is a table of data elements including at least the names, data types, and lengths of every data element in the subject database.

A database design task often begins with the development of a list of data elements. Whether or not you begin with such a list, you should eventually create one. Table 2-1 is an example of a list of data elements for a small personnel system database. This example illustrates the foundation of database design—the data element dictionary.

Table 2-1. Personnel System Database Data Element List.

Data Element Name	Data Type	Length
employee_no	numeric	5
employee_name	alphanumeric	25
date_hired	date	6
salary	currency	10
dept_no	numeric	5
dept_name	alphanumeric	25
proj_no	numeric	5
proj_name	alphanumeric	25
start_date	date	6
completion_date	date	6
labor_budget	numeric	5
hours_charged	numeric	5

The data element dictionary is central to the application of the database management tools presented in this book. You must define all data items that appear on screens, reports, and in data files in the data element dictionary. An exception to this rule is the constant, literal information used on screens and reports for report titles and input prompting messages. You do not need to define data elements for this information, although some database management systems do not distinguish such information from real data elements.

Data Element Types

The data element dictionary just shown includes names, types, and lengths of the elements. The types are from the following list:

- Date
- Currency
- Numeric
- Alphanumeric

These element types are not the only ones you might use, but you can describe any data element in a commercial application with one of these types. Other generic data element types are phone number, zip code, state, social security number, and so on. The extent to which you can support generic data element types varies from system to system. One of the most popular (and expensive) mainframe database management systems allows you to define data elements as numeric or alphanumeric only, allowing no more specific generic types. Often, generic data element types are insufficient to describe data without further definition. Some generic types imply form and function, but the implication is not rigid. The implied format of a date or a phone number seems obvious, but you will encounter any of several formats for these two simple types. Consider the following examples:

<u>Telephone numbers</u>	<u>Dates</u>
(202)555-1212	01/02/86
PE 6-5000	3 Jan 64
Dial 'M' for Myrtle	The Ides of March

Data Element Names

Users and computer programs alike use the names of data elements in data element dictionaries to identify the data elements. The names describe the function and purpose of the data elements they name. If you have the common programmer's bent for cryptic and short data element names, you are encouraged to reconsider that practice. In a database management system built from the tools in this book, the user as well as the programmer sees the data element names. They must be readable by human beings as well as by machines.

The software in this book uses C language data identifiers as data element names. These identifiers are unique to the first 31 characters, providing sufficient room for a descriptive name. Chapter 5 explains how to apply this technique.

Data Value Representation

Data elements in a database will often exploit the data types of the computer or language to preserve storage space or time. You might find a mix of binary integers, floating point numbers, strings, and bit flags combined in the same file. In this book's approach to data storage, all data element values are stored as null-terminated ASCII strings. This practice has advantages. The data can be viewed from system utilities that display file data on the screen (TYPE, DUMP, etc.). The use of ASCII character strings allows the data formats and the programs that process them to be portable between computers with different internal data formats. You need not concern yourself with the alignment differences between computers and compilers and you need not worry that the language or the operating system will treat a particular bit pattern as something inappropriate, such as an end-of-file mark.

FILES

A database contains a group of files that are related to one another by a common purpose. Do not confuse the file and the record. The file is a collection of records. The records in a file are alike in format, but each individual record is unique in its content; therefore, the records in a particular file all have the same data elements but different data element values.

Definition:

A file is a collection of records that share a format but that each have unique data element values.

You can derive the format of a record from a list of the data elements that appear in the file.

The organization of a file provides functional storage of data related to the purpose of the system that the database supports. Interfile relationships are based on the functional relationships of their purposes. To describe a file, you give it a name and supply a list of data elements. The name should relate to the purpose of the file, just as the name of a data element describes the functional meaning of the data in the element. If you are designing an employee file, it makes sense to call the file something like "EMPLOYEES." Following is an example of how you might specify the file structure:

EMPLOYEES:

employee_no, employee_name, date_hired, salary

If you refer to the data element dictionary (shown in an earlier example) for the lengths of the data elements, you can see that the sum of the lengths of the elements in EMPLOYEES is equal to 46 characters. The C language terminates an ASCII string of data with a null-value byte, so you must add one character to the length of each data element, bringing the total to 50, which is how long a record in the hypothetical EMPLOYEES file would be.

THE DATABASE SCHEMA

When designing a database, you must prepare a formal description or specification of the files, the format of each file and the relationship(s) between files (if any). The techniques for preparing this specification will vary according to the style preferred by the developer, but such specifications must include at least the files, their data elements, the data elements in each file used to identify specific records, and the implied relationships between files. This description of a database is called its "schema."

Definition:

A schema is the expression of the database in terms of the files it stores, the data elements in each file, the key data elements used for record identification, and the relationships between files.

The small database used in these discussions is a relational database. The techniques for record indexing and file-to-file relationships come from the relational theory of database management. Other models of data are available, and the data models you might use are discussed in Chapter 4. Since this book includes database management software and because that software supports the relational model, these discussions will lean in that direction.

The Specification of Data Files

In its simplest form, a database is a set of files that share a purpose and the specifications for that database must accurately reflect that purpose. The purpose of the previous example (the file named EMPLOYEES) was to list employees and the database file specification reflected that purpose. If the purpose of the database were to be expanded to include other personnel data, such as departments and projects, the database specification would have to be modified to reflect the new purpose. This modified database (now named PERSONNEL) might be specified as follows:

PERSONNEL:	
EMPLOYEES:	employee_no, employee_name, date_hired, salary
DEPARTMENTS:	dept_no, dept_name
PROJECTS:	proj_no, proj_name, start_date, completion_date, labor_budget

The Data Definition Language

The translation of a schema into a database management software system usually involves using a language to describe the schema to the database management software. This language is sometimes called the "Data Definition Language." In this book's approach to database definition, the facilities of the C language will be used to build a Data Definition Language, as discussed in Chapter 4.

A schema contains two more data constructs to complete the database design: the individual file index key specifications and the interfile relationships.

Key Data Elements

The records in every file must be uniquely identified so the system can find and retrieve each one. You need a way to tell the computer which record to retrieve, and the computer needs to know how to find a requested record. Some file management

systems allow you to retrieve a record by calling out its position in the file or record number, but this method isn't adequate. You don't want to remember where a record is stored, just that it is in the file and available to you. It is, however, reasonable to expect you to remember something about the data in the record you want to retrieve. Given what you tell the computer about the data, the computer can find the record by searching for one that contains the provided data value.

The definition of a file includes the specification of the data element or elements that are the **key** to the file. A file key logically points to the record that it indexes.

Because a key is an index into the file, database management systems use keys for file retrievals. With indexed data elements, more than one key can be assigned to a file, which means that you can retrieve a record based on the values of several data elements. Even so, the file will usually have a single, unique key that distinguishes each record from all others in the file. This key is called the "primary key" because only one record in a file can contain a given value of a primary key.

Definition:

The primary key data element in a file is the data element used to uniquely describe and locate a desired record. The key can be a combination of more than one data element.

In the example EMPLOYEES file, the **employee_no** data element is the primary key. Non-primary keys that index a file are secondary keys, and more than one record can contain the same value. Initially, the database designer would identify only the primary data elements for the files in the PERSONNEL database. To do this, you repeat the the database schema with the primary key data element emphasized in each file. The following example shows the primary key data elements emphasized by angle brackets:

PERSONNEL:	
EMPLOYEES:	<employee_no>, employee_name, date_hired, salary
DEPARTMENTS:	<dept_no>, dept_name
PROJECTS:	<proj_no>, proj_name, start_date, completion_date, labor_budget

You can find employee records if you know the employee number. Similarly, department records are keyed on the department number, and the project number indexes the project file.

Interfile Relationships

The schema developed for the PERSONNEL database does not show how the files are related. The ability to maintain the relationships between files is one of the strengths of a database management system. Files are related when a data record in one file associates with a data record in another file. When employees are assigned to projects, the record for employee Oppenheimer (in the EMPLOYEES file) could be related to the record for the Manhattan project (in the PROJECTS file). It is not necessary that all employees be assigned projects or that all projects have employee assignments for the relationship to exist between the files. You could have a lot of employees, a lot of projects, and no project assignments, and the relationship would still exist if the database design provided for the potential assignment of employees to projects. In the example PERSONNEL database, no potential is shown for files to be related, but that capability will be added to the database schema next.

In a database, you can relate one file to another in one of three ways:

- one-to-one
- many-to-one
- many-to-many

Consider the relationships involved in the records of teachers and students. When students hire full-time tutors to better learn a subject, a one-to-one relationship occurs between a file of tutors and a file of students; a tutor has just one student and a student has just one tutor. A database that supports the tutor-to-student relationship will be said to support a one-to-one relationship between the two files. Even though the database has many students and many tutors, the relationship is one-to-one because each tutor or student has only one of the other.

In elementary school, one teacher has many students, and each student has only one teacher. This constitutes a one-to-many relationship.

In high school and college, a teacher teaches many students, and each student has many teachers. This is an example of a many-to-many relationship.

Figure 2-2 uses this example to illustrate how relationships between teachers and students can be depicted. To design a database with teacher-student relationships, you would start with a chart similar to one of the three parts of Figure 2-2. Each box represents a file and each connecting arrow represents an interfile relationship. The arrows have single and double arrowheads connecting the files. A single arrowhead points to a "to-one" relationship; a double arrowhead points to a "to-many" relationship.

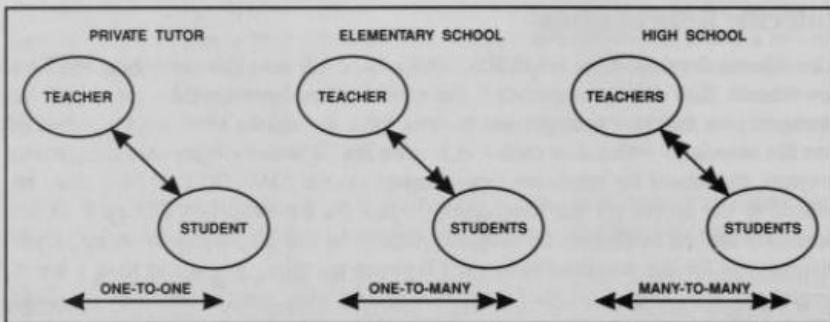


Figure 2-2. The Three Relationships.

Figure 2-3 shows the relationships in the PERSONNEL database, using the arrowhead symbols.

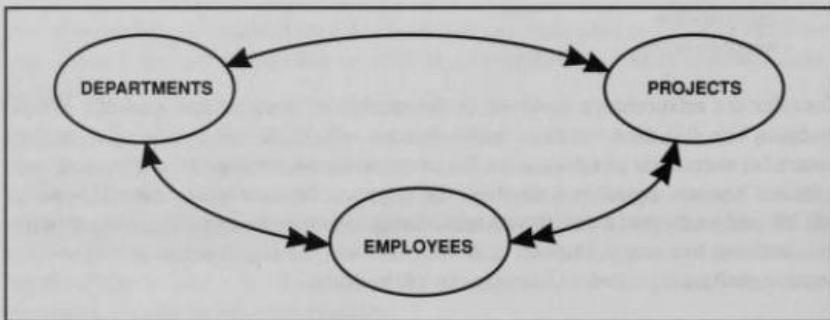


Figure 2-3. The PERSONNEL Database.

Each department works on only one project, and each project belongs to only one department; therefore, the relationship between the DEPARTMENTS file and the PROJECTS file is one-to-one.

A department employs more than one employee, and an employee can work for only one department, so the relationship of the DEPARTMENTS file to the EMPLOYEES file is one-to-many. Conversely, the EMPLOYEES file has a many-to-one relationship with the DEPARTMENTS file.

A project has more than one employee assigned to it, and an employee can work on many projects, so the EMPLOYEES file and the PROJECTS file have many-to-many relationships with each other.

The technique for describing these relationships is not complicated. A file has a primary key data element. If another file has a many-to-one relationship with this file, the other file includes (among its data elements) the primary key data element of the first file. Following is the PERSONNEL database schema with key elements—indicated by asterisks—repeated in the files where the relationships exist:

PERSONNEL:	
EMPLOYEES:	<employee_no>, employee_name, date_hired, salary, *dept_no*
DEPARTMENTS:	<dept_no>, dept_name
PROJECTS:	<proj_no>, proj_name, start_date, completion_date, labor_budget, *dept_no*

Notice that the data element **dept_no** has been added to the EMPLOYEES file and the PROJECTS file. As a result, these files have a many-to-one relationship with the DEPARTMENTS file. Suppose that the relationship between departments and projects is one-to-one because a department may not run two projects and departments may not share a project. You can use two techniques to support this relationship: (1) concatenating—physically bonding—the two records into one; or (2) enforcing the one-to-one circumstance with the applications software that adds records to the PROJECTS file. A record addition is not permitted if another project is already assigned to the department.

Both approaches have their disadvantages, however. The first approach requires that all department records have space for project data, even when the department has no projects. The second approach requires the applications software to enforce the relationship. This approach has potential for loss of data integrity due to careless programming; it works better if the database design includes inherent integrity constraints that a program cannot violate. Which approach you choose will depend on the problem at hand and the relative merits of each approach for your applications.

Figures 2-4 and 2-5 illustrate how to build the one-to-many relationship. The two files become related when you place **dept_no**—the primary key to the

DEPARTMENTS file—in the EMPLOYEES records as a data element. Subsequently, each EMPLOYEES record can point to one DEPARTMENTS record, and, since more than one EMPLOYEES record can point to a DEPARTMENTS record, the design supports the one-to-many relationship. Figure 2-5 shows the two files with data values in the records. Observe that while two of the EMPLOYEES records (**employee_no.**'s 00002 and 00005) contain the same **dept_no** (0020), no EMPLOYEES record can relate to multiple DEPARTMENTS records.

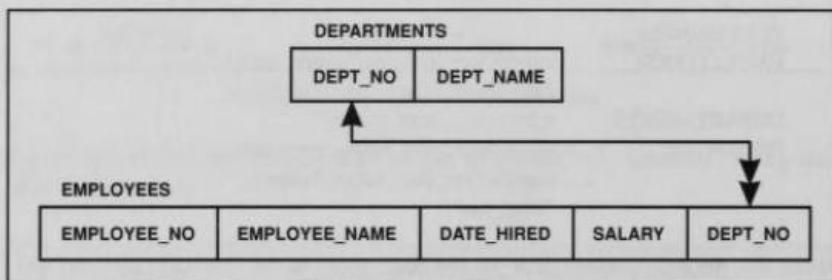


Figure 2-4. DEPARTMENTS and EMPLOYEES: One-to-Many.

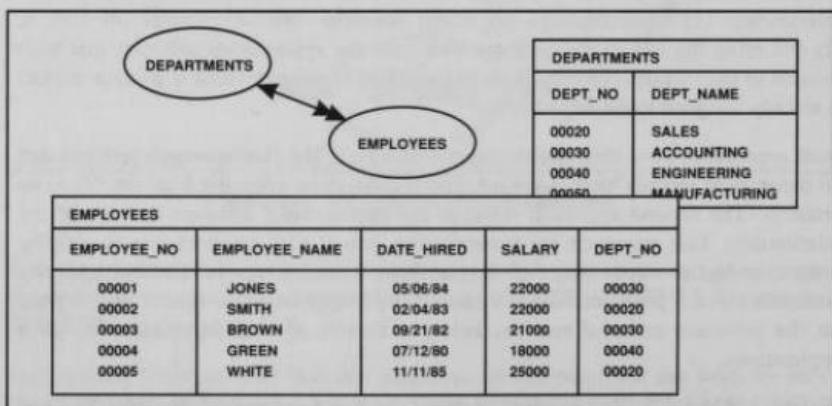


Figure 2-5. Forming the One-to-Many Relationship.

There is still one relationship to establish. The PROJECTS file and the EMPLOYEES file have a many-to-many relationship, which is not as easy to build as the other relationships. At first glance, you might think that you could put the **proj_no** data element in the EMPLOYEES file and the **employee_no** data element in the PROJECTS file, but this will not work because a project could point to an employee that points to a different project. Instead, you must build an additional file whose purpose is maintaining the relationship. This technique is known as a connector file because the records in the new file produce a logical connection between the records in two other files. Usually, however, you can find additional functional purposes for the file beyond its role as a connector for the other two files. Information often exists that is relevant to the relationship itself but is without meaning when applied to either of the participants alone. For example, in the relationship between employees and projects, you may want to record the hours each employee spent on each project. The expanded schema, with its new file for employee-project assignments, now looks like the following:

PERSONNEL:
EMPLOYEES: <employee_no>, employee_name,
 dept_no
DEPARTMENTS: <dept_no>, dept_name
PROJECTS: <proj_no>, proj_name, start_date,
 completion_date, labor_budget,
 dept_no
ASSIGNMENTS: <employee_no, proj_no>, hours_charged

Notice that the **employee_no** and **proj_no** data elements are identified as a single key. These two items are concatenated into the primary key of the file because both items combine to uniquely describe a record. The file can contain many records with the same employee and many records with the same project, but only one record can have a given combination of **employee_no** and **proj_no**.

Figures 2-6 and 2-7 are examples of many-to-many relationships. In Figure 2-6, the PROJECTS and EMPLOYEES files are unchanged from the earlier schema, which supported no relationships. The addition of the ASSIGNMENTS file to the schema provides the ability to account for hours charged by each employee to each project. In addition, it supports the many-to-many relationship between PROJECTS and EMPLOYEES. The ASSIGNMENTS file has a many-to-one relationship with each of the PROJECTS and EMPLOYEES files because it has their primary keys as data elements. So a given ASSIGNMENTS record can point to one PROJECTS record and one EMPLOYEES record. Further, a record in either of the PROJECTS or EMPLOYEES files can be pointed to by many ASSIGNMENTS records. The many-to-many relationship is built through these two one-to-many relationships. From a given EMPLOYEE record, you can find all the ASSIGNMENTS records that point to it by extracting the ASSIGNMENTS records containing the pertinent employee_no. Then, you can derive the many PROJECTS related to the EMPLOYEE by using the proj_no data element from the ASSIGNMENTS records. Conversely, the many EMPLOYEES related to a PROJECT can be determined with a similar retrieval through the ASSIGNMENTS file.

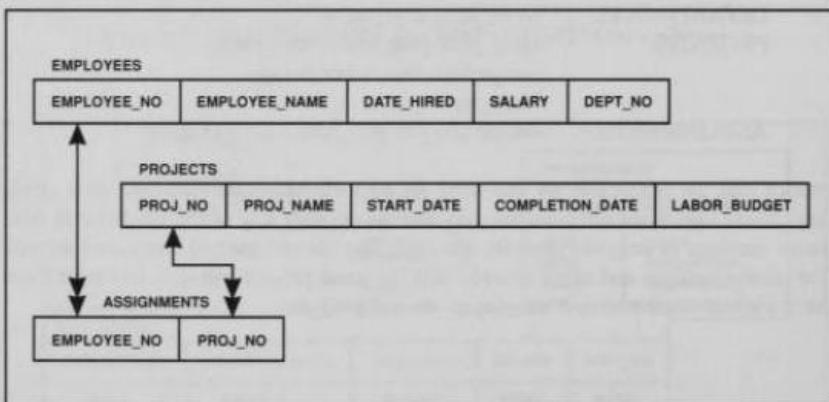


Figure 2-6. EMPLOYEES and PROJECTS: Many-to-Many.

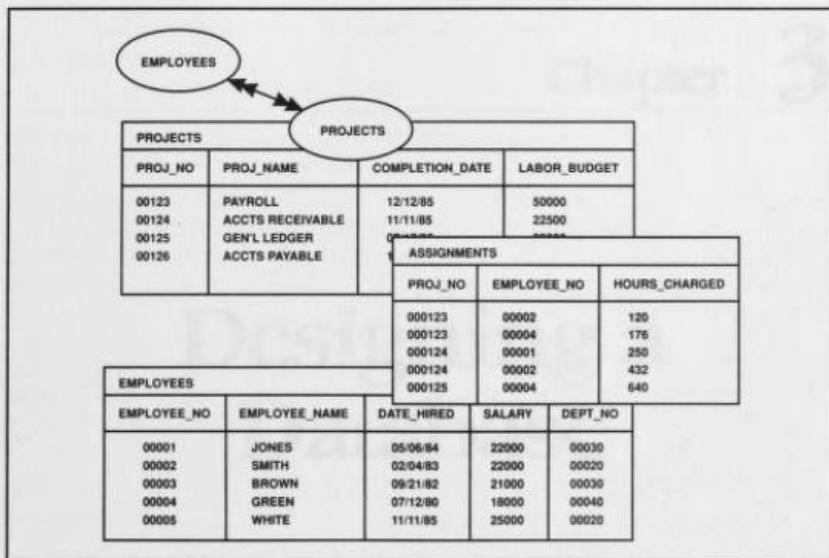


Figure 2-7. Forming the Many-to-Many Relationship.

Figure 2-7 shows the three files as they might look with data values included. You can see how the ASSIGNMENTS file associates the 00123 proj_no with employee_nos 00002 and 00004, and employee_no 00004 with proj_nos 00123 and 00125. Thus, a many-to-many relationship is maintained through the connector properties of the ASSIGNMENTS file.

These three relationships are the only ones you need to be concerned with. You can store, retrieve, and maintain anything you might ever want to store in a database by using one of these three models.

SUMMARY

This chapter described a database and discussed its components. The next chapter addresses the often misunderstood and frequently overstated process of designing a database from scratch.



THE UNITED STATES GOVERNMENT'S POSITION

It is the position of the United States Government that the right of the people to keep and bear arms is a fundamental right guaranteed by the Second Amendment to the Constitution of the United States. This right is not limited to hunting and sport, but extends to all other forms of self-defense, including the defense of the home and family. The right to keep and bear arms is a fundamental right that must be protected from encroachment by any level of government.

RECOMMENDATION

Given the above analysis, it is recommended that each state adopt a Constitutional amendment that protects the right of the people to keep and bear arms from infringement by any level of government.

Chapter 3

Designing a Database

The discussions so far have covered what a database is, what its component parts are, and how they are related. From these lessons, it is assumed that you would now recognize a database if you saw one. But how do you design one from scratch? It isn't enough that you have seen a lot of databases. You can recognize a car when you see one, but you probably don't know how to design or build one even though you might see hundreds of them every day. Recognizing a database and designing one are two different abilities. Now that you can identify one when you come across it, you need to develop that understanding into an ability to design a working, useful database from the requirements for a particular application.

For years database design has been the coveted domain of a privileged few, those inscrutable gurus who practice the mystic arts of database administration in the

shrines of corporate MIS directorates. Such persons have held sway over the information resources of their benefactors for most of the last thirty years. Why? Because only they knew how to design and maintain a database.

All that elitism is changing. Traditionally, the tools of database administration were available only where they could be afforded. The big computer and the database management software were large, expensive, and difficult to get at. No more. The past ten years have seen a revolution in the availability of computing power. Users no longer rely on the MIS manager and staff of programmers and database wizards to bring them the power of computing. The desktop PC is now commonplace, and people are developing their own software.

This chapter explores the basics of database design, without getting into some of the arcane disciplines that are often advanced as proper design methodology. No myths are exploded, but an idea is put forth that might surprise some data processing seers: anyone can design a workable database by using intuition and common sense and without applying or even understanding the mysteries of database administration.

You won't need special forms, templates, rigid rules, or disciplined procedures for design review and refinement. Those tools and procedures are for the big development project with a big system, a big staff, a big problem to solve, structured management, and the need to avert blame when something goes wrong. Such crutches are not necessary for the successful design of a database, but they are often required by those who find comfort in them.

You do, however, need a clear understanding of the problem you are trying to solve, the solutions you might consider, and the ways to translate the solutions into a database design, and you must be willing to modify the solution as it evolves.

THE NINE STEPS OF DATABASE DESIGN

The design of a relational database involves nine steps taken, for the most part, in succession:

1. Identify the basis for the database requirements.
2. Define the database functional and performance requirements.
3. Identify the data items.
4. Separate the data elements from the files.

5. Build the data element dictionary.
6. Gather data elements into files.
7. Identify the retrieval characteristics of each file.
8. Identify the relationships between files.
9. Develop the schema for the DBMS you are using.

There is always a tenth step, which is to reiterate the first nine. Let the solution to the problem modify the problem, and let each successive solution enhance your understanding of the problem. As it does, retrace your steps through the design process, and change your results. This step is called refinement, and it is often left out.

The nine steps will now be examined in some detail, using the PERSONNEL database described as an example in Chapter 2.

IDENTIFY THE BASIS FOR THE DATABASE REQUIREMENTS

If you are about to design a database, you must have a mission and a purpose. Someone has asked you to automate something. How was that request made? How much was known about the problem when the solution was requested? Is the problem being solved now? Is the present solution automated or not? Has anyone ever solved a similar problem in a different environment? You need to define the functional and performance requirements for the database, and the definition of these requirements should proceed from understanding the functions to be supported. You start by identifying the basis of that understanding.

The basis for a database specifies the problem that is to be solved, the availability of resources that can be used in the solution, and some likely approaches to the solution.

The problem specification can be as simple as a one-line statement of objectives, or it can be a multi-volume report.

Now recall the personnel database from Chapter 2. Its completed design resembles the following:

3 Designing a Database

PERSONNEL:	
EMPLOYEES:	<employee_no>, employee_name, date_hired, salary, *dept_no*
DEPARTMENTS:	<dept_no>, dept_name
PROJECTS:	<proj_no>, proj_name, start_date, completion_date, labor_budget, *dept_no*
ASSIGNMENTS:	<employee_no, proj_no>, hours_charged

A problem specification for this database could be as simple as the one shown here:

Problem: Keep track of the employees within the departments where they work, including their salaries and the date they were hired. Report the status of projects assigned to departments. Record the employees who are working on projects.

The list of available resources includes such information as source documents (employee time cards, for example), interfaces to other databases, personnel (the sales clerk in a point-of-sale application), and so on. Where appropriate, include examples in the list. The resources list for the personnel database could resemble the following:

Company Personnel Action Form
Project Work Order
Employee Time Card

Next, an analysis is needed of the possible approaches to the problem. Be as creative and unconventional as you want, but be careful. Make it clear to everyone, including yourself, that these designs are preliminary. Don't make an emotional investment in a premature design, and don't get caught in a situation where someone can hold you to one particular approach.

Your analysis will be influenced by the environment. Perhaps the user already has a computer, so you need not select one. You might be considering the conversion of an existing database or software system.

To define the basis, start with the present solution. If there is no present solution, start from scratch.

Starting from an Existing Solution

If the user is already using a computer to solve the problem, then you have a good place to start. The existing system will provide insight into the requirements for the data that must be maintained. The user will have experience with an automated solution and will be able to tell you where the new solution can improve on the old. The best way to begin building a new outhouse is to look at an old outhouse. An existing system is an excellent basis for a database design.

If the user works with a manual system, you would start there. You must collect the functions that the system supports and turn them into a set of requirements for a database, which will be discussed in more detail later. A manual system will contain enough information to allow you to design an automated one to replace it. As such, it is a good basis for your design. If the purpose of automation is to improve the system's performance, functionality, or both, then the basis will be extended to include the improvements.

Starting from Scratch

It is hard to imagine designing a database from scratch with no prior procedure for a model. Almost everything has been either automated or set up as a manual procedure. If a user comes to you with the idea of initiating a complete new procedure and solving a brand new problem, you should look around for an existing solution to a similar problem. Chances are someone has already developed the solution you want—or one like it. If you find one, proceed as if you had found it with the new user. If you do not find one, you must start from scratch. The best approach is to develop some manual procedures, use them for a while, and allow them to influence the basis for a new database. If you cannot do that, you must work with the user to develop an initial basis. Be advised that all of the design and implementation that you do will no doubt change as the database takes form and is used. You rarely see the best solution the first time.

Write Down the Basis

Once you have the basis for your database, write it down as you would write a report or a letter. Make it understandable to anyone who understands the problem. Keep your written information around while the system is being developed, and make updates to it as the basis for the system changes. The basis is your foundation for the requirements analysis discussed next.

DEFINE THE DATABASE FUNCTIONAL AND PERFORMANCE REQUIREMENTS

This next step is a refinement of the first. From the basis, you define the requirements for the database. You need to address functional requirements and performance requirements.

Functional requirements specify the kind of data the database will contain. In these requirements, you should document everything you know about the functions that are supported. Be specific in identifying pieces of information the database must know about.

Performance requirements specify frequencies, speeds, quantities, and sizes—how often, how fast, how many, and how big—the database must support.

The statement of a database requirement should be clear and unambiguous. It should address itself to one specific aspect of the functions or performance of the system. Each statement should stand alone, and each statement should completely define the requirement. It should be worded so that users and programmers alike can understand it.

Your list of requirements is the planning document for the database. Anyone who has it and understands it should know what to expect from the system.

Here is an example of a list of requirements that might be developed for the PERSONNEL database from Chapter 2.

Functional Requirements

1. The system will record and report the departments in the organization by department number and name.
2. The system will record and report the organization's employees by employee number. Each employee's record will include the employee's name, salary, and date hired.
3. The system will record and report the organization's projects by project number. Each project's record will include its name, the start and completion dates, and the number of labor hours budgeted to it.

4. The system will record the assignment of employees to departments. An employee is assigned to one department at a time, but a department may have many employees.
5. The system will record the assignment of projects to departments. A project is assigned to one department, and a department may be responsible for many projects.
6. The system will record the assignment of employees to projects. An employee may work on several projects, including projects assigned to departments other than the department to which the employee is assigned. A project may have multiple employees assigned to it.
7. The system will record the hours worked by each employee on each project.

Performance Requirements

1. The system will support up to 10 departments.
2. The system will support up to 100 employees.
3. The system will support up to 20 projects.
4. The system will support up to 200 employee-to-project assignments.
5. Retrieval of data recorded about an employee, project, or department will be on-line and will be in response to the user's entry of the employee, project, or department number. The system will deliver the related data within three seconds of entering the associated control number.

IDENTIFY THE DATA ITEMS

Once you have written down the requirements for the database, you can begin to translate those requirements into identifiable elements of data that are suitable for automation.

You will need a way to note and organize the data items that you identify. Start with a stack of 3x5 cards, an on-line notepad on your computer, or some other technique that allows you to itemize, sort, and shuffle.

To identify a data item, you must rummage around in the work you have already done, looking for potential data items. From your basis and your requirements, extract every reference to anything that looks like a data item and write its name and anything else you know about that item onto one of your 3x5 cards. You can start by pulling the nouns out of your work. Each noun is a potential data item. The example PERSONNEL system basis and requirements list delivers the following collection of nouns:

employees, departments, salary, date hired, projects, schedules, hours budgeted, hours worked, department number, department, name, employee name, project number, project name, start date, completion date, assignments, hours expended.

This list can be used as an initial collection of the items that will be recorded in and reported from the PERSONNEL system database you are designing. Of course, as your work proceeds, you will add to and remove from this list.

SEPARATE THE DATA ELEMENTS FROM THE FILES

Here, you apply your designer judgement, intuition, and guesswork. Look at the data items you have collected. Which of these items seem to be individual data elements, and which seem more like logically organized aggregates of data elements? Shuffle them up, sort them out, and move them around. The 3x5 card method works well here. It should become obvious which items are elements and which are not. A date, for example, is clearly a data element. It is not as obvious that an "assignment" is not a data element. Can you see how a department number is a data element while a department is a data entity, one that might be represented by a number of data elements, possibly including the department number?

There is room for confusion here. Some data elements will break down into component parts, leading you to suspect that they might be aggregates. In a typical example, a date can be divided into its month, day, and year. Is each of those pieces of information a data element? Sometimes they can be. Is the date still a data element? Again, sometimes it can be. Your requirements will dictate the level of detail to which an entity of data is defined. If you need to report the "closing day" or the "day placed in service," then you will need to identify these components of a date as separate data elements.

The objective of this exercise is to separate the data elements from the files so that you can take the next two steps. The nouns that have been culled out as data elements from the list above can be described in this subset of the list:

salary, date hired, hours budgeted, hours worked, department, number, department name, employee name, project number, project name, start date, completion date, hours expended.

BUILD THE DATA ELEMENT DICTIONARY

Recall from Chapter 2 that you define and identify the data element dictionary as a fundamental component in the design of a database. Once you have identified what the data elements are and have a list like the one above, you can build your data element dictionary.

You must collect everything that is known or that can be determined about each data element. At the very least, you will need to know its size and data type. If you are basing the design on an existing system, the documentation—if any—can contribute to your dictionary. If the documentation is inadequate, but the source code is available, you can “reverse-engineer” the data element characteristics by seeing how the existing software uses them. If you are automating a manual system, you can look at the entry forms (time cards, posting ledgers, and so on) to see how the data elements are used. Sometimes these forms have accompanying procedures, and you can use these procedures to find the descriptions of the manual entries, thus learning the requirements for the data elements.

Describing dates, social security numbers, telephone numbers, zip codes, or shoe sizes is easy: these data elements have known limits and formats. But other data elements are not so well defined by common usage. Quantities and amounts need to be described as to their limits—how high can they get? can they be negative? how many decimal places? and so on. Data elements that contain nomenclature are usually called names, addresses, descriptions, remarks, and such. Their lengths must be determined. Account numbers, vendor numbers, client numbers, employee numbers, part numbers, stock numbers, customer numbers, and untold numbers of other numbers are seldom built according to any standards. Many of them are called numbers but consist of letters and punctuation as well as digits.

It is important that you clearly define the physical properties of all the data elements that will be in the database. You must complete as much of this definition as possible before you proceed with the design. Then you must build the data element dictionary.

Find someone in the user community to approve your data element dictionary. If no one will put a signature on your work at this point in the design, then neither you nor they clearly understand the requirements. You need to do more analysis before the design proceeds.

GATHER THE DATA ELEMENTS INTO FILES

When you separate the data elements from the aggregates, the aggregates are left over. You must deal with those aggregates, and chances are they are going to be files.

Following are the data items that did not make it into the data element dictionary:

employees, departments, projects, schedules, assignments

You can look at your requirements now to see which of the above items should be files in your database. None of the requirements calls for the permanent retention of specific schedule records. The information that might be used to report a schedule condition is information that would be recorded about a project (the dates, the budget) or an assignment (the hours worked). So you can drop schedules from the list. Those that remain are the going to be the database files.

It now remains for you to describe the files. Remember from Chapter 2 that a file is a collection of records. To define a file, you must define the format of the record that it stores. To define the record, you specify the data elements that will appear in it.

Use your trusty 3x5 cards for this next exercise, which takes you through the definition of the employees file. Fan out the cards and pull out everything related to an employee. This is not as simple as it sounds. It is obvious that the department name is not related to the employee and that the employee name is. But is the data element that records the hours worked by an employee on a project specifically related to the employee? It is not, at least not immediately. You cannot put that data element into the employee's record because there could be a number of different values for that element for the same employee—the employee can work on multiple projects. The department name is not related to the employee, but is the department number? Yes. An employee is assigned to a department, and only one department,

and the departments are identified by department number. So the department number can be a part of the employee record.

By examining each data element, and viewing it in the light of its relevance to an employee, you can build a stack of 3x5 cards that are the data elements for the employee record. This stack will contain the cards for the employee number, employee name, date hired, salary, and department number. If you refer to the schema used in Chapter 2, you will see that this list corresponds to the EMPLOYEES file description.

You now need to design the rest of the files in the database. Remember that you are never done. Always be willing to change what you have done before. Do not allow your current design task to be constrained by the ones that preceded it. If you cannot get something designed correctly because of an earlier design, then retreat and review that earlier design.

IDENTIFY THE RETRIEVAL CHARACTERISTICS OF EACH FILE

Once you have the files laid out, you need to specify the methods of retrieval that each file requires. You have several alternatives, and a design might consist of any combination of them.

The purpose for identifying the retrieval requirements of the database is so that you can decide which fields should be primary key index values, which fields should be secondary index values, and which files should be related to one another.

Following are the kinds of retrievals that a system could perform.

Specific Key Retrieval

Suppose you work at the computer and call for the display of a selected record. To do this, you need to know something about the data values in the record that is to be retrieved. To retrieve a record from the EMPLOYEES file, a user will know different information depending on who the user is. One user might know the employee number while another knows the employee's name.

A retrieval request is called a "query" and is expressed as search criteria in a query expression. A specific key retrieval will identify the key data element and the data value being searched for. Any record that matches the expressed retrieval criteria will appear in the response. The following is what a key retrieval query expression looks like:

```
dept_no = 123
```

If you use this expression to retrieve records from the EMPLOYEES file, records that have the value 123 in the department number data element will be retrieved.

The query is the expression of the search criteria. The response is the record or records that are retrieved when the retrieval is processed.

Some key retrievals will be based on a partial value of a data element. You might know that the employee's last name is Smith. You might not know the precise spelling of the name. In these cases, the retrieval must be able to deliver a response that contains a list of those records that are in the neighborhood of the specified key. Then you can make a more precise selection.

Some key retrievals can deliver only one record. A retrieval against the EMPLOYEES file that uses a specific employee number as the specified key will deliver only the matching employee record—if one exists that matches the specified key value.

Some key retrievals will deliver multiple records. If, for example, you retrieve employee records that contain a specified department number, as was done in the example above, the response will be a list of those employees—if any—who are assigned to the specified department.

You must understand the nature of the response to a given retrieval because you must know whether you can display the record as a full-screen template that contains all of the data in the record or whether a list of retrieved records is to be displayed.

Some retrievals will tell you how many responses occurred. You can then decide whether or not the retrieval responses should be viewed. If you get 49,000 responses from a file of 50,000 records, you are not likely to want to page through all of them.

Key Range Searches

You might base your retrievals on a range of values for a given data element. For example, you might want to provide a report of all employees who were hired this month or those who are in a particular salary scale. Range searches must always be expected to deliver more than one record for each retrieval.

Boolean Query Retrievals

A Boolean query is one that uses the rules of Boolean logic. A series of true/false conditions are combined in an expression that uses three operators: AND, OR, and NOT. The conditions are expressions that match data element values to key values in the retrieval.

Here are two conditions that are combined with a Boolean operator:

```
dept_no = 123 AND salary < 20,000
```

This expression will retrieve records for employees who are assigned to department 123 and whose salaries are less than \$20,000.

As with programming languages, Boolean expressions can use parentheses to provide the precedence of the search. Consider these two expressions where the data elements and relational operators are the same but where parentheses define the precedence of the expression:

```
dept_no = 123 AND ( salary < 20,000 OR date_hired > 01/01/80)
```

```
(department = 123 AND salary < 20,000) OR date_hired > 01/01/80
```

The first expression would retrieve any employee in department 123 who either makes less than \$20,000 or who was hired after January 1, 1980.

The second expression would retrieve the record for any employee in department 123 whose salary is less than \$20,000 or any employee in any department who was hired after January 1, 1980.

File Scans: Select, Sort, and Display

Many of your retrievals will consist of scanning a file from beginning to end, selecting records that match your search criteria, sorting the selected records into a

new sequence, and preparing a response from the selected records in the new sequence. You can distinguish these retrievals from the on-line retrievals by the frequency of the process (how often is it done?), the urgency of need for the response (how soon is it needed?), and the expected number of records in the response (how big is it?).

If a retrieval is done periodically in a routine fashion against fixed criteria, then the requirement is likely a report rather than an on-line retrieval. The production of a periodic report can be done by the computer when it is not being used for on-line tasks, i.e., when a person is not needed to run it.

It is to your advantage to have as many of the retrieval requirements as possible fulfilled by reports. Their production does not demand your attention beyond starting the program, and they do not require the use of key indexes to provide their responses. Key indexes are required for many of your retrievals, but they exact a cost. Each data element in a file that is a key index is supported by another file to store the index, which implies a storage overhead for each index. Whenever a record is added to, changed in, or deleted from a database file, the supporting index files must be modified accordingly. This requirement implies a processing overhead for each index.

Multiple-File Retrievals

So far, this chapter has discussed retrievals as if they were always performed against individual files. But often you will process retrievals where the responses consist of data elements taken from more than one file in the database.

If you are required to display the employee number, the employee name, and the department number of the department where the employee is assigned, then you can get all that information from the EMPLOYEES file in the PERSONNEL database. But if the requirement is to display the department name as well, then you must build your response from data elements taken from two files because the department name is not in the EMPLOYEES file. To support this search, you must first locate the employee record, which will be based on the search criteria specified in the query expression. Since each employee record has the department number in it, you will use that department number to retrieve the matching department record in the DEPARTMENTS file. The department name from the department record goes into your response.

Using a data element from a record in one file to find a record in another file is the basis for the relational data model. With this technique, you can describe retrieval paths that navigate the many files in the database, gathering data elements to form a response to a retrieval.

A retrieval that uses data elements from more than one file implies that a relationship exists among the files. This relationship leads into the next discussion of interfile relationships.

IDENTIFY THE RELATIONSHIPS BETWEEN FILES

In a relational database, files are related when they contain a common data element which is the primary key for one of the files. In the PERSONNEL database, the EMPLOYEES file is related to the DEPARTMENTS file because the employee record includes the department number data element, and the department number is the primary key to the DEPARTMENTS file.

Because of this relationship, you can view the DEPARTMENTS file from two perspectives based upon the two purposes it serves. First, it is the system's record of everything related to a department. All by itself, it could be a DEPARTMENTS database rather than a file in a larger database. It needs nothing from the EMPLOYEES file to fulfill its purpose. Second, the DEPARTMENTS file is the table of information about the department in which an employee works. It is used to validate the department number when one is being stored in an employee's record and is also used to provide the name of and other information about the department for an employee.

This relationship implies some responsibility. For it to be effective, it must have integrity. If an employee record contains a department number, there should be a matching department record. If no such department record exists, the integrity of the database has been compromised. This condition is caused by one of two circumstances: (1) a program has added a department number to an employee record without first checking the department file; (2) a program has deleted a department record without checking for the existence of its department number in the employee records.

Your applications software is responsible for ensuring the integrity of these relationships. Whenever you add a department number to an employee record, you can retrieve the department record. If it isn't there, the department number you are adding is invalid. If you make the department number a secondary key into the EMPLOYEE file, then you can enforce the relationship in the other direction. Whenever you want to delete a department record, you can do a retrieval to see if any employee records have the department number. If so, the delete should not occur because the integrity of the relationship would be compromised.

There can be other relationships not shown by the appearance of key data elements in files. These relationships are defined by your retrieval requirements. You might not have realized that you needed to record a key data element in a file when you designed that file. Look at the output requirements to see if you missed any relationships, and change your file design where appropriate.

You must evaluate each such potential relationship to see if it is real. In some cases, a controlling number is recorded in a record as information only. Users do not care if its corresponding record is still in the other file; it may have been retired.

More often, the relationships are real and must be protected. In one-to-one and one-to-many relationships, the software must preserve data integrity. In the case of the many-to-many relationship, you must provide a connector file to support the relationship, and then you must ensure that it is always synchronized with the two files it connects.

The relationships between files can describe potential retrieval paths for multiple-file retrievals. Sometimes, the apparent paths are incorrect. Figure 3-1 shows the PERSONNEL database and the relationships between the three files. If you designed a retrieval that began by retrieving a department record, each project for the department, and then each employee who is assigned to each project, you might not have the answer you are looking for. If the purpose of the retrieval is to list the employees who are working on projects that are assigned to the department, then the response is correct. But if the retrieval is supposed to deliver a list of employees who work in the department, then the response is incorrect. In this case, you have chosen a retrieval path based on interfile dependencies that don't work. Remember from the requirements that an employee in one department can work on a project that is assigned to a different department. The database design supports these relationships, but, as seen in this example, the potential exists for you to describe paths that deliver incorrect results. This is not a flaw in the database design. It is correct. The problem lies in the description of a retrieval path. You must develop these paths carefully to ensure that the answer is going to be the one you are looking for.

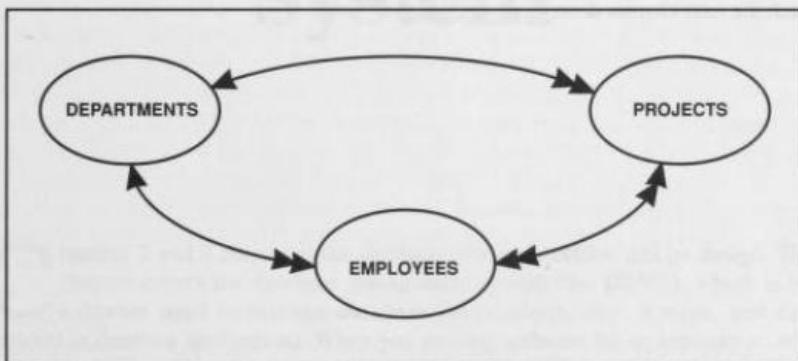


Figure 3-1. Retrieval Paths.

DEVELOP THE SCHEMA FOR THE DBMS YOU ARE USING

This last step requires knowledge of the schema language of the database management system that you will be using. You must translate the pile of 3x5 cards and diagrams into a format that the DBMS can understand. The next chapter provides more detail on the nature of the DBMS and its schema language.

SUMMARY

Now you have an idea of what constitutes a database and how to design one. So where can you go from here? A database design is of little use without a software system to process it. Database software comes in two parts—your applications programs that process the data records in their functional formats and the general-purpose system of software that manages database definition, organization, storage, and retrieval. This software is called the database management system, or DBMS, which the next chapter describes.

Chapter 4

The Database Management System

Chapters 2 and 3 discussed the database—its architecture, and its design. This chapter covers the database management system (the DBMS), which is the software used to manage database descriptions, data storage, and data retrieval in database applications. When you develop software for an application, why develop code to read and write records, maintain index files, and store and retrieve data? These processes have common requirements in most systems. You shouldn't need to rewrite them every time you build a new application.

THE DBMS

Software packages exist that manage routine storage and retrieval of data in databases. Such software is called the database management system or DBMS. The

DBMS sits between the application programs and the database and stores and retrieves data records. It runs with and responds to requests from the application software. Figure 4-1 shows the relationship between the application programs, the DBMS, and the database.

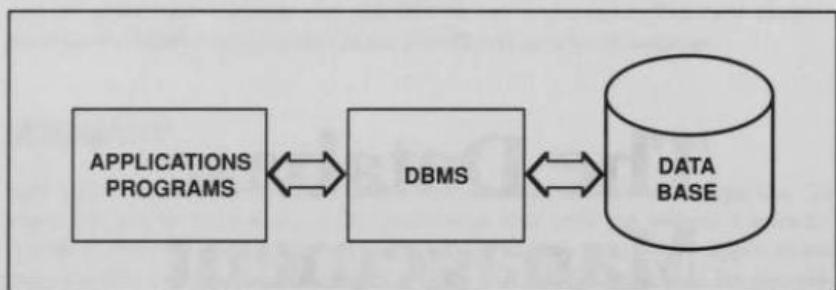


Figure 4-1. Managing the Database.

When an application program needs a record from the database, it calls the DBMS and passes it parameters that tell the DBMS how to retrieve the record. This call can be a request such as "find the inventory record with stock number 0123." The DBMS finds the record and passes a copy of it back to the application program. When the application program has a record to write to the database, it calls the DBMS and passes a copy of the record. The DBMS places the record in an appropriate location where applications programs can later find it. The DBMS manages the indexing of the record and the relationships it has with other records in other files in the database. The applications programs need to know only that they have originated or retrieved a record, changed it, and returned it to the database.

COMMERCIAL DBMS PACKAGES

To use a database, you must have access to one, usually through a DBMS. If you work with a mainframe or minicomputer, you might develop software with a large commercial DBMS. If you develop software for the IBM PC, you might use one of the DBMS packages available in that environment. Most packages will support a large, complex database. Some spreadsheet programs offer a database manager function. Spreadsheet databases are usually limited to fit within the capacity of the

spreadsheet as determined by the software and the memory available on your machine.

You can apply the ideas discussed in Chapter 2 to commercially available database packages. These packages have their advantages and weaknesses. If you use any of these packages to develop medium and large database systems, you will find they have mixed blessings and curses. They all share two disadvantages—the cost per copy for use is high, and the tendency for their developers to release improved versions that make older versions obsolete is frequent. Cost can be important. When you develop software to sell, you want to be able to install it in as many different sites as possible. It must be attractively and competitively priced. You might even want to give the software away as an incentive or a promotion. Suppose you develop a system to support a large potential customer base. To sell the system, you must provide the supporting software required to run it, which includes the DBMS. If it is necessary to add several hundred dollars to the sale price to cover the cost of a DBMS license, you might not be competitive, or the price might not be as attractive as it could be.

Consider another problem with packages. A recent project involved the development of a software system that was installed in several locations. The system included a word processor, a DBMS, and custom software for the IBM PC. Development of the custom software was under way for about a year. On the basis of some benchmarks, the word processor and DBMS packages were selected because of their features and their performance. Then, after project was in process and committed to these packages, the vendors decided to improve them. The improvements were a disaster for the project; the custom software no longer worked with the text file produced by the word processor, and the DBMS itself no longer worked the way it formerly did.

You can't fault the word processing people for making their product better; they want to maintain their competitive position; however, you can fault the DBMS vendor for releasing a new, improved, bug-ridden product. Both new packages (the one that worked and the one that did not) took their toll on the project. Because the project team was not in control of the development of the packages, it could not influence the impact of those packages on the requirements of the project. The obvious solution was to retreat to the earlier versions, but earlier versions were no longer supported and licenses were not available to use them.

DBMS packages can restrict use with limits and boundaries. A database can have no more than a specified number of files; a file can have only so many records; a record

can have a limited number of data elements and only so many characters; the number of key data elements might be limited. The list of limitations has no limit. One nationally advertised DBMS reduced the number of data elements allowed in a file in a new and improved version; however, the company forgot to improve their advertising and documentation to communicate this improvement.

Such a system would encounter harsh realities if it tried to automate something for a large bureaucracy where fiefdoms survive and are protected. There is a control number for everything, a code for everything, an acronym for everything, an organization in charge of everything, and a date when everything happened. Users won't part with any of the data elements and want everything in one file so they can write queries without understanding database architecture. It is nearly impossible to fit all the information into a small number of data elements.

You should understand why DBMS packages have limits. The software needs maximum values for its array dimensions and its buffer sizes, but you, the programmer, are stuck with whatever the DBMS developer decides are reasonable limits. You cannot increase the limits, even if you have a lot of memory in your computer, and you cannot trade one limit for another to suit a particular application problem because you do not have the source code to the DBMS.

Many DBMS packages have yet another disadvantage. The formats of data files and indexes are hidden from the software developer. If you need to develop a utility function for data management, a lack of knowledge of the database internal structures can hamper your efforts. The same circumstance prevents you from integrating the database with other software systems that do not use the DBMS. A system that does not reveal its internals to its users has a closed architecture. In almost every case, a closed architecture is an advantage only to those who sell the system. It is a hindrance to those who try to use it.

AN ALTERNATIVE TO THE DBMS PACKAGE

There are alternatives to using a packaged DBMS. The C programmer will find few DBMS products with functions that a C program can call. Until recently, most IBM PC DBMS packages provided their own application language and did not support calls from other programming languages. To accommodate today's trends, some of the vendors offer a C developer's toolkit; however, most such kits produce code that fares poorly when compared with the efficiency of the typical C program. This book

presents alternatives to using a packaged DBMS involving a library of C functions that do for you what a DBMS would do. The advantages to this approach are the efficiency of the resulting system, the absence of licensing costs, the control you have over the DBMS and its destiny, and the open architecture of the database files and indexes.

DATABASE MANAGEMENT SYSTEM FUNDAMENTALS

If you are to build your own DBMS, you should understand the fundamentals of DBMS technology. These fundamentals will define the basis for the software system that you build in the following chapters.

A DBMS is used to manage a database. You express the database design in a format that the DBMS can interpret. On one side are applications programs wanting to get at the data records; on the other side is the database waiting to be processed. In the middle is the DBMS, which passes data back and forth between the programs and the database. For the DBMS to pass this information, something must tell it what the data records in the database look like. Statements called the Data Definition Language provide a description of the database in a format that the DBMS can interpret. Once the DBMS knows the format and organization of the data, it must have a way to communicate with the application programs. An inter-program protocol called the Data Manipulation Language provides this communication. To build a DBMS, you must consider how these two languages will look.

The Data Definition Language

In Chapter 2, a small personnel database was designed by describing the data elements, the files, and the relationships between the files. The design took the form of tables that listed the characteristics of each data element and each file. Those tables are a statement of the format of the database, and, as such, constitute a Data Definition Language; however, to be useful in a DBMS, the tables must be in a format that the computer can translate into physical storage characteristics for the data. After you design the database, write the DDL. It is a language just as C is a language. And just as there are many programming languages, there are many DDLs. Which DDL you use depends on which DBMS you use. So, even if you design the database with tables such as those shown in Chapter 2, you must still express that

design in a DDL suitable to your DBMS. Figure 4-2 shows the database design reduced to a DDL that the DBMS can read.

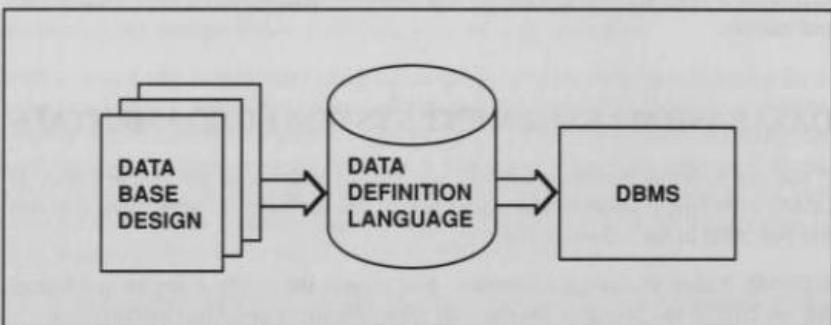


Figure 4-2. The Data Definition Language.

Various techniques exist for writing the DDL. One widely used technique is to write the DDL statements into a text file much as you would build the source file for a program. Some DBMS packages use a DDL compiler program to process the DDL file into tables the DBMS can understand. Others have the DBMS interpret the DDL in its original code. In this book, C language statements will be used to express the DDL. Chapter 6 discusses the characteristics of the C language that make this expression possible. Later, you will use a DDL compiler program that translates a text DDL into the C language DDL.

Data Models

When designing a DDL, you must be aware of the data model that the DBMS supports. Database terminology defines the data model as one of three traditional forms for the organization of data: the hierarchical model, the network model, and the relational model.

A hierarchical database is one where the relationships between the files form a hierarchy. In a hierarchy, a parent file may have several child files, but each child file may have only one parent file. In a hierarchy, files rank from top to bottom, with higher level files being the parents of lower files.

Figure 4-3 shows a hierarchy of a department's employees and projects.

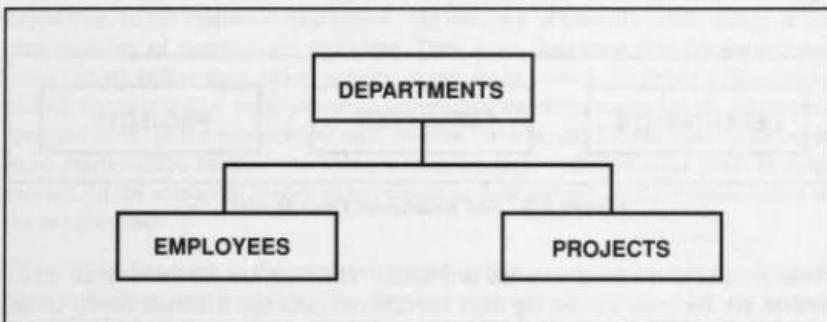


Figure 4-3. The Hierarchical Data Model.

A network database is similar to a hierarchical database except that a file can have multiple parents. In Figure 4-4, the organization of the three files shows that the project file is also a parent of the employee file.

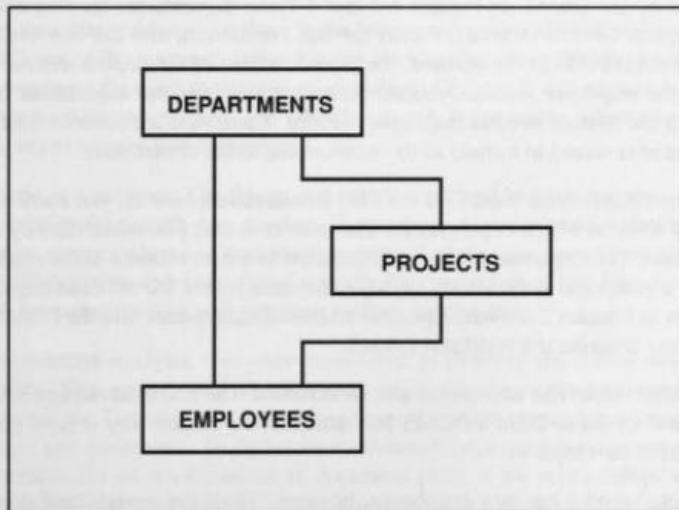


Figure 4-4. The Network Data Model.

In the relational database, files have no parents and no children. They are unrelated. Figure 4-5 shows the three files in a relational database.

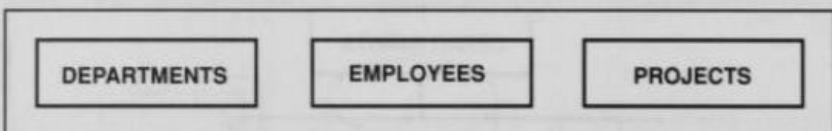


Figure 4-5. The Relational Data Model.

These models don't seem to make any sense. The files are the same in all three models, but the lines connecting them are different, and the relational model is the one where the files are unrelated.

The lines between the files define the physical relationships that the DBMS establishes and maintains. The lines represent internal pointers that are recorded in the records and that point to the records hosted by the parents. Databases supported by hierarchical and network DBMSs contain control values inside the records to establish the relationships. These values are pointers unseen by the user and maintained by the DBMS. In Figures 4-3 and 4-4, the departments are connected to the employees. Several techniques exist for this connection, and the one used will depend on the DBMS. In one method, the record address of the department record is stored in the employee record. Another technique involves the department record pointing to the first of several employee records. The employee records then each point to the next record in a chain of those belonging to the department.

What about the relational model? If the files are unrelated, how do you know which employees work in which departments? The answer is that you relate files by using data elements. The department number is included as a data element in the employee file. It is a relational data model because the data forms the relationships. The discussions in Chapter 2 are built upon the relational data model, and the DBMS that later chapters describe is a relational DBMS.

The relational model has advantages and weaknesses. The major advantage is that it is the easiest model to build a DBMS for, which is one reason why it is so popular among DBMS developers.

The relational model has two drawbacks, however. First, the model itself does not enforce relationships. In a situation such as the one in Figure 4-5, you can have an

employee who has no department. Worse, you can assign an employee to a department that does not exist. The model does not prevent this situation from happening. In the relational data model, the integrity of interfile relationships is the responsibility of applications programs. They must determine that the department record exists before they add or change an employee record. To delete a department record, the application programs must ensure that the department has no employees assigned to it. In the hierarchical and network models, the DBMS usually enforces these relationships because the pointers must be there and they must point to valid records. In the relational model, enforcement of data integrity is the responsibility of the programmer.

The second drawback to the relational model occurs when two files in the database have a many-to-many relationship. Suppose that employees can work on several projects, and each project can use several employees, as in Figure 2-6. For a hierarchical or network database, the DBMS maintains a pointer mechanism invisible to the user and programmer. For the relational model, the database designer designs a connector file into the database. In Chapter 2, the ASSIGNMENTS file was an example connector file. Such a file often exists only to support the relationship. It is a contrived structure because of the nature of the model, and, usually, the programmer and the user must be aware of it.

Nonetheless, the relational database is the backbone of most DBMS packages for the IBM PC and is the approach taken for this book's alternative DBMS; however, you must recognize the need to enforce relationships. Chapter 6 will explain how if you design your database properly, the DBMS will not, for example, allow you to assign employees to nonexistent departments.

The world is a network. The things and people in it tend to have complex, many-to-many relationships with one another. If you have ever designed a database for a system to support what was previously a manual application, you know that people will build complex and convoluted systems for themselves. They find comfort and security in understanding and operating esoteric mechanisms.

A requirements analysis was once conducted to develop the initial design for a database. The automated system was to track the status of engineering documentation. The user had an existing manual system consisting of many forms, drawings, and documents. To derive their relationships, a questionnaire was designed to determine, for all combinations of document pairs, if the relationships were one-to-one, one-to-many, or many-to-many. The questions were in the form of true-false

statements such as "There may be many Engineering Orders for a given Drawing; True or False." An analyst interviewed the proprietors of these documents to determine the relationships. Amazingly, every relationship in every pair from a dozen documents was many-to-many.

The reduction of such a mess into a relational data model involves a multi-step process called normalization. Several books explain this procedure in detail. Two excellent examples are listed here:

James Martin, *Computer Data-Base Organization*, Second Edition,
Prentice-Hall, Inc., 1977.

C.J. Date, *An Introduction to Database Systems*, Second Edition,
Addison-Wesley Publishing Company, 1977.

It is not within the scope of this book to tackle the weighty subject of normalization. You should read either or both of the two references just cited; they cover the subject of database technology and are valuable to anyone involved in database design and programming.

Incidentally, computer science writers do not agree on how the term "database" should be written. Martin hyphenates the term ("data-base"). Date makes it one word ("database"). Others use the two words ("data base").

The Data Manipulation Language

Given a Data Definition Language (DDL) to describe the database to the DBMS, you need a corresponding language for programs to use so they can communicate with the DBMS. Such a language is called the Data Manipulation Language (DML), and it and part of the DDL are used whenever an application program wants access to the data. The DDL describes the records to the application program, and the DML provides an interface to the DBMS. The DDL uses record formats, and the DML uses external function calls, both in the fashion of the host programming language. The record formats are a part of the DDL that translates data definition into data manipulation. To retrieve and update records, you must know their formats. The calls to functions provide ways to store and retrieve data to and from the database. Figure 4-6 shows how the DDL relates the DBMS and the application programs and how the DDL and the DML provide communication between the DBMS and the programs.

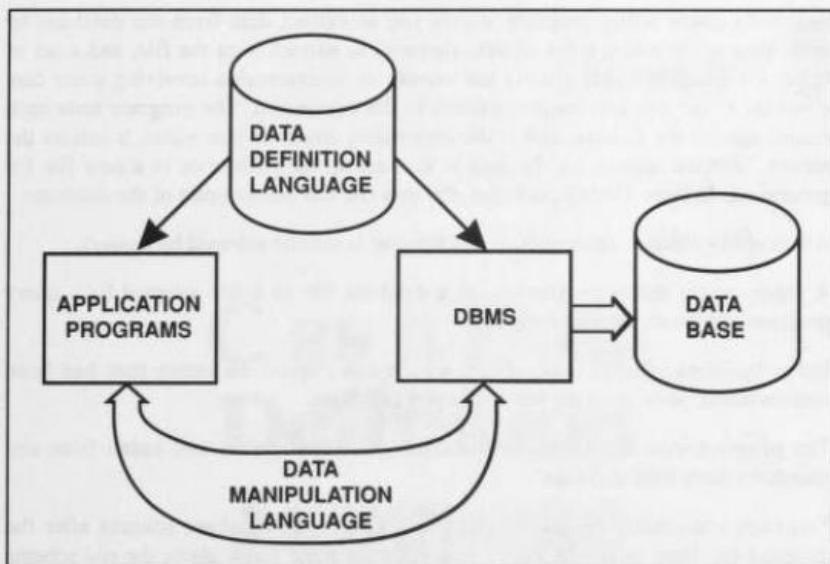


Figure 4-6. The Data Manipulation Language.

The DML must provide functional access to the database. This access includes functions that retrieve records from files by key search, add records to files, delete records from files, retrieve records sequentially in a key sequence, retrieve records in the physical sequence in which they are recorded, and rewrite records that have been updated.

DBMS Utility Programs

Most DBMS software packages will include more than a DDL and a DML. To support the management of data in a database and to allow infrequent ad hoc retrievals of the data, the DBMS usually has a library of utility programs including a data entry program, a query package, a sort, and a report writer. Other useful utilities for tending the database are index-building tools, a program to initialize the database files with null data, and programs to convert files when you modify the schema.

A general-purpose data entry program allows you to specify a file and a subset of its data elements and then to add, change, or delete records in the file using a screen displaying the data elements.

A DBMS query utility program allows you to extract data from the database by specifying a file name, a list of data elements to extract from the file, and a set of select criteria. The select criteria are usually in an expression involving some data elements in the file and some constants in the expression. The program tests each record against the criteria, and if the expression returns a true value, it selects the record. Selected records can be sent to the screen, the printer, or to a new file for processing. In some DBMS packages, the new file can become part of the database.

A sort utility changes the sequence of a file that is usually selected by a query.

A report writer utility program reads a database file or a file selected by a query program and builds a report from it.

Index builders rebuild index files, which can correct an index that has been contaminated, perhaps as the result of a power failure.

The program that initializes the database creates the data and index files and initializes them with null data.

You need conversion programs when you modify the database schema after the database has been in use. A conversion program must know about the old schema and the new one. It reads file records using the old schema and writes them using the new schema. New data elements are set to null values.

A DBMS will include most of these utility functions in one form or another. The extent and complexity of the support provided by a particular utility program depends on the package itself.

SUMMARY

Chapters 2 through 4 have established the groundwork for the approach to an automated database. Chapter 5 begins to reveal the technique for having database management software without having a commercial database management system. You can build your schema in source code such that applications programs are unaware of the properties of the data and its indexes, and the DBMS is unaware of the specifics of the application that the DBMS supports. This code requires a programming language with certain language constructs that support these goals. C is such a language, as you will see in Chapter 5.

Chapter 5

C as a Data Definition Language

Chapters 2 and 3 discussed databases and design; Chapter 4 discussed the database management system (DBMS). This chapter introduces C as a Data Definition Language. Following this chapter, Chapter 6 presents a DBMS that is available to the C programmer. The technique involved uses features of the Standard C language to emulate the Data Definition and Data Manipulation Languages of larger, more complex systems. These languages include three components of the automated database environment:

- the schema
- the applications software
- the DBMS

First, you describe the schema in a source code module that you include in the applications source program and compile into a relocatable object module. Then you link this module with the DBMS. You compile the DBMS separately with external references to a generic schema. When you link the applications program to the DBMS, the references to a nonspecific schema in the DBMS are resolved with the schema source program. With this approach, the schema and the DBMS become a part of the applications program in one executable object program. Figure 5-1 illustrates this technique.

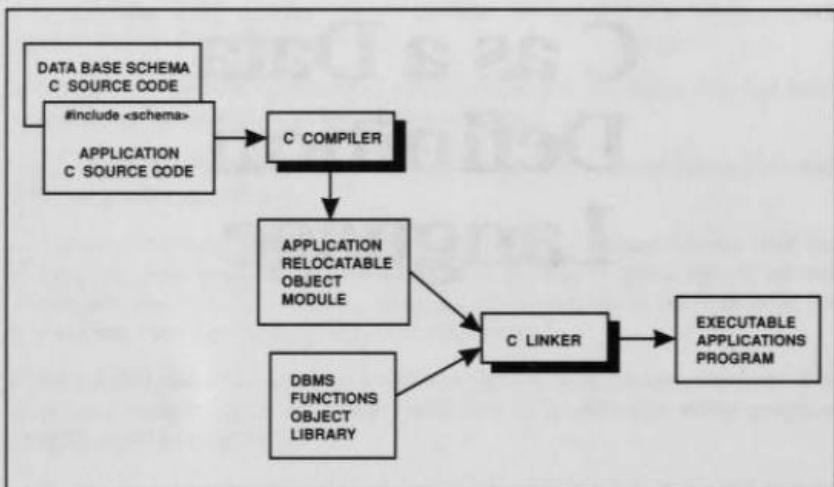


Figure 5-1. Building a C data Application.

Each program in this approach is self-contained; together, they include everything needed to manage the database. The schema is a set of external arrays; the DBMS is a library of reusable C functions, and each application's function individually supports its specific purpose. You link all three together into a single executable code module, as shown in Figure 5-1. To support this technique, the programming language requires certain features. You will find these features in C.

The programming language must allow the assignment of global alphanumeric names to integral values. To describe a database schema that adjusts itself to

modifications in the data formats, you must be able to give names to files and data elements. As the number of names grows or diminishes, the software should adjust without attention. The files will be called by their names, and the software will use the names to offset into tables of data elements. The data elements will also be called by name, and the software will use those names to offset into tables of types and lengths. Remember from Chapters 2 and 3 that you describe a database by a file list, and you describe a file by a data element list. You want these to be lists of integers. They could be ASCII string lists, but that would involve the memory overhead of string storage and the execution overhead of string searches. In almost every case, the DBMS must reduce the names to integers for subscripts into the schema tables. Integer lists are used here because they are easier and faster to process.

C includes the enumerated data type. You can use it to equate the file names and data element names to integers. Consider this list of file names:

```
enum files {
    PROJECTS,
    EMPLOYEES,
    DEPARTMENTS
};
```

Consider this list of data elements:

```
enum elements {
    EMPLOYEE_NO,
    EMPLOYEE_NAME,
    DATE_HIRED
};
```

By using the enumerated data type, you assign integers to the names of the files and data elements. Now you can build integer arrays that contain instances of these names in a way that is readable to both programmer and machine. The source language has data elements and file names that have meaning, and the software has integers to process. You can seem to pass file names and data element names to functions that are expecting integral values. The library functions that store and retrieve database records can use these values as subscripts into the tables of the schema.

The next requirement is for the programming language to support multiple source files that are independently compiled. Some dialects of Pascal and BASIC are deficient in this respect. The DBMS is code in a source file, and you do not want to

compile it every time you compile an applications program. The DBMS should be available as a relocatable object module. You should not have to recompile it every time you change the application. The C language supports this requirement for independently compiled source modules.

The language must also support variable identifiers that are significant to enough positions to allow their use as file and data element names. The significance should provide for names long enough to be titles on reports and queries because that is how you will use them. Standard C defines the significance of identifiers to be 31 positions for names within a source file, which works in this case. You do not need long names across source files because each of your applications source files will use the enumerated data type declarations, and the library functions deal only with the integer equivalents of the file and data element names.

The language must also support external variable types and should not be a "strongly typed" language. To understand this concept, consider what "strongly typed" means.

A strongly typed language is one that enforces the types and dimensions of variables. If a function expects a certain type of variable as a parameter (for example, an integer), then any violation of this type by a calling function is an error when the function is compiled. Further, if an array is of a certain size (dimension), then any attempt to subscript an element outside of the array's bounds is an error.

C allows a function to declare an external array with specified dimensions. A separately compiled function can operate on that array without knowing its dimensions. Here is an example:

```
/* source file A */
int i [] = {1,2,3,4,0};
main()
{
    foo();
}
/* source file B */
extern int i[];
foo()
{
    int *j = i;
    while (*j)
        printf("\n%d", *j++);
}
```

Integer array **i** in source file A contains five elements. In source file B, it is known only as an array of integers. The assumption is that the code that uses the integer array will somehow determine its length. In the example, the terminating zero value defines the end of the table. The critical feature is the ability of the function named **foo** to subscript beyond the known boundaries of an array. If you change source file A so that the array is a different length, you do not need to recompile source file B or attend to it in any way. All you must do is relink B's object module with A after A has been recompiled.

In a variation on this feature, the array is a local variable inside the **main** function. The **main** function passes the address of the array to the **foo** function, which accepts it as an integer pointer rather than as an array of integers. Once again, the **foo** function is unaware of the dimensions of the array. It only cares that the integer list is terminated with a zero value integer. Here is an example:

```
main()
{
    static int i [] = {1,2,3,4,0};
    foo(i);
}

foo(int *j)
{
    while (*j)
        printf("\n%d", *j++);
}
```

These advantages of the C language are significant in a systems programming environment. Their implications are extensive, and you rely on them to support a data definition language. Since you will define a file name as an enumerated integer and records as lists of data element enumerated integers, you can allow the following.

The DBMS functions that manipulate records can, given a file name integer, subscript to an array of data element integers without knowing the length of the array, which means you can compile the DBMS functions once, and then link them with many different applications and many different schemas for many different databases without having to modify the DBMS code.

There is one final requirement for a programming language that will support this approach to a database schema. The language must have a complete set of data pointer operations. This area is one in which C shines. C contains facilities for pointers to arrays and structures, pointers to pointers, pointers to arrays of pointers, and so on, as well as a close association between pointers and arrays. All of these pointer operations are critical to this approach to a data definition language. These features are also the most difficult aspects of the C language to master. Probably no one has become totally comfortable with some of the ways you can write code for pointer and array operations in C. After several years of C programming, a programmer still needs to shift some mental gears in order to understand what happens to a pointer to a multiply-dimensioned array of pointers to an array of structure member arrays of function pointers. It is fortunate that this situation doesn't happen often.

SUMMARY

This chapter began to reveal the technique for having database management software without having a commercial database system. Chapter 6 presents a DBMS that is available to the C programmer.

Chapter 6

Cdata: The Cheap Database Management System

As explained in Chapter 5, you can use the facilities of the C language to describe the schema of an integrated database in the relational data model. This book applies that technique. The C functions that support it are called Cdata, the Cheap Database Management System. Cdata is not a product name; it is an abbreviation used in this book to identify the technique.

THE CONSULTANT'S BILLING SYSTEM

To describe the approach, the book uses the Consultant's Billing System (CBS) as an example application. This system is a no-frills application that exists primarily to illustrate the database software; however, it is functional, and a consulting firm

could use it. This chapter and the next use CBS to describe Cdata and its utility programs from an applications perspective. Chapter 8 adds the programs that you would develop to support a specific application (in this case, CBS).

CBS has a database that records time and expense charges against projects for clients. It prepares invoices for labor hours and expenses, and it computes labor charges from the hourly rates of consultants assigned to projects. You can post expenses directly.

The above paragraphs describe the problem definition for the system—the initial stage in database design, as discussed in Chapter 3.

This new database is not the PERSONNEL database used in the examples in the previous chapters. Although the two are similar, there are several differences. The initial design processes described in Chapter 3 will not be restated here. Instead, assume that the nine steps of database design have been completed and that you have a workable design for the CBS database. The database will have four files: a CONSULTANTS file, a CLIENTS file, a PROJECTS file, and an ASSIGNMENTS file. The rest of the design is revealed as it is translated into the Cdata schema.

In Chapter 8, you will build a software system around the CBS database to show how you integrate applications code with the Cdata Data Manipulation Language.

THE CDATA DATA DEFINITION LANGUAGE

Cdata uses a DDL similar to the approach discussed in earlier chapters. In its first incarnation, the Cdata DDL used C language statements that the programmer wrote and compiled along with the applications programs. This was a workable approach, but maintenance of the DDL was difficult with a complex database. For that reason, the Cdata schema compiler program was developed to make the task simpler. Before discussing the compiler, this section will describe the DDL C language statements that you compile along with your application. Then you will see how the compiler can serve as the shortcut for preparing and maintaining the DDL.

The Cdata Data Element Dictionary

One step in the design of the CBS database is the development of a data element dictionary. By using the approach from Chapters 2 and 3, you can build a table of data elements for CBS, as shown in Table 6-1:

Table 6-1. CBS Data Element List.

<u>Data Element</u>	<u>NameData</u>	<u>TypeLength</u>
client_no	numeric	5
client_name	alphanumeric	25
address	alphanumeric	25
city	alphanumeric	25
state	alphanumeric	2
zip	numeric	5
phone	numeric	10
amt_due	currency	8
project_no	numeric	5
project_name	alphanumeric	25
amt_expended	currency	9
consultant_no	numeric	5
consultant_name	alphanumeric	25
rate	currency	5
payment	currency	9
expense	currency	9
hours	numeric	2
date_paid	date	6

Next, you describe these data elements in a format that the applications programs, the DBMS, and the utility programs can use. You express the characteristics of the dictionary in the syntax of the C language.

The first part of the dictionary is found in the definitions of an enumerated data type where the identifiers serve as mnemonic names of the data elements, and their values equate them to integers. Applications programs will use these identifiers to reference the data elements in tables and function parameters. The Cdata programs use the identifiers to assign data elements to files and indexes. Listing 6.1 is the first component of the Cdata data element dictionary and schema for the CBS.

Listing 6.1

```
1| /*  
2|  * Data Element Dictionary  
3|  */  
4|  
5| typedef enum elements {  
6|     CLIENT_NO=1,  
7|     CLIENT_NAME,  
8|     ADDRESS,  
9|     CITY,  
10|    STATE,  
11|    ZIP,  
12|    PHONE,  
13|    AMT_DUE,  
14|    PROJECT_NO,  
15|    PROJECT_NAME,  
16|    AMT_EXPENDED,  
17|    CONSULTANT_NO,  
18|    CONSULTANT_NAME,  
19|    RATE,  
20|    PAYMENT,  
21|    EXPENSE,  
22|    HOURS,  
23|    DATE_PAID  
24| } ELEMENT;
```

Observe in Listing 6.1 that each data element name from the initial paper design is now a globally defined identifier available to any program that includes the ELEMENT enumerated data type when it is compiled.

The Cdata functions need to know the lengths of each of the data elements. An array of integers contains these lengths. Each integer is the length of a data element, and the integer's position in the array tells which data element it belongs to. The first length is for the first data element in Listing 6.1; the second is for the second, and so on. Listing 6.2 shows the array that contains the data element lengths. Each

applications program includes it as an external array so that the Cdata functions can derive the data element lengths.

Listing 6.2

```
1| /*  
2| * Data Element Lengths  
3| */  
4|  
5| const int ellen [] = {  
6|     5,25,25,25,2,5,10,8,5,25,9,5,25,5,9,9,2,6  
7| };
```

Some utility programs and functions use string versions of the data element names. The utilities use the strings to display the data element names in queries and reports. They also use the strings to convert data element names entered by the user into their corresponding enumerated integer values so that the Cdata functions can use them. Listing 6.3 is the array of character pointers that supply those strings. You compile it with the applications programs and utilities. To work properly with the utility programs that are described in the next chapter, the data element string names must be in uppercase.

Listing 6.3

```

1| /*
2|  * Data Element Name Strings
3|  */
4|
5| const char *denames [] = {
6|     "CLIENT_NO",
7|     "CLIENT_NAME",
8|     "ADDRESS",
9|     "CITY",
10|    "STATE",
11|    "ZIP",
12|    "PHONE",
13|    "AMT_DUE",
14|    "PROJECT_NO",
15|    "PROJECT_NAME",
16|    "AMT_EXPENDED",
17|    "CONSULTANT_NO",
18|    "CONSULTANT_NAME",
19|    "RATE",
20|    "PAYMENT",
21|    "EXPENSE",
22|    "HOURS",
23|    "DATE_PAID",
24|    NULL
25| };

```

The same utility functions that display data element names need the attributes of the data element. These attributes describe the format and content of each data element. By using these attributes, the software knows how to control the data entry and display of the data elements. The utility functions use screen management functions that describe data elements in terms of a display mask and a data element type code. The display mask uses the underscore character to define character positions and allows punctuation characters to appear in the mask. The data element type code is as follows:

- A = alphanumeric
- C = currency
- Z = numeric, zero-filled
- N = numeric, space-filled
- D = date

Listing 6.4 shows the character array of data element type codes and the array of display mask character pointers that correspond with the CBS data element dictionary. You compile these arrays with the applications programs and utilities as external arrays.

Listing 6.4

```

1  /*
2   * Data Element Display Characteristics
3   */
4  const char eltype [] = "ZAAAAANNNCACZACCCND";
5  const char *elmask [] = {
6      "_____",
7      "_____",
8      "_____",
9      "_____",
10     "___",
11     "____",
12     "(____)-____",
13     "$_____.____",
14     "____",
15     "_____",
16     "$_____.____",
17     "____",
18     "_____",
19     "$_____.____",
20     "$_____.____",
21     "$_____.____",
22     "___",
23     "___/___/___"
24 };

```

File Specifications

Next, you need to decide what data files will be in the CBS database. Since the database must maintain a record of clients, projects, and consultants, it follows that you will build a data file for each. Figure 6-1 shows the three files and their relationships. The symbolic approach discussed in Chapter 2 is used to illustrate the files.

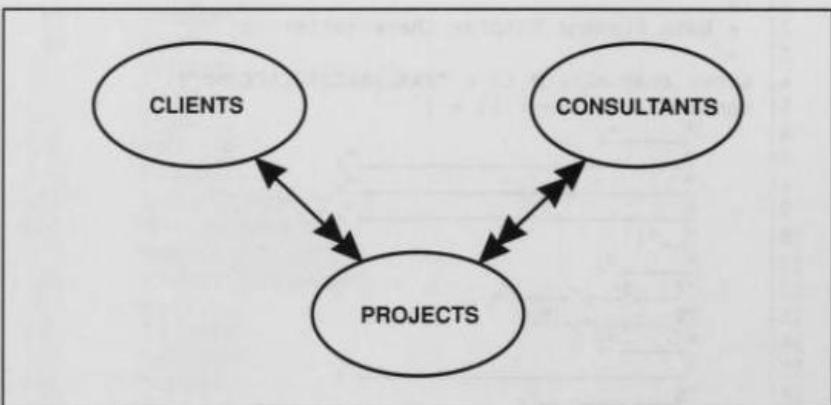


Figure 6-1. The Consultant's Billing System Data Base.

A many-to-many relationship exists between consultants and projects, so you must add a connector file to maintain the relationship. Since the connector file stores data

about consultants assigned to projects, it is called the ASSIGNMENTS file. Figure 6-2 shows the database with this file added.

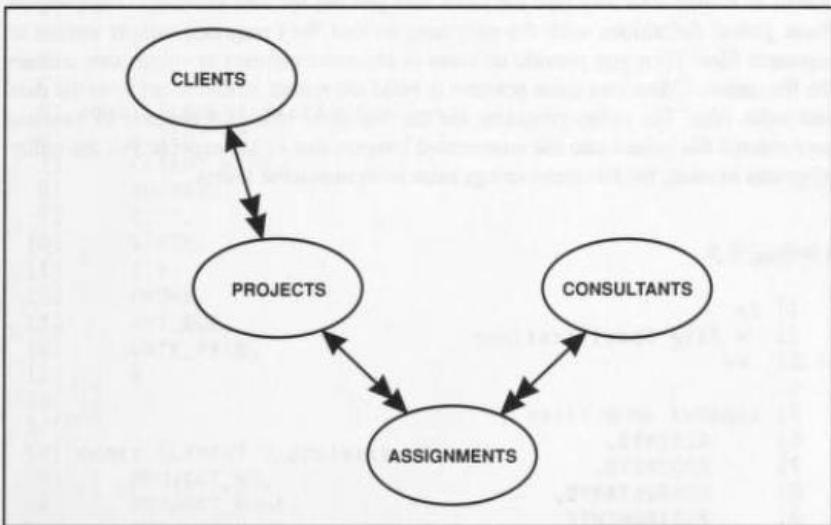


Figure 6-2. Addition of the ASSIGNMENTS File.

Next, you assign data elements to files by using the approach described in Chapter 2. This effort produces the following table:

CBS:

CLIENTS: client_no, client_name, address,

city, state, phone, amt_due, date_paid

PROJECTS: project_no, project_name, amt_expended,

client_no

CONSULTANTS: consultant_no, consultant_name

ASSIGNMENTS: consultant_no, project_no, rate

This table is the result of the database paper design. Now, you must translate it into the Cdata Data Definition Language. The expression of the file specifications in the DDL involves three stages. First, tell Cdata the names of the files; next, specify the data elements that are in each file; and, finally, describe record structures for the files.

Naming the Files

Naming the files involves two constructs, both seen in Listing 6.5. You will define files within an enumerated data type the same way you did for data elements. You compile these global definitions with the programs so that they may use integer values to represent files. Then you provide an array of character pointers to strings that contain the file names. Cdata uses these pointers to build file names so that it can open the data and index files. The utility programs use the file name character pointers to translate user-entered file names into the enumerated integers that Cdata expects. For the utility programs to work, the file name strings must be in uppercase letters.

Listing 6.5

```
1| /*  
2|  * File Specifications  
3|  */  
4|  
5| typedef enum files {  
6|     CLIENTS,  
7|     PROJECTS,  
8|     CONSULTANTS,  
9|     ASSIGNMENTS  
10| } DBFILE;  
11|  
12| const char *dbfiles [] = {  
13|     "CLIENTS",  
14|     "PROJECTS",  
15|     "CONSULTANTS",  
16|     "ASSIGNMENTS",  
17|     NULL  
18| };
```

Assigning Data Elements to Files

The next step in file specification defines the contents of each file in terms of its data elements. First, you provide an array of ELEMENT enumerated data types for each file. The array contains an entry for each data element in the file. The enumerated values come from the global data element definitions of the data element dictionary. Each array is a list of data element enumerated integers terminated by a zero value integer. Listing 6.6 shows the four arrays for the CBS database.

Listing 6.6

```

1| /*
2|  * File Contents
3| */
4|
5| const ELEMENT f_clients [] = {
6|     CLIENT_NO,
7|     CLIENT_NAME,
8|     ADDRESS,
9|     CITY,
10|    STATE,
11|    ZIP,
12|    PHONE,
13|    AMT_DUE,
14|    DATE_PAID,
15|    0
16| };
17|
18| const ELEMENT f_projects [] = {
19|     PROJECT_NO,
20|     PROJECT_NAME,
21|     AMT_EXPENDED,
22|     CLIENT_NO,
23|     0
24| };
25|
26| const ELEMENT f_consultants [] = {
27|     CONSULTANT_NO,
28|     CONSULTANT_NAME,
29|     0
30| };
31|
32| const ELEMENT f_assignments [] = {
33|     CONSULTANT_NO,
34|     PROJECT_NO,
35|     RATE,
36|     0

```

continued...

..from previous page (Listing 6.6)

```
37};  
38;  
39 const ELEMENT *file_ele [] = {  
40     f_clients,  
41     f_projects,  
42     f_consultants,  
43     f_assignments,  
44     0  
45};
```

Listing 6.6 also shows a fifth array named **file_ele**, which is an array of pointers to arrays of element enumerated data types. One pointer exists for each file in the database and points to the array of data elements for the file. This pointer array allows a program to get the address of a file's description by using this expression:

```
file_ele[filename]
```

This is where **filename** is one of the values assigned in the DBFILE enumerated data type defined in Listing 6.5.

Defining Record Structures

The last step in file specification defines a record structure for each file. In this step, you provide a structure that programs can use to declare a record buffer. The programs read into and write from the record buffers. The structure's members correspond to the record's data elements. Their lengths are taken from the data element dictionary with one added for the null string terminator. Listing 6.7 shows the four structures for the four files in the CBS database.

Listing 6.7

```

1| /*
2|  * File Record Structures
3|  */
4| struct clients {
5|     char client_no [6];
6|     char client_name [26];
7|     char address [26];
8|     char city [26];
9|     char state [3];
10|    char zip [6];
11|    char phone [11];
12|    char amt_due [9];
13|    char date_paid [7];
14| };
15|
16| struct projects {
17|     char project_no [6];
18|     char project_name [26];
19|     char amt_expended [10];
20|     char client_no [6];
21| };
22|
23| struct consultants {
24|     char consultant_no [6];
25|     char consultant_name [26];
26| };
27|
28| struct assignments {
29|     char consultant_no [6];
30|     char project_no [6];
31|     char rate [6];
32| };

```

You will see a convention for data identifier names in the Cdata DDL. Look at the clients file. Its DBFILE enumerated value is CLIENTS. Its file description is an array named **f_clients**. Its structure type is named **clients**. Similarly, the members in a structure definition carry the same names as the data elements they represent,

except that the global data names are in uppercase while the member names are not. The data element for client number is named **CLIENT_NO**, and its corresponding member name in a record is named **client_no**. If you define a structure of type **clients** with the name **client**, the client number's identifier is **client.client_no**.

Index Specifications

The last part of the DDL tells Cdata which data elements are key indexes in which files. In the example of CBS, the CLIENTS file has the **client_no** as the primary key; the PROJECTS file uses **proj_no**, and the CONSULTANTS file uses **consultant_no**. The ASSIGNMENTS file uses the concatenated data elements **consultant_no** and **project_no**. The database design is completed with the following table:

CBS:	
CLIENTS:	<client_no>, client_name, address, city, state, phone, amt_due, date_paid
PROJECTS:	<project_no>, project_name, amt_expended, client_no
CONSULTANTS:	<consultant_no>, consultant_name
ASSIGNMENTS:	<consultant_no, project_no>, rate

The Cdata DDL technique for defining index data elements involves building a series of integer arrays to describe the various key indexes and their characteristics. First, a file can have several indexes. Second, an index can consist of several concatenated data elements. Both of these dimensions are variable in length, so you must build an array architecture to represent any possibility.

Each index is represented in an array of ELEMENT enumerated data types that contains the values for the key data elements. Each key is an array. If a key contains several concatenated data elements, its array contains several data element values. If a file has several indexes, it will have several arrays. Each array is terminated by a zero value. Each file has an array of pointers to its index arrays. This array is terminated with a zero pointer.

The database has an array of pointers to the index pointer arrays of the files. Each entry in this array represents one file. This array is an array of pointers to pointers to integer arrays. The array is terminated with a null pointer, and it must be named **index_ele**.

Listing 6.8 shows the arrays required to complete the CBS Cdata DDL.

Listing 6.8

```
1  /*
2   * Index Specifications
3   */
4
5  const ELEMENT x1_clients [] = {
6      CLIENT_NO,
7      0
8  };
9
10 const ELEMENT *x_clients [] = {
11     x1_clients,
12     NULL
13 };
14
15 const ELEMENT x1_projects [] = {
16     PROJECT_NO,
17     0
18 };
19
20 const ELEMENT *x_projects [] = {
21     x1_projects,
22     NULL
23 };
24
25 const ELEMENT x1_consultants [] = {
26     CONSULTANT_NO,
27     0
28 };
29
30 const ELEMENT *x_consultants [] = {
31     x1_consultants,
32     NULL
33 };
34
```

continued...

...from previous page (Listing 6.8)

```
35| const ELEMENT x1_assignments [] = {
36|     CONSULTANT_NO,
37|     PROJECT_NO,
38|     0
39| };
40|
41| const ELEMENT x2_assignments [] = {
42|     CONSULTANT_NO,
43|     0
44| };
45|
46| const ELEMENT x3_assignments [] = {
47|     PROJECT_NO,
48|     0
49| };
50|
51| const ELEMENT *x_assignments [] = {
52|     x1_assignments,
53|     x2_assignments,
54|     x3_assignments,
55|     NULL
56| };
57|
58| const ELEMENT **index_ele [] = {
59|     x_clients,
60|     x_projects,
61|     x_consultants,
62|     x_assignments,
63|     NULL
64| };
```

THE CDATA DDL COMPILER

The application of C language features to describe a database schema to the DBMS software is an approach marked by its simplicity. It has helped the development of many applications since it was first developed, and the C language has served its purpose well. When the database becomes complex, however, the schema becomes difficult to manage. The proximity of items in their arrays is critical in this approach, and their order must be carefully maintained. If you add a data element to the dictionary, you must be careful to get its length in the proper place in the length array and its name in the proper place in the name array. If you add a file, you must carefully adjust the arrays that point to other arrays.

Because of the problems associated with maintaining a complex schema, it was decided that Cdata should use a compiled schema language. Listing 6.9 is the CBS schema coded in the Cdata Data Definition Language. This schema is shorter and simpler than all the enumerated data types, arrays, and structures of the DDL source code, yet it contains all the necessary components to describe the Consultant's Billing System database.

Listing 6.9 cbs.sch

```

1| ; ----- cbs.sch      schema for consultant's billing system
2|
3| ; ----- data element dictionary
4|
5| #schema CBS
6|
7| #dictionary
8|     CLIENT_NO,          Z, 5
9|     CLIENT_NAME,        A, 25
10|    ADDRESS,            A, 25
11|    CITY,               A, 25
12|    STATE,              A, 2
13|    ZIP,                N, 5
14|    PHONE,              N, 10, "(____)____-____"
15|    AMT_DUE,             C, 8, "$_____._"
16|    PROJECT_NO,          Z, 5
17|    PROJECT_NAME,        A, 25
18|    AMT_EXPENDED,        C, 9,   "$_____._"

```

continued...

...from previous page (Listing 6.9)

```
19| CONSULTANT_NO, Z, 5
20| CONSULTANT_NAME, A, 25
21| RATE, C, 5, "$_____.__"
22| PAYMENT, C, 9, "$_____.__"
23| EXPENSE, C, 9, "$_____.__"
24| HOURS, N, 2
25| DATE_PAID, D, 6, "__/__/__"
26| #end dictionary
27|
28| ; ---- file specifications
29|
30| #file CLIENTS
31| CLIENT_NO
32| CLIENT_NAME
33| ADDRESS
34| CITY
35| STATE
36| ZIP
37| PHONE
38| AMT_DUE
39| DATE_PAID
40| #end file
41|
42| #file PROJECTS
43| PROJECT_NO
44| PROJECT_NAME
45| AMT_EXPENDED
46| CLIENT_NO
47| #end file
48|
49| #file CONSULTANTS
50| CONSULTANT_NO
51| CONSULTANT_NAME
52| #end file
53|
```

continued.....

...from previous page (Listing 6.9)

```
54| #file ASSIGNMENTS
55|     CONSULTANT_NO
56|     PROJECT_NO
57|     RATE
58| #end file
59|
60| ; ----- index specifications
61|
62| #key CLIENTS      CLIENT_NO
63| #key PROJECTS    PROJECT_NO
64| #key CONSULTANTS CONSULTANT_NO
65| #key ASSIGNMENTS CONSULTANT_NO, PROJECT_NO
66| #key ASSIGNMENTS CONSULTANT_NO
67| #key ASSIGNMENTS PROJECT_NO
68|
69| #end schema CBS
```

This section will discuss the language first and then the program that compiles it. Throughout this discussion, refer to Listing 6.9 as an example.

THE CDATA DDL

The Cdata DDL is a text file that you can prepare with any text editor that produces ASCII files. It consists of lines of descriptive information about the data element dictionary, the files, and the indexes.

Comments

Any line that begins with a semicolon is a comment. The comment in a DDL is exactly like a comment in any other source language. Its purpose is to document the code for the human reader. The schema compiler ignores the comment.

DDL Directives

Directives begin with a pound sign (#) in column 1. One of several key phrases follows the pound sign. These phrases must appear in their proper order. The directives are as follows:

```
#schema  
#dictionary  
#end dictionary  
#file  
#end file  
#key  
#end schema
```

The first directive that appears in a Cdata DDL file is the **#schema** directive, which names the database. At least one space or tab character and the database name follow the directive. The compiler will use the database name in comments in the compiled C language schema. The **#schema** directive cannot appear anywhere else in the file.

The **#end schema** directive is the last directive that must appear in a Cdata DDL file. It cannot appear anywhere else in the file.

Data Element Dictionary

The data element dictionary is the first part of the database that you define. It begins with the **#dictionary** DDL directive and ends with the **#end dictionary** directive. The statements between the two directives describe the data elements. Each data element statement has four parts to describe the data element: its name, its data element type, its length, and its display mask. Commas separate the four parts. The name follows the C language convention for variable identifiers, and the data element type is one of the following codes:

```
A = alphanumeric  
C = currency  
Z = numeric, zero-filled  
N = numeric, space-filled  
D = date
```

The data element length is an integer that expresses the data element's length without the null string terminator of the C language. Dates are six characters long, but other

types can be any length you choose. The display mask is a string enclosed in double quotes, and the underscore character represents character positions in the mask. You can insert punctuation characters anywhere in the mask. The string for a date could be as follows:

"____"

You should include as many underscore characters as there are characters in the length of the data element. Even though you can assign any size to a data element, the system places a practical limit on the size of a display mask. You cannot display all of a data element if it is wider than your screen or printer. If you code a mask shorter than the data element, the data entry and display utility programs will truncate the data element when they display it.

File Specifications

The Cdata DDL allows you to specify as many files as you want in a database. File specifications begin with the **#file** directive, which names the file. The file name can be up to 31 characters and must obey the rules for C language identifiers. Since the DOS names for the data and index files use the first eight characters of the schema's file name, that file name must use DOS file name conventions and must be unique from other file names in the first eight positions.

Each line in a file specification names a data element. The data element name must be one of those already defined in the data element dictionary. You complete the specification of each file with the **#end file** directive.

Index Specifications

Index specifications tell Cdata which data elements to use for indexes in the files. The **#key** directive identifies an index specification, which is a single-line entry. You follow the **#key** directive keyword with a file name that you have already specified in a **#file** directive. You then follow the file name with one or more spaces. Finally, you list the data element name or names that make up the key. Where a key contains multiple data element names, you must separate the names with commas. Such a key definition becomes a concatenated key.

The first key named for a file is its primary key. Cdata will ensure that data values in a file's primary key data element are unique. Further, if the primary key data element appears in another file, Cdata relates the files. Cdata will not allow you to add a

record to the other file unless its value for the data element is null or matched to a record in this file. As an example, with Cdata you cannot assign a project to a nonexistent client but you can establish a project before the client is known.

Nonprimary keys are secondary keys. These keys can have multiple values in the same file. So, if you wanted to establish the data element **client_no** as a secondary key in the PROJECTS file, you could have more than one project record for the same client. But you could not establish more than one project record with the same project number because **project_no** is the primary key into the PROJECTS file.

THE CDATA DBMS HEADER SOURCE FILE

To compile most of the programs that follow in this and later chapters, you will need the source file named cdata.h. This file, shown in Listing 6.10, contains global definitions that you use throughout the system during compilation. You can change these definitions to suit your own application.

Listing 6.10 cdata.h

```
1: /* ----- cdata.h ----- */
2:
3: #include <stdio.h>
4:
5: #ifndef CDATA_H
6: #define CDATA_H
7:
8: #define MXFILS    11      /* maximum files in a data base      */
9: #define MXELE    100     /* maximum data elements in a file   */
10: #define MXINDEX   5      /* maximum indexes per file         */
11: #define MXKEYLEN  80     /* maximum key length for indexes   */
12: #define MXCAT    3      /* maximum elements per index       */
13: #define NAMLEN   31     /* data element name length         */
14:
15: /* initialize this to call your function for data base errors */
16: extern void (*database_message)(void);
17:
```

continued...

...from previous page (Listing 6.10)

```

18| #define ERROR -1
19| #define OK 0
20|
21| #ifndef TRUE
22| #define TRUE 1
23| #define FALSE 0
24| #endif
25|
26| typedef long RPTR;      /* B-tree node and file address */ *
27|
28| #ifndef APPLICATION_H
29| typedef int DBFILE;
30| typedef int ELEMENT;
31| #endif
32|
33| /* ----- schema as built for the application ----- */
34| extern const char *dbfiles[];      /* file names */ *
35| extern const char *denames[];      /* data element names */ *
36| extern const char *elmask[];       /* data element masks */ *
37| extern const char eltype[];       /* data element types */ *
38| extern const int ellen[];         /* data element lengths */ *
39| extern const ELEMENT *file_ele[]; /* file data elements */ *
40| extern const ELEMENT **index_ele[]; /* index data elements */ *
41|
42| /* ----- data base prototypes ----- */
43|
44| /* ----- Cdata API functions ----- */
45| void db_open(const char *, const DBFILE *);
46| int add_rcd(DBFILE, void *);
47| int find_rcd(DBFILE, int, char *, void *);
48| int verify_rcd(DBFILE, int, char *);
49| int first_rcd(DBFILE, int, void *);
50| int last_rcd(DBFILE, int, void *);
51| int next_rcd(DBFILE, int, void *);
52| int prev_rcd(DBFILE, int, void *);
53| int rtn_rcd(DBFILE, void *);
54| int del_rcd(DBFILE);

```

continued...

..from previous page (Listing 6.10)

```
55| int curr_rcd(DBFILE, int, void *);  
56| int seqrcd(DBFILE, void *);  
57| void db_cls(void);  
58| void dberror(void);  
59| int rlen(DBFILE);  
60| void init_rcd(DBFILE, void *);  
61| void clrrcd(void *, const ELEMENT *);  
62| int epos(ELEMENT, const ELEMENT *);  
63| void rcd_fill(const void *, void *, const ELEMENT *,  
64|                 const ELEMENT *);  
65|  
66| /* ----- functions used by Cdata utility programs ----- */  
67| void build_index(char *, DBFILE);  
68| int add_indexes(DBFILE, void *, RPTR);  
69| DBFILE filename(char *);  
70| void name_cvt(char *, char *);  
71| int ellist(int, char **, ELEMENT *);  
72| void clist(FILE *,const ELEMENT *,const ELEMENT *,  
73|             void *,const char *);  
74| void test_eop(FILE *, const char *, const ELEMENT *);  
75| void dblist(FILE *, DBFILE, int, const ELEMENT *);  
76|  
77| /* ----- dbms error codes for errno return ----- */  
78| enum dberrors {  
79|     D_NF=1,      /* record not found */  
80|     D_PRIOR,    /* no prior record for this request */  
81|     D_EOF,       /* end of file */  
82|     D_BOF,       /* beginning of file */  
83|     D_DUPL,     /* primary key already exists */  
84|     D_OM,        /* out of memory */  
85|     D_INDXC,    /* index corrupted */  
86|     D_IOERR     /* i/o error */  
87| };  
88|  
89| #endif
```

Remember the earlier discussion about the limitations placed by DBMS packages. Cdata also has limitations, but you can change them. The global values, MXFILS, MXELE, and MXINDEX in cdata.h define the maximum number of files in a database, the maximum number of data elements in a file, and the maximum number of indexes per file. The global symbol MXKEYLEN establishes the maximum character size of a key data element or elements. The MXCAT symbol defines the maximum number of data elements that you can concatenate into an index. If you change any of these global symbols, you must recompile the Cdata programs and your applications programs.

Cdata.h sets the type of a file address to the long integer with the RPTR **typedef** statement, which allows a 32-bit file address, meaning that files can have up to 2^{32} records. If your files each contain 65535 or fewer records, you can change this **typedef** from a **long** to an **unsigned** integer and reduce the disk overhead for your database.

Observe the **typedefs** for DBFILE and ELEMENT. These **typedefs** are integers when you compile cdata.h into the database code in this chapter or the utility programs in Chapter 7. When you include the file in an application, the **typedefs** are assigned to the enumerated data types of the files and the data element dictionary.

Cdata.h declares the Cdata schema external arrays that define files, their names, formats, and indexes. Remember that you compile the schema separately and that Cdata and the applications and utility programs treat them as externally declared arrays. The declaration of these external arrays is in cdata.h to make them available to your code as well as Cdata's code.

The prototypes for the Cdata applications program interface (API) appear in cdata.h. There are also prototypes for some global functions used internally by the Cdata programs.

Cdata.h defines the error codes that Cdata functions return in the **dberrors** enumerated data type.

THE COMPILER, SCHEMA.C

Schema.c (Listing 6.11) is a program that reads the Cdata DDL and compiles it into two C language source files that comply with the schema requirements of Cdata. If you run the schema program with the CBS DDL in Listing 6.9 as input, it will

produce two files containing the source files from Listings 6.1 through 6.8. The first output file, cbs.h, contains the database record structure definitions and the **enum** statements for the data element and file names. The second file, cbs.c, contains the string pointer arrays for file names, data element names, data element types, and display masks, all of which support the query and report utilities and application programs that use screen drivers. The cbs.c file also defines the arrays of record and index definitions that constitute the database schema. You must include the cbs.h in any program that links with Cdata so that you can use the global file and data element names in your calls to the Cdata DML functions. You must compile cbs.c, which itself includes cbs.h, into a relocatable object module and link it with your program and the DML functions.

Listing 6.11 schema.c

```

1| /* ----- schema.c ----- */
2|
3| /*
4| * Build two C language schema files from a CDATA schema
5| * Input:
6| *     <application>.sch
7| * Output:
8| *     <application>.h contains:
9| *         data element enums
10| *         file enums
11| *         record structs
12| *     <application>.c contains:
13| *         ascii strings for file and data element names
14| *         data element masks array
15| *         data element types array
16| *         element length array
17| *         data base schema arrays
18| */
19|
20| #include <stdio.h>
21| #include <stdlib.h>
22| #include <string.h>
23| #include <ctype.h>
24| #include <process.h>
```

continued...

...from previous page (Listing 6.11)

```

25| #include "cdata.h"
26|
27| static struct dict {           /* data element dictionary */
28|     char dename [NAMLEN+1];    /* name */
29|     char detype;              /* type */
30|     int delen;                /* length */
31|     char *demask;             /* display mask */
32| } dc [MXELE];
33|
34| static int dectr = 0;          /* data elements in dictionary */
35| static int fctr = 0;           /* files in data base */
36| static char filenames [MXFILS] [NAMLEN+1]; /* filenames */
37| static int fileele [MXFILS] [MXELE];        /* file elements */
38| static int ndxele [MXFILS] [MXINDEX] [MXCAT]; /* indexes */
39|
40| static char word[NAMLEN+1];
41| static int lnctr = 0;          /* input stream line counter */
42| static char ln [160];
43|
44| /* ----- error codes ----- */
45| enum error_codes {
46|     ER_NAME=1,
47|     ER_LENGTH,
48|     ER_COMMA,
49|     ER_TYPE,
50|     ER_QUOTE,
51|     ER_SCHEMA,
52|     ER_COMMAND,
53|     ER_EOF,
54|     ER_DUPLNAME,
55|     ER_UNKNOWN_ELEMENT,
56|     ER_TOOMANY_ELEMENTS,
57|     ER_MEMORY,
58|     ER_UNKNOWN_FILENAME,
59|     ER_TOOMANY_INDEXES,
60|     ER_TOOMANY_IN_INDEX,
61|     ER_DUPL_ELEMENT,
62|     ER_TOOMANY_FILES,
```

continued...

...from previous page (Listing 6.11)

```

63:     ER_NOSCHEMA,
64:     ER_NOSUCH_SCHEMA,
65:     ER_TERMINAL
66: };
67:
68: /* ----- error messages ----- */
69: static struct {
70:     enum error_codes ec;
71:     char *errmsg;
72: } ers[] = {
73:     {ER_NAME, "invalid name",
74:      ER_LENGTH, "invalid length",
75:      ER_COMMA, "comma missing",
76:      ER_TYPE, "invalid data type",
77:      ER_QUOTE, "quote missing",
78:      ER_SCHEMA, "#schema missing",
79:      ER_COMMAND, "#<command> missing",
80:      ER_EOF, "unexpected end of file",
81:      ER_DUPLNAME, "duplicate file name",
82:      ER_UNKNOWN_ELEMENT, "unknown data element",
83:      ER_TOOMANY_ELEMENTS, "too many data elements",
84:      ER_MEMORY, "out of memory",
85:      ER_UNKNOWN_FILENAME, "unknown file name",
86:      ER_TOOMANY_INDEXES, "too many indexes in file",
87:      ER_TOOMANY_IN_INDEX, "too many elements in index",
88:      ER_DUPL_ELEMENT, "duplicate data element",
89:      ER_TOOMANY_FILES, "too many files",
90:      ER_NOSCHEMA, "no schema file specified",
91:      ER_NOSUCH_SCHEMA, "no such schema file",
92:      ER_TERMINAL, NULL
93: };
94:
95: static void de_dict(void);
96: static void files(void);
97: static void keys(void);
98: static void defout(const char *);
99: static void schout(const char *);
100: static void lcase(char *, const char *);
101: static void error(const enum error_codes);

```

continued...

...from previous page (Listing 6.11)

```

102| static void get_line(void);
103| static void skip_white(char **);
104| static char *get_word(char *);
105| static void name_val(void);
106| static void numb_val(void);
107| static void expect_comma(char **);
108|
109| #define iswhite(c) ((c)==' '||(c)=='\t')
110| #define REMARK ';'
111|
112| static FILE *fp;
113|
114| /* ----- main program ----- */
115| void main(int argc, char *argv[])
116|
117|     char fname[64];
118|     char *cp;
119|
120|     if (argc > 1) {
121|         strcpy(fname, argv[1]);
122|         if ((cp = strrchr(fname, '.')) == NULL) {
123|             cp = fname+strlen(fname);
124|             strcpy(cp, ".sch");
125|         }
126|         if ((fp = fopen(fname, "r")) == NULL) {
127|             error(ER_NOSUCH_SCHEMA);
128|             exit(1);
129|         }
130|         *cp = '\0';
131|         get_line();
132|         if (strncmp(ln, "#schema ", 8))
133|             error(ER_SCHEMA);
134|         else {
135|             get_word(ln + 8);
136|             name_val();
137|         }
138|         get_line();
139|         while (strncmp(ln, "#end schema", 11)) {

```

continued...

..from previous page (Listing 6.11)

```

140:             if (strncpy(ln, "#dictionary", 11) == 0)
141:                 de_dict();
142:             else if (strncpy(ln, "#file ", 6) == 0)
143:                 files();
144:             else if (strncpy(ln, "#key ", 5) == 0)
145:                 keys();
146:             else
147:                 error(ER_COMMAND);
148:             get_line();
149:         }
150:         fclose(fp);
151:         defout(fname);
152:         schout(fname);
153:     }
154:     else
155:         error(ER_NOSCHEMA);
156:     exit(0);
157: }
158: /* ----- build the data element dictionary ----- */
159: static void de_dict(void)
160: {
161:     char *cp, *cp1;
162:     int el, masklen, buildmask;
163:     while (TRUE) {
164:         get_line();
165:         if (strncpy(ln, "#end dictionary", 15) == 0)
166:             break;
167:         if (dectr == MXELE) {
168:             error(ER_TOOMANY_ELEMENTS);
169:             continue;
170:         }
171:         cp = get_word(ln);
172:         name_val();
173:         for (el = 0; el < detr; el++)
174:             if (strcmp(word, dc[el].dename) == 0) {
175:                 error(ER_DUPL_ELEMENT);
176:                 continue;
177:             }

```

continued...

...from previous page (Listing 6.11)

```

178      }
179      strcpy(dc[dectr].dename, word);
180      expect_comma(&cp);
181      skip_white(&cp);
182      switch (*cp) {
183          case 'A':
184          case 'Z':
185          case 'C':
186          case 'N':
187          case 'D': break;
188          default : error(ER_TYPE);
189          continue;
190      }
191      dc[dectr].detype = *cp++;
192      expect_comma(&cp);
193      cp = get_word(cp);
194      numb_val();
195      dc[dectr].delen = atoi(word);
196      skip_white(&cp);
197      if (*cp++ == ',') {
198          buildmask = FALSE;
199          /* --- comma means display mask is coded --- */
200          skip_white(&cp);
201          if (*cp != '"') {
202              error(ER_QUOTE);
203              continue;
204          }
205          cp1 = cp + 1;
206          while (*cp1 != '"' && *cp1 && *cp1 != '\n')
207              cp1++;
208          if (*cp1++ != '"') {
209              error(ER_QUOTE);
210              continue;
211          }
212          *cp1 = '\0';
213          masklen = (cp1-cp)+1;
214      }

```

continued...

...from previous page (Listing 6.11)

```

215:     else    {
216:         /* ----- no display mask, build one ----- */
217:         buildmask = TRUE;
218:         masklen = dc[dectr].delen+3;
219:     }
220:     if ((dc[dectr].demask = malloc(masklen)) == NULL) {
221:         error(ER_MEMORY);
222:         exit(1);
223:     }
224:     if (buildmask) {
225:         dc[dectr].demask[0] = "";
226:         memset(dc[dectr].demask+1, '_', masklen-3);
227:         dc[dectr].demask[masklen-2] = "";
228:         dc[dectr].demask[masklen-1] = '\0';
229:     }
230:     else
231:         strcpy(dc[dectr].demask, cp);
232:     dectr++;
233: }
234 }
235
236 /* ----- build the file definitions ----- */
237 static void files(void)
238 {
239:     int i, el = 0;
240:     if (fctr == MXFILS)
241:         error(ER_TOOMANY_FILES);
242:     get_word(ln + 6);           /* get the file name */
243:     name_val();                /* validate it */
244:     for (i = 0; i < fctr; i++) /* already assigned? */
245:         if (strcmp(word, filenames[i]) == 0)
246:             error(ER_DUPLNAME);
247:     strcpy(filenames[fctr], word);
248:     /* ----- process the file's data elements ----- */
249:     while (TRUE) {
250:         get_line();
251:         if (strncmp(ln, "#end file", 9) == 0)
252:             break;
253:         if (el == MXELE) {
254:             error(ER_TOOMANY_ELEMENTS);
255:             continue;

```

continued...

..from previous page (Listing 6.11)

```

256:         }
257:         get_word(ln);           /* get a data element */
258:         for (i = 0; i < dectr; i++) /* in dictionary? */
259:             if (strcmp(word, dc[i].dename) == 0)
260:                 break;
261:             if (i == dectr)
262:                 error(ER_UNKNOWN_ELEMENT);
263:             else if (fctr < MXFILS)
264:                 fileele [fctr] [el++] = i + 1; /* post to file */
265:         }
266:         if (fctr < MXFILS)
267:             fctr++;
268:     }
269:
270: /* ----- build the index descriptions ----- */
271: static void keys(void)
272: {
273:     char *cp;
274:     int f, el, x, cat = 0;
275:     cp = get_word(ln + 5);      /* get the file name */
276:     for (f = 0; f < fctr; f++) /* in the schema? */
277:         if (strcmp(word, filenames[f]) == 0)
278:             break;
279:     if (f == fctr) {
280:         error(ER_UNKNOWN_FILENAME);
281:         return;
282:     }
283:     for (x = 0; x < MXINDEX; x++)
284:         if (*ndxele [f] [x] == 0)
285:             break;
286:     if (x == MXINDEX) {
287:         error(ER_TOOMANY_INDEXES);
288:         return;
289:     }
290:     while (cat < MXCAT) {
291:         cp = get_word(cp);          /* get index name */
292:         for (el = 0; el < dectr; el++) /* in dictionary? */
293:             if (strcmp(word, dc[el].dename) == 0)
294:                 break;

```

continued...

..from previous page (Listing 6.11)

```

295:         if (el == dectr)      {
296:             error(ER_UNKNOWN_ELEMENT);
297:             break;
298:         }
299:         ndxele [f] [x] [cat++] = el + 1; /* post element */
300:         skip_white(&cp);
301:         if (*cp++ != ',')           /* concatenated index? */
302:             break;
303:         if (cat == MXCAT)          {
304:             error(ER_TOOMANY_IN_INDEX);
305:             break;
306:         }
307:     }
308: }
309:
310: /* ----- write the data base .h header file:
311:    schema enums and struct definitions ----- */
312: static void defout(const char *fname)
313: {
314:     int f, el, fel;
315:     char name [NAMLEN+1];
316:     char fn[64];
317:
318:     strcpy(fn, fname);
319:     strcat(fn, ".h");
320:     fp = fopen(fn, "w");
321:
322:     fprintf(fp, /* ----- %s ----- */"\n", fn);
323:     fprintf(fp, "\n#define APPLICATION_H\n");
324:
325:     /* ----- data element enums ----- */
326:     fprintf(fp, "\ntypedef enum elements {\n");
327:     for (el = 0; el < dectr; el++)
328:         fprintf(fp, "\n\t%s%s,", dc[el].dename, el ? "" : "=1");
329:     fprintf(fp, "\n\tTermElement = 32367");
330:     fprintf(fp, "\n} ELEMENT;\n");
331:     /* ----- write the file enum statements ----- */
332:     fprintf(fp, "\ntypedef enum files {\n");
333:     for (f = 0; f < fctr; f++)

```

continued...

...from previous page (Listing 6.11)

```

334:         fprintf(fp, "\n\t%s,", filenames [f]);
335:         fprintf(fp, "\n\tTermFile = 32367");
336:         fprintf(fp, "\n\tDBFILE;\n");
337:         /* ----- write the record structures ----- */
338:         for (f = 0; f < fctr; f++) {
339:             lcase(name, filenames [f]);
340:             fprintf(fp, "\nstruct %s {" , name);
341:             el = 0;
342:             while ((fel = fileele[f] [el++]) != 0) {
343:                 lcase(name, dc[fel-1].dename);
344:                 fprintf(fp, "\n\tchar %s [%d];",
345:                         name, dc[fel-1].delen + 1);
346:             }
347:             fprintf(fp, "\n);\n");
348:         }
349:         fprintf(fp, "\n#include \"cdata.h\"\n");
350:         fclose(fp);
351:     }
352:
353: /* ----- write the data base schema source code ----- */
354: static void schout(const char *fname)
355: {
356:     int f, el, x, x1, cat, fel;
357:     char name [NAMLEN+1];
358:     char fn[64];
359:
360:     strcpy(fn, fname);
361:     strcat(fn, ".c");
362:     fp = fopen(fn, "w");
363:
364:     fprintf(fp,"/* ----- %s ----- */\n", fn);
365:     fprintf(fp, "\n#include \"%s.h\"\n", fname);
366:
367:     /* ----- data element ascii names ----- */
368:     fprintf(fp, "\nconst char *denames [] = {");
369:     for (el = 0; el < dectr; el++)
370:         fprintf(fp, "\n\t\"%s\",", dc[el].dename);
371:     fprintf(fp, "\n\tNULL\n};\n");
372:     /* ----- data element types ----- */

```

continued...

...from previous page (Listing 6.11)

```

373:     fprintf(fp, "\nconst char eltype [] = \"\"");
374:     for (el = 0; el < dectr; el++)
375:         putc(dc[el].dtype, fp);
376:     fprintf(fp, "\";\n");
377:     /* ----- data element display masks ----- */
378:     fprintf(fp, "\nconst char *elmask [] = {");
379:     for (el = 0; el < dectr; el++)
380:         fprintf(fp, (el < dectr-1 ?
381:                         "\n\t%", :
382:                         "\n\t%", dc[el].demask));
383:     fprintf(fp, "\n};\n");
384:     free(dc[el].demask);
385:     /* ----- write the ascii file name strings ----- */
386:     fprintf(fp, "\nconst char *dbfiles [] = {");
387:     for (f = 0; f < fctr; f++)
388:         fprintf(fp, "\n\t\"%s\",", filenames [f]);
389:     fprintf(fp, "\n\tNULL\n};\n");
390:
391:     /* ----- data element lengths ----- */
392:     fprintf(fp, "\n\nconst int ellen [] = {");
393:     for (el = 0; el < dectr; el++) {
394:         if ((el % 25) == 0)
395:             fprintf(fp, "\n\t");
396:         fprintf(fp, (el < dectr-1 ? "%d," : "%d"), dc[el].delen);
397:     }
398:     fprintf(fp, "\n};\n");
399:     /* ----- write the file contents arrays ----- */
400:     for (f = 0; f < fctr; f++) {
401:         lcase(name, filenames [f]);
402:         fprintf(fp, "\n\nconst ELEMENT f_%s [] = {",
403:                         name);
404:         el = 0;
405:         while ((fel = fileele[f] [el++]) != 0)
406:             fprintf(fp, "\n\t%", dc[fel-1].dename);
407:         fprintf(fp, "\n\t0\n};\n");
408:     }
409:     /* ----- write the file list pointer array ----- */
410:     fprintf(fp, "\n\nconst ELEMENT *file_ele [] = {");
411:     for (f = 0; f < fctr; f++) {
412:         lcase(name, filenames [f]);

```

continued...

...from previous page (Listing 6.11)

```

413|         fprintf(fp, "\n\tf_%s,", name);
414|     }
415|     fprintf(fp, "\n\t\t0\n");
416|     /* ----- write the index arrays ----- */
417|     for (f = 0; f < fctr; f++) {
418|         lcase(name, filenames [f]);
419|         for (x = 0; x < MXINDEX; x++) {
420|             if (*ndxele [f] [x] == 0)
421|                 break;
422|             fprintf(fp,
423|                     "\nconst ELEMENT x%d_xs [] = {",
424|                         x + 1, name);
425|             for (cat = 0; cat < MXCAT; cat++)
426|                 if (ndxele [f] [x] [cat])
427|                     fprintf(fp, "\n\t\t%s",
428|                             dc[ndxele [f] [x] [cat] - 1].dename);
429|             fprintf(fp, "\n\t\t0\n};\n");
430|         }
431|         fprintf(fp, "\nconst ELEMENT *x_xs [] = {",
432|                         name);
433|         for (x1 = 0; x1 < x; x1++)
434|             fprintf(fp, "\n\ttx%d_xs,", x1 + 1, name);
435|         fprintf(fp, "\n\tNULL\n");
436|     }
437|     fprintf(fp, "\nconst ELEMENT **index_ele [] = {");
438|     for (f = 0; f < fctr; f++) {
439|         lcase(name, filenames [f]);
440|         fprintf(fp, "\n\ttx_xs,", name);
441|     }
442|     fprintf(fp, "\n\tNULL\n");
443|
444|     fprintf(fp, "\n\n#endif NULL_IS_DEFINED");
445|     fprintf(fp, "\n\t#undef NULL");
446|     fprintf(fp, "\n\t#undef NULL_IS_DEFINED");
447|     fprintf(fp, "\n#endif\n");
448|
449|     fclose(fp);
450| }
```

continued...

...from previous page (Listing 6.11)

```

451: /* ----- convert a name to lower case ----- */
452: static void lcase(char *s1, const char *s2)
453: {
454:     while (*s2) {
455:         *s1 = tolower(*s2);
456:         s1++;
457:         s2++;
458:     }
459:     *s1 = '\0';
460: }
461: }
462:
463: /* --- get a line of data from the schema input stream --- */
464: static void get_line(void)
465: {
466:     char *cp;
467:     *ln = '\0';
468:     while (*ln == '\0' || *ln == REMARK || *ln == '\n') {
469:         cp = fgets(ln, 120, fp);
470:         if (cp == NULL) {
471:             error(ER_EOF);
472:             exit(1);
473:         }
474:         lnctr++;
475:     }
476: }
477:
478: /* ----- skip over white space ----- */
479: static void skip_white(char **s)
480: {
481:     while (iswhite(**s))
482:         (**s)++;
483: }
484:
485: /* ----- get a word from a line of input ----- */
486: static char *get_word(char *cp)
487: {
488:     int wl = 0, fst = 0;

```

continued...

...from previous page (Listing 6.11)

```

489 skip_white(&cp);
490 while (*cp && *cp != '\n' &&
491         *cp != ',' &&
492         iswhite(*cp) == 0) {
493     if (wl == NAMLEN && fst == 0) {
494         error(ER_NAME);
495         fst++;
496     }
497     else
498         word [wl++] = *cp++;
499 }
500 word [wl] = '\0';
501 return cp;
502 }
503 }
504
505 /* ----- validate a name ----- */
506 static void name_val(void)
507 {
508     char *s = word;
509     if (isalpha(*s)) {
510         while (isalpha(*s) || isdigit(*s) || *s == '_') {
511             *s = toupper(*s);
512             s++;
513         }
514         if (*s == '\0')
515             return;
516     }
517     error(ER_NAME);
518 }
519
520 /* ----- validate a number ----- */
521 static void numb_val(void)
522 {
523     char *s = word;
524     do {
525         if (isdigit(*s++) == 0) {
526             error(ER_LENGTH);
527             break;

```

continued...

..from previous page (Listing 6.11)

```

528     }
529     } while (*s);
530 }
531
532 /* ----- expect a comma next ----- */
533 static void expect_comma(char **cp)
534 {
535     skip_white(cp);
536     if ((*(*cp)++) != ',')
537         error(ER_COMMA);
538 }
539
540 /* ----- errors ----- */
541 static void error(const enum error_codes n)
542 {
543     static int erct = 0;
544     static int erlin = 0;
545     int err;
546
547     for (err = 0; ers[err].ec != ER_TERMINAL; err++)
548         if (n == ers[err].ec)
549             break;
550     if (erlin != lnctr) {
551         erlin = lnctr;
552         fprintf(stderr, "\nLine %d: %s", lnctr, ln);
553     }
554     fprintf(stderr, "Error %d: %s\n", n, ers[err].errmsg);
555     if (erct++ == 5) {
556         erct = 0;
557         fprintf(stderr, "\nContinue? (y/n) ... ");
558         if (tolower(getc(stdin)) != 'y')
559             exit(1);
560     }
561 }
```

Listings 6.12 and 6.13 are cbs.h and cbs.c as the schema program produces them from the cbs.sch file.

Listing 6.12 cbs.h

```

1  /* ----- cbs.h ----- */
2
3 #define APPLICATION_H
4
5 typedef enum elements {
6     CLIENT_NO=1,
7     CLIENT_NAME,
8     ADDRESS,
9     CITY,
10    STATE,
11    ZIP,
12    PHONE,
13    AMT_DUE,
14    PROJECT_NO,
15    PROJECT_NAME,
16    AMT_EXPENDED,
17    CONSULTANT_NO,
18    CONSULTANT_NAME,
19    RATE,
20    PAYMENT,
21    EXPENSE,
22    HOURS,
23    DATE_PAID,
24    TermElement = 32367
25 } ELEMENT;
26
27 typedef enum files {
28     CLIENTS,
29     PROJECTS,
30     CONSULTANTS,
31     ASSIGNMENTS,
32     TermFile = 32367
33 } DBFILE;
34
35 struct clients {
36     char client_no [6];
37     char client_name [26];
38     char address [26];

```

continued...

...from previous page (Listing 6.12)

```
39:     char city [26];
40:     char state [3];
41:     char zip [6];
42:     char phone [11];
43:     char amt_due [9];
44:     char date_paid [7];
45: };
46:
47: struct projects {
48:     char project_no [6];
49:     char project_name [26];
50:     char amt_expended [10];
51:     char client_no [6];
52: };
53:
54: struct consultants {
55:     char consultant_no [6];
56:     char consultant_name [26];
57: };
58:
59: struct assignments {
60:     char consultant_no [6];
61:     char project_no [6];
62:     char rate [6];
63: };
64:
65: #include "cdata.h"
```

Listing 6.13 cbs.c

```

1| /* ----- cbs.c ----- */
2|
3| #include "cbs.h"
4|
5| const char *denames [] = {
6|   "CLIENT_NO",
7|   "CLIENT_NAME",
8|   "ADDRESS",
9|   "CITY",
10|  "STATE",
11|  "ZIP",
12|  "PHONE",
13|  "AMT_DUE",
14|  "PROJECT_NO",
15|  "PROJECT_NAME",
16|  "AMT_EXPENDED",
17|  "CONSULTANT_NO",
18|  "CONSULTANT_NAME",
19|  "RATE",
20|  "PAYMENT",
21|  "EXPENSE",
22|  "HOURS",
23|  "DATE_PAID",
24|  NULL
25| };
26|
27| const char eltype [] = "ZAAAANNCZACZACCCND";
28|
29| const char *elmask [] = {
30|   "____",
31|   "_____",
32|   "_____",
33|   "_____",
34|   "___",
35|   "____",
36|   "(____)____-____",
37|   "$_____.____",
38|   "____",

```

continued...

...from previous page (Listing 6.13)

continued...

..from previous page (Listing 6.13)

```
77 const ELEMENT f_projects [] = {
78     PROJECT_NO,
79     PROJECT_NAME,
80     AMT_EXPENDED,
81     CLIENT_NO,
82     0
83 };
84
85 const ELEMENT f_consultants [] = {
86     CONSULTANT_NO,
87     CONSULTANT_NAME,
88     0
89 };
90
91 const ELEMENT f_assignments [] = {
92     CONSULTANT_NO,
93     PROJECT_NO,
94     RATE,
95     0
96 };
97
98 const ELEMENT *file_ele [] = {
99     f_clients,
100    f_projects,
101    f_consultants,
102    f_assignments,
103    0
104 };
105
106 const ELEMENT x1_clients [] = {
107     CLIENT_NO,
108     0
109 };
110
111 const ELEMENT *x_clients [] = {
112     x1_clients,
113     NULL
114 };
```

continued...

...from previous page (Listing 6.13)

```
115 | const ELEMENT x1_projects [] = {
116 |     PROJECT_NO,
117 |     0
118 | };
119 |
120 | const ELEMENT *x_projects [] = {
121 |     x1_projects,
122 |     NULL
123 | };
124 |
125 |
126 | const ELEMENT x1_consultants [] = {
127 |     CONSULTANT_NO,
128 |     0
129 | };
130 |
131 | const ELEMENT *x_consultants [] = {
132 |     x1_consultants,
133 |     NULL
134 | };
135 |
136 | const ELEMENT x1_assignments [] = {
137 |     CONSULTANT_NO,
138 |     PROJECT_NO,
139 |     0
140 | };
141 |
142 | const ELEMENT x2_assignments [] = {
143 |     CONSULTANT_NO,
144 |     0
145 | };
146 |
147 | const ELEMENT x3_assignments [] = {
148 |     PROJECT_NO,
149 |     0
150 | };
```

continued...

...from previous page (Listing 6.13)

```

151|
152| const ELEMENT *x_assignments [] = {
153|     x1_assignments,
154|     x2_assignments,
155|     x3_assignments,
156|     NULL
157| };
158|
159| const ELEMENT **index_ele [] = {
160|     x_clients,
161|     x_projects,
162|     x_consultants,
163|     x_assignments,
164|     NULL
165| };
166|
167|
168| #ifdef NULL_IS_DEFINED
169|     #undef NULL
170|     #undef NULL_IS_DEFINED
171| #endif

```

Observe the **TermElement** and **TermFile** constants in the ELEMENT and DBFILE enumerated data types. These values guarantee that the enumerated types are equated to integers. Some compilers—WATCOM, for example—will make an enumerated data type into a signed char if the values do not exceed 127. Because the database software deals with the arrays as arrays of integers, the schema must include these constants to assure that the enums will be integers as well.

To run schema.c, enter its name and provide the name of the database on the command line. There must be a file with the .SCH extension and the database name as the file name, for example, cbs.sch:

A>schema cbs

This command reads the cbs.sch file and produces the cbs.h and cbs.c files.

Schema.c is an example of a program that uses some of the standard features of the C language. It is a simple compiler that compiles DDL statements into C statements.

Schema.c uses the C language **argc/argv** command line parameter convention to allow the operator to specify the database name. You will find a description of argc and argv on page 110 of *The C Programming Language* (Kernighan and Ritchie, Prentice Hall, 1978). Standard C includes this command line argument convention.

Schema.c is rigorous in its syntax checking. Earlier, you learned the Cdata DDL syntax. The compiler makes sure that you stick to the rules. If you code some bad syntax, the compiler delivers an appropriate error message and keeps running. The error message identifies the line number of the DDL file where the program found the error and displays the offending DDL statement and the compiler's complaint about the statement. Because one error could cause more errors to occur, the compiler pauses every five errors and asks if you want to keep going. You might find it more productive to quit, correct the errors, and restart the compiler.

Following is a list of the compiler's error messages and their meanings.

Error 1: invalid name

You have coded a file or data element name that is incorrect. Names must begin with letters and may contain letters, digits, and underscores.

Error 2: invalid length

The specification of a data element length is not all digits.

Error 3: comma missing

A comma is missing where one is expected.

Error 4: invalid data type

The data type for a data element is not one of the following: A, Z, C, N, or D.

Error 5: quote missing

The display mask for a data element is not bound by double quotation marks. Either the beginning mark or the terminating mark is missing.

Error 6: #schema missing

The compiler did not find the #schema statement where it expected it to be.

Error 7: #<command> missing

The compiler has expected to find either a #dictionary, a #file, or a #key DDL statement and found none of these.

Error 8: unexpected end of file

The compiler has reached the end of file on the DDL statement file without reaching a logical end to the specification of the database.

Error 9: duplicate file name

You have given two files the same name.

Error 10: unknown data element

You have specified a data element in a file or index that is not in the data element dictionary.

Error 11: too many data elements

The number of data elements in your schema exceeds the value assigned to the global symbol MXELE in cdata.h.

Error 12: out of memory

The compiler has run out of memory while attempting to allocate a buffer.

Error 13: unknown file name

The index specification names a file that does not appear in the file specifications.

Error 14: too many indices in file

You have assigned more keys to a file than the global symbol MXINDEX specifies. You will find MXINDEX defined in cdata.h.

Error 15: too many data elements in file

You have given more data elements to a file than the global symbol MXELE specifies. You will find MXELE defined in cdata.h.

Error 16: duplicate data element

Two data elements in the dictionary have the same name.

Error 17: too many files

You have more files in your database than the global symbol MXFILS specifies.
You will find MXFILS defined in cdata.h.

Error 18: no schema file specified

You did not specify a schema file on the command line when you ran the schema program.

Error 19: no such schema file

The schema file that you specified on the command line when you ran the schema program does not exist.

APPLICATION SOFTWARE ARCHITECTURE

Now that you have designed a database and built its schema, you are ready for some applications programs to process some data. You will now design a program that prints invoices for clients from the data in the CLIENTS file of the CBS database. This program will use some of the functions of the Cdata DML, which haven't been covered yet because they are described later in this chapter. This example is meant to show you how your programs will incorporate the schema and link with the Cdata functions.

Listing 6.14 is the invoice.c program, an example of a program that uses the Cdata DDL and DML.

Listing 6.14 invoice.c

```
1| /* ----- invoice.c ----- */
2|
3| /*
4|  * produce invoices from the clients file
5|  */
6|
7| #include <stdio.h>
8| #include <stdlib.h>
```

continued...

..from previous page (Listing 6.14)

```

9| #include "cbs.h"
10|
11| struct clients cl;
12|
13| void main(void)
14| {
15|     static DBFILE fl[] = {CLIENTS, -1};
16|
17|     db_open("", fl);
18|     while (TRUE)    {
19|         if (next_rcd(CLIENTS, 1, &cl) == ERROR)
20|             break;
21|         printf("\n\nInvoice for Services Rendered\n");
22|         printf("\n%s", cl.client_name);
23|         printf("\n%s", cl.address);
24|         printf("\n%s, %s %s", cl.city, cl.state, cl.zip);
25|         printf("\n\nAmount Due: $%ld.%02ld\n",
26|                atol(cl.amt_due) / 100,
27|                atol(cl.amt_due) % 100);
28|     }
29|     db_cls();
30| }
```

To run the invoice program, type its name—invoice. Since the invoice data is sent to stdout, you must redirect it to the printer as shown here:

```
A>invoice >prm
```

For the program to do anything, you must have some data in the CLIENTS file. Later, after you have loaded data and posted transactions, you will integrate this program into a complete billing system. But for now, since it is small, you can use it to see how to connect to the Cdata DDL.

The #include statements at the top of the listing connect you to the DDL for the CBS database. The first one, stdio.h, is a standard C header file. The next header file is cdata.h, which was explained earlier in this chapter.

The file cbs.h is the schema for the Consultant's Billing System. You will always include its counterpart in a program that uses Cdata. It defines the structures that describe each of the records in the database. This file is generated by schema.c, the schema compiler.

The structure of type **clients** is a part of the CBS schema in cbs.h. You will have a similar structure for each file in your database. The schema defines the structures. You must declare variables of the structures to reserve memory for buffers.

Note the uses of the CLIENTS identifier in cbs.h and invoice.c. The schema defines CLIENTS as an enumerated data type value that is associated with to the CLIENTS file. The Cdata functions will use the integer value of CLIENTS to subscript into the arrays that describe the CLIENTS file.

The **printf** statements use members of the structure named **cl** to address data elements in the record buffer. These names come from the data element dictionary and are in lowercase, which will distinguish them from the same names in uppercase that define the data element enumerated data type integer values.

The calls to **db_open**, **next_rcd**, and **db_cls** represent the Data Manipulation Language interface between the application program and the Cdata database manager.

After you learn all the Cdata functions and begin coding, you will see that this little program has everything needed to connect an application program to the schema. If you include the proper files and link the program with the Cdata library, you will have built a fully functioning database management system into your own software.

CDATA FILE FORMATS

This discussion covers the format and architecture of files and indexes in a database built with the Cdata functions. You do not need an understanding of these internal formats to use Cdata, but it can help you in deciding whether to use this approach for a particular application. If you want to integrate the data with other software, you will need to know what it looks like.

Data Files

Each database file described in the Cdata DDL has a corresponding disk file. The file name comes from the first eight characters of the database file name as expressed in the DDL #file directive. The file extension is always .DAT. With this naming convention, you can see how the CLIENTS file in the CBS database gets its name, CLIENTS.DAT.

A file consists of a header record followed by a series of fixed-length data records. The header record is the same length and format for all files, but the data record length depends on the data element composition of the file.

Following is the format of the header record:

```
struct fhdr {           /* header on each file          */
    RPTR first_record; /* first available deleted record */
    RPTR next_record;  /* next available record position */
    int record_length; /* length of record             */
};
```

A `typedef` in the header file `cdata.h` equates the variable type `RPTR` to either `unsigned` or `long`. Which type you use will depend on the number of records you can support in a file. An `unsigned` `RPTR` will support 65536 records, and a `long` `RPTR` will support 2^{32} records. The length of the file header record will depend on the `RPTR` definition. For an `unsigned` `RPTR`, the record is 6 characters long. For a `long` `RPTR`, the record is 10 characters long.

Two variables in the file header record contain the file's current record count and its record length. A third variable controls the file's reusable deleted record space. Deleting a record adds the space it occupies to a reusable record linked list. The `RPTR` variable `first_record` points to the first of these spaces. The records themselves are marked with a delete flag, and each one points to the next record in the list. The delete mark occupies the first `RPTR` space in the record and is set to the value `-1`. The list pointer in the record uses the second `RPTR` space in the record and contains the record number of the next available deleted record. Records are added to and taken from the beginning of the list. Record numbers are relative to one.

Reusable deleted record space eliminates the need to pack or compress the data file to clean up the deleted records.

A data record in a file is equal in length to the sum of the lengths of the data elements plus one for each data element for the null string terminator.

Cdata stores data values in files as null-terminated ASCII strings, regardless of the data types. Numbers may or may not have leading zeros. Currency fields are stored without the decimal point. Dates have six characters in day, month, and year format.

Cdata stores new records in the first available record space from the deleted record list or at the end of the file. Initially, the file contains only the file header record; the physical file size grows as Cdata adds records.

Index Files

Cdata supports the relational data model with inverted indexes into data files. The inverted index processes use B-tree algorithms.

The B-tree is an index structure that R. Bayer and E. McCreight developed in 1970. It is a balanced tree of key values used to locate the data file record that matches a specified key argument. The tree is a hierarchy of nodes where each node contains from one to a fixed number of keys.

A B-tree consists of a root node and two or more lower nodes. If the total number of keys in the tree is equal to or less than the number that a node can contain, then only the root node exists. When that number exceeds the capacity of a node, the root node splits into two lower nodes, retaining the key that is logically between the key values of the two new nodes. Higher nodes are parents of the lower nodes. Nodes store keys in key value sequence. When the tree has multiple levels, each key in a parent node points to the lower node that contains keys greater than the parent key and less than the next adjacent key in the parent. The nodes at the lowest level are called leaves. The keys in a leaf node point to the file records that match the indexed values. Because values occur at all levels in the tree, the first key in a leaf is preceded by a pointer to the record of a key value from a higher node.

Figure 6-3 is an example of a B-tree that uses the letters of the alphabet as keys to locate matching words from the phonetic alphabet used by airplane pilots. The letters are analogous to the data values for key data elements in a file; the list of phonetic alphabet words is similar to the file records that the B-tree indexes.

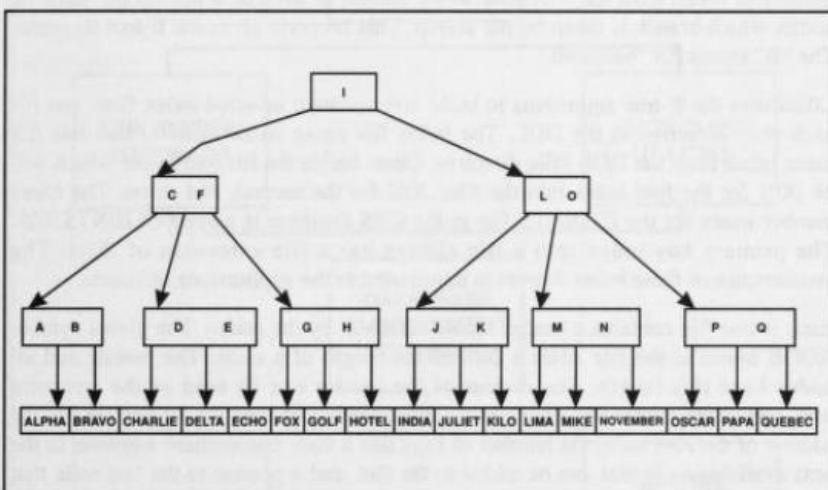


Figure 6-3. The B-Tree.

When a node becomes overpopulated by key value insertions, the insertion algorithm splits it into two nodes and inserts the middle key value into the parent node of the original presplit node. If the root node splits, a new root node is grown as the parent of the split nodes.

When a node becomes underpopulated as the result of key value deletions, it combines with an adjacent node. The key value that is between the two nodes and that is in the common parent node of the two is moved into the combined node and deleted from the parent node. If the deleted key is the last key in the root node, then the old root node is deleted, and the newly combined node becomes the root node.

In the example in Figure 6-3, the root node has one key value, and the other nodes each have two keys. This example makes for a simple illustration of the structure,

but, in actual practice, a typical B-tree will have as many keys in a node as are practical. The arrows in the figure represent actual record pointers that point to lower nodes in the tree or, in the leaf nodes, point to the records in the file being indexed.

The B-tree algorithms provide that the tree always remains in balance; that is, the number of levels from the root node to the bottom of the tree is always the same no matter which branch is taken by the search. This property gives the B-tree its name. The "B" stands for "balanced."

Cdata uses the B-tree algorithms to build and maintain inverted index files, one for each #key directive in the DDL. The index file name is the same as the data file name taken from the DDL #file directive. Cdata builds the file extension, which will be .X01 for the first index into the file, .X02 for the second, and so on. The client number index for the CLIENTS file in the CBS database is named CLIENTS.X01. The primary key index into a file always has a file extension of .X01. The maintenance of these index B-trees is transparent to the applications software.

Each B-tree file contains a header record followed by the nodes. The global symbol NODE found in the file cdata.h defines the length of a node. The header and all nodes have this length. The format of the header can be seen as the structure **tree_hdr** in the source file btree.c later in this chapter. It contains the key length, the address of the root node, the number of keys that a node can contain, a pointer to the next available node that can be added to the file, and a pointer to the last node that was deleted from the tree.

The format of the nodes is described in the structure **treenode**, also in btree.c. Each node contains a flag that identifies the node as a leaf or a nonleaf, pointers to the parent and left and right sibling nodes, and the node/file pointers and key values.

CDATA SYSTEM ARCHITECTURE

Cdata manages a database that consists of data files and B-tree index files. Figure 6-4 shows the layers of software that combine to provide this support.

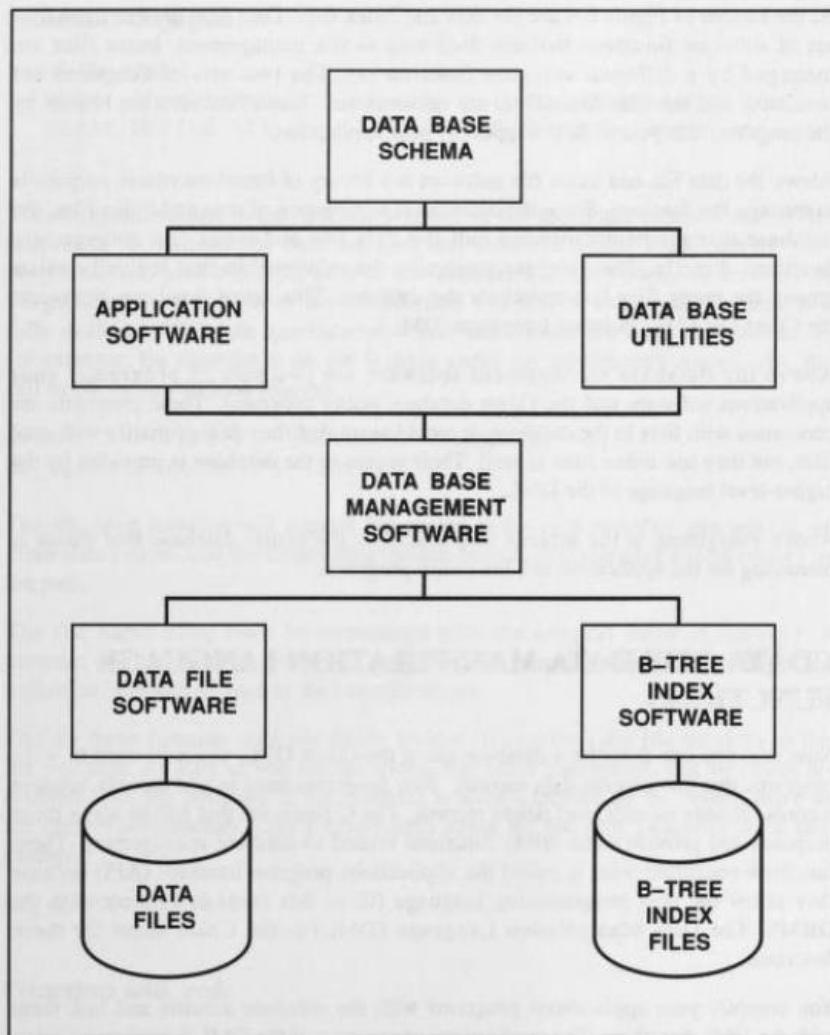


Figure 6-4. The Cdata System Architecture.

At the bottom of Figure 6-4 are the data and index files. Data files are managed by a set of software functions that are dedicated to file management. Index files are managed by a different software function set. The two sets of functions are unrelated, and the files themselves are unconnected. These functions are unseen by the programs that you write in support of your application.

Above the data file and index file software is a library of functions whose purpose is to manage the database. Since the database is a collection of data and index files, the database management functions call the data file and index file management functions directly. The database maintains the relationship that logically exists among the many files that constitute the database. This set of functions represents the Cdata Data Manipulation Language (DML).

Above the database management software are two sets of programs: your applications software and the Cdata database utility programs. These programs are concerned with files in the database. It would seem that they deal primarily with data files, but they use index files as well. Their access to the database is provided by the higher-level language of the DML.

Above everything is the schema that describes the actual database that Cdata is managing for the application and the utility programs.

CDATA: THE DATA MANIPULATION LANGUAGE FUNCTIONS

Now that you can describe a database using the Cdata DDL, you will want to write programs that can process data records. Your programs need to add records, retrieve records, change records, and delete records. The C functions that follow serve those purposes and provide some utility functions related to database management. These functions constitute what is called the applications program interface (API) because they allow the host programming language (C, in this case) to connect with the DBMS. The Data Manipulation Language (DML) is the Cdata name for these functions.

You compile your applications programs with the database schema and link them with the DML functions. The applications programs call the DML functions by using the mnemonic file and data element names of the schema. The DML functions operate on the files as described by the schema.

Function db_open:

```
void db_open(
    const char *path,      /* DOS path to database      */
    const DBFILE *fl       /* list of files to open    */
)
```

The **db_open** function is the first one you call to use the Cdata function toolset. Its purpose is to initialize the database files and their index files for access by your programs. You pass it a DOS path to the database and the address of an array of integers representing a list of the file numbers you will be accessing. The path is a fully qualified DOS path specification which can include the drive designation. If, for example, the database is on the B drive under the subdirectory named \cbs, the call to **db_open** will look like this:

```
static int list[] = {CLIENTS, PROJECTS, -1};
db_open("b:\\cbs\\", list);
```

The **db_open** function will append file names to the path specifier you send it, so make sure you include the terminating double-backslash if you pass a subdirectory in the path.

The file name array must be terminated with the integral value -1. Usually, a program will initialize its array with the global file names defined in the Cdata data definition language as seen in the example above.

The **db_open** function does not return a value. It expects valid file numbers in the list you pass. As long as you use the global file name convention, you will have no problem with file numbers. If the database is always located on the default drive in the current subdirectory, pass a zero-length string for the path parameter as in this example:

```
db_open("", list);
```

Function add_rcd:

```
int add_rcd(
    DBFILE f, /* file number*/
    void *bf   /* buffer containing record*/
)
```

Use this function to add a record to a file. Pass it the file number and a pointer to the buffer containing the new record's data. The **add_rcd** function will return OK if it finds no errors. If it returns ERROR, the global variable **errno** will contain a code describing the error. One error code you can get is D_DUPL, which is defined in cdata.h. This code says that the record contains a primary key value that is assigned to another record. The code D_NF says that one of the data elements in the record is the primary key index data element for another file, and the other file has no record keyed on the value in the record that you are adding.

Function find_rcd:

```
int find_rcd(
    DBFILE f,      /* file number          */
    int k,         /* key number          */
    char *key,     /* key value           */
    void *bf       /* buffer for record */
)
```

This function is the first of several that you use to retrieve records from the database. The file number is required—use one of the globally defined file names. The key number is an integer that specifies which of the indexes you will use. If you are searching the primary index, then pass the value 1 as the key number parameter. The number of the key is the relative position of the key in the schema definition array. The second key is number 2; the third is number 3, and so on. The character pointer named **key** points to the key value that is the argument to the search. The **bf** pointer points to the buffer in the caller's space where the retrieval will store the data record if it finds one.

The function will return OK if it finds a record that matches the key value in the key data element. If the function cannot find a record, it returns ERROR with the code D_NF in **errno**.

Function verify_rcd:

```
int verify_rcd(
    DBFILE f,      /* file number          */
    int k,         /* key number          */
    char *key     /* key value           */
)
```

This function is similar to **find_rcd** except that it does not retrieve a record. Its purpose is to verify that a particular record exists. You use this function when you want to know if a record is in the database, but you do not need the record's contents. You might use this function to validate the presence of an item needed by your processing. Its calling parameters are the same as for **find_rcd** except that you do not provide a record buffer.

If the record exists, the function returns OK; otherwise, it returns ERROR, and D_NF is in **errno**.

Function first_rcd:

```
int first_rcd(
    DBFILE f,      /* file number          */
    int k,         /* key number           */
    void *bf       /* buffer for record   */
)
```

You use the **first_rcd** function to retrieve the first record from a file based on the sequence of the index specified by the key number. You pass the file number, the key number, and a pointer to the buffer where the function should write the record that it retrieves.

If the function returns OK, it located a record and placed it in the buffer. If the function returns ERROR, no records exist in the file, and the value D_EOF (end of file) is in **errno**.

Function last_rcd:

```
int last_rcd(
    DBFILE f,      /* file number          */
    int k,         /* key number           */
    void *bf       /* buffer for record   */
)
```

The **last_rcd** function is the reverse of **first_rcd**. You use it to retrieve the last record from a file based on the sequence of the index specified by the key number. As with **first_rcd**, you pass the file number, the key number, and a pointer to the buffer where the function is to write the record it retrieves.

If the function returns OK, it located a record and placed it in the buffer. If the function returns ERROR, no records exist in the file, and the value D_BOF (beginning of file) is in `errno`.

Function next_rcd:

```
int next_rcd(
    DBFILE f,      /* file number          */
    int k,         /* key number           */
    void *bf       /* buffer for record   */
)
```

You use the `next_rcd` function to retrieve the next record from a file based on the sequence of the index specified by the key number parameter. If no prior access has been made to the file through the same index, then this function behaves exactly like `first_rcd`. You pass the file number, the key number, and a pointer to the buffer where the function is to write the record. If the function returns OK, it located a record and placed it in the buffer.

Successive calls to `next_rcd` will deliver the records in the ascending sequence of the index.

If the function returns ERROR, no records exist in the file past the current record in the chosen sequence, and the value D_EOF (end of file) is in `errno`.

Function prev_rcd:

```
int prev_rcd(
    DBFILE f,      /* file number          */
    int k,         /* key number           */
    void *bf       /* buffer for record   */
)
```

The `prev_rcd` function is the opposite of `next_rcd`, and you use it to retrieve the previous record from a file based on the sequence of the index specified by the key number parameter. If no prior access has been made to the file through the same index, then this function behaves exactly like `last_rcd`. You pass the file number, the key number, and a pointer to the buffer where the function is to write the record. If the function returns OK, it located a record and wrote it in the buffer.

Successive calls to **prev_rcd** will deliver the records in the descending sequence of the index.

If the function returns ERROR, no prior records exist in the file, and the value D_BOF (beginning of file) is in **errno**.

Function rtn_rcd:

```
int rtn_rcd(
    DBFILE f,      /* file number          */
    void *bf       /* buffer for record   */
)
```

Use this function to return to the database a record that was previously retrieved by one of the functions **find_rcd**, **first_rcd**, **last_rcd**, **next_rcd**, **prev_rcd**, and **seqrcd**. Pass it the file number and a pointer to the buffer containing the record's new data.

If **rtn_rcd** finds no errors, it will return OK. If it returns ERROR, the global variable **errno** will contain a code that identifies the error. One possible error code is D_DUPL (duplicate key). As it does in **add_rcd**, this code says that the record contains a primary key value that is already assigned to another record in the file. This situation can occur if you change the value of the primary key data element to a value already in use. It is also possible to receive the error code D_PRIOR (no prior record), which will happen if you have not previously retrieved a record using one of the retrieval functions. The error code D_NF may also appear if one of the data elements in the record is the primary key index data element for another file; the other file has no record keyed on the value that is in the new record.

Function del_rcd:

```
int del_rcd(
    DBFILE f      /* file number          */
)
```

You will use this function to delete a record that was previously retrieved by one of the functions **find_rcd**, **first_rcd**, **last_rcd**, **next_rcd**, **prev_rcd**, and **seqrcd**. Pass it the file number.

If **del_rcd** finds no errors, it will return OK. If it returns ERROR, the global variable **errno** will contain a code that identifies the error. One possible error code is D_PRIOR (no prior record), which will happen if you have not previously retrieved a record using one of the retrieval functions.

Function curr_rcd:

```
int curr_rcd(
    DBFILE f,      /* file number          */
    int k,         /* key number           */
    void *bf       /* buffer for record   */
)
```

This function returns the current record pointed to by the specified key for the specified file. You could use this function after a call to **find_rcd** returned ERROR to read the next record in the index above the key value used in the **find_rcd** call. If **curr_rcd** returns ERROR, the file is empty, the **find_rcd** key pointed past the last record in the index's collating sequence, or the call was not preceded by a call to **find_rcd**, **first_rcd**, **last_rcd**, **next_rcd**, or **prev_rcd** to position the index's pointers.

Function seqrcd:

```
int seqrcd(
    DBFILE f,      /* file number          */
    void *bf       /* buffer for record   */
)
```

This function retrieves records from a file in their physical sequence in the file. Pass it the file number and a pointer to the record buffer.

The function bypasses deleted records and returns OK if it finds a record. If it finds no record, it returns ERROR, and D_EOF is in **errno**.

Function db_cls:

```
db_cls()
```

Call **db_cls** to close all database files, flush the buffers, and close the index files. Do not forget to call this function. If you leave it out, any changed data will be unreliable, and the indexes will be locked.

Function dberror:

```
dberror()
```

This function is a utility you can call whenever any of the Cdata functions discussed in this chapter returns the value ERROR. dberror displays a message indicating the cause of the error. The message displayed is based on the value in the global variable **errno**. If the error is a programming violation of the rules of Cdata, dberror will abort the program; otherwise, it will return to the caller. This function provides a convenient way to let Cdata handle your error messages.

Function rcd_fill:

```
void rcd_fill(
    const void *s,          /* source record buffer      */
    void *d,                /* destination record buffer */
    const ELEMENT *slist,   /* source data element list */
    const ELEMENT *dlist   /* destination data element list */
)
```

This function moves groups of data elements from one record to another. Only the data items with the same names in both records names are moved. Data elements that are in the source record and not in the destination are not moved, and data items that are in the destination record but not in the source are not disturbed.

Cdata defines records as sequences of data elements; data element names are integer values of an enumerated data type, and records are represented by arrays of those data element integers. Given two record descriptions and two buffer addresses, the function uses the data element lengths from the dictionary to find the relative locations of common data elements in both buffers; then, it moves the data values from the source record to the data element positions in the destination record.

Suppose you are retrieving records from the PROJECTS file in the CBS database and your retrieval does not require all the data elements from the record. Only the project name and the amount expended are required. It is possible to describe a record with an array of data element name integers and then allow the functions of Cdata to manage that record's format for you. The code might look like this:

```
static int rlist[]={PROJECT_NAME, AMT_EXPENDED, 0};  
char rcd [36];  
struct projects pr;  
  
. . .  
rcd_fill(&pr, rcd, file_ele[PROJECTS], rlist);
```

Assume that at the time of the **rcd_fill** call you have retrieved a data record from the PROJECTS file into the structure named **pr**. You pass the addresses of the two lists and the addresses of the two buffers to the **rcd_fill** function. (One list, **rlist**, is special to this process; the other, **file_ele[PROJECTS]**, is a part of the database schema.) When the function returns, the two data elements from the file are in the buffer named **rcd**.

Function **epos**:

```
int epos(  
    ELEMENT el,           /* data element */  
    const ELEMENT *list   /* element list */  
)
```

This function returns the relative character position of a data element in a buffer that has its format described by the data element list. The list is an array of integers that represent data elements as defined by the Cdata schema. If the data element integer passed as the first parameter is not in the list, or is zero, then the function returns the length of a buffer required to hold the data elements in the list.

You can use this function to improve the example just shown by using **epos** to compute the size of the extracted record:

```

static int rlist[]={PROJECT_NAME, AMT_EXPENDED, 0};
char *rcd, malloc();
struct projects pr;
.
rcd = malloc(epos(0, rlist));
.
rcd_fill(&pr, rcd, file_ele[PROJECTS], rlist);

```

The **epos** function returns the length required for a buffer that can contain a record as described by the data element list named **rlist**. You can use the function as the parameter for the standard C memory allocation function, **malloc**. When you have changed the list or a data element length, modification of the buffer length is unnecessary. In the earlier example, **rcd** is a character array whose length must match that of the extracted record format. In this example, **rcd** is a character pointer that points to a space equal in length to the computed length of the extracted record.

Function rlen:

```

int rlen(
    DBFILE f      /* file number */
)

```

The **rlen** function returns the length of the database record for the file specified in the parameter. Use this function wherever you need to allocate or move a buffer where the buffer length is not known at compile time.

Function init_rcd:

```

init_rcd(
    DBFILE f,      /* file number           */
    void *bf       /* buffer for record   */
)

```

Use this function when you want to initialize a database file's record buffer to empty values. It sets each data element field in the buffer to a null-terminated string of spaces.

Function clrrcd:

```
void clrrcd(
    void *bf,           /* buffer */
    const ELEMENT *els /* data element list */
)
```

The **clrrcd** function is similar to **init_red** in that it sets a buffer to null-terminated strings of spaces. But **clrrcd** has more general application. Rather than operating on database file records as defined in the schema, **clrrcd** will initialize a buffer by using the data element list you pass to it. This permits you to construct working buffers made up of data elements where the format of the buffer does not necessarily conform to the format of one of the files.

CDATA SOURCE LISTINGS

The Cdata DDL and DML are now complete. You have the tools necessary to design, define, and build a relational database. Immediately following are Listings 6.15 through 6.19. These listings contain the source code for the Cdata DML functions and supporting software. The functions of the DML are included along with several low-level file and index management functions. These functions, when linked with your code, represent the Cheap Database Management System (Cdata).

To complete the package, Chapter 7 presents a set of utility functions and programs that support the Cdata database environment. These programs include initialization utilities, a query and data entry program, a general purpose report program, and a program to regenerate index files from data already in a data file.

Database Manager (**cdata.c**)

Cdata.c, Listing 6.15, is the source file containing the functions described in this chapter that your applications programs will use for database storage and retrievals. It also contains the utility functions described above. Besides the functions already described, **cdata.c** includes several index management functions that are called from within the database functions to manage the maintenance and retrieval of index key values in the B-tree index files. These index managers react to the schema key specifications whenever records are to be retrieved, stored, or deleted by the database management functions.

Listing 6.15 cdata.c

```

1| /* ----- cdata.c ----- */
2|
3| #include <stdio.h>
4| #include <errno.h>
5| #include <stdlib.h>
6| #include <string.h>
7| #include "cdata.h"
8| #include "datafile.h"
9| #include "btree.h"
10| #include "keys.h"
11|
12| void (*database_message)(void);
13| static void init_index(const char *, const DBFILE);
14| static void cls_index(DBFILE);
15| static void del_indexes(DBFILE, RPTR);
16| static int relate_rcd(DBFILE, void *);
17| static int data_in(char *);
18| static int getrcd(DBFILE, RPTR, void *);
19| static int rel_rcd(DBFILE, RPTR, void *);
20|
21| static int db_opened = FALSE; /* data base opened indicator */
22| static int curr_fd [MXFILS]; /* current file descriptor */
23| static char * bfs [MXFILS]; /* file i/o buffers */
24| static int bffd [MXFILS] [MXINDEX];
25| static char dbpath [64];
26|
27| RPTR curr_a [MXFILS]; /* current record file address */
28| /* - macro to compute tree number from file and key number - */
29| #define treeno(f,k) (bfd[f][(k)-1])
30|
31| /* ----- open the data base ----- */
32| void db_open(const char *path, const DBFILE *fl)
33| {
34|     char fnm [64];
35|     int i;
36|
37|     if (!db_opened) {
38|         for (i = 0; i < MXFILS; i++)

```

continued...

...from previous page (Listing 6.15)

```

39:         curr_fd [i] = -1;
40:         db_opened = TRUE;
41:     }
42:     strcpy(dbpath, path);
43:     while (*fl != -1) {
44:         sprintf(fnm, "%s%.8s.dat", path, dbfiles [*fl]);
45:         curr_fd [*fl] = file_open(fnm);
46:         init_index(path, *fl);
47:         if ((bfs [*fl] = malloc(rlen(*fl))) == NULL) {
48:             errno = D_OM;
49:             derror();
50:         }
51:         fl++;
52:     }
53: }
54:
55: /* ----- add a record to a file ----- */
56: int add_rcd(DBFILE f, void *bf)
57: {
58:     RPTR ad;
59:     int rtn;
60:
61:     if ((rtn = relate_rcd(f, bf)) != ERROR) {
62:         ad = new_record(curr_fd [f], bf);
63:         if ((rtn = add_indexes(f, bf, ad)) == ERROR) {
64:             errno = D_DUPL;
65:             delete_record(curr_fd [f], ad);
66:         }
67:     }
68:     return rtn;
69: }
70:
71: /* ----- find a record in a file ----- */
72: int find_rcd(DBFILE f, int k, char *key, void *bf)
73: {
74:     RPTR ad;
75:
76:     if ((ad = locate(treeno(f,k), key)) == 0) {

```

continued...

...from previous page (Listing 6.15)

```

77:         errno = D_NF;
78:         return ERROR;
79:     }
80:     return getrcd(f, ad, bf);
81: }
82:
83: /* ----- verify that a record is in a file ----- */
84: int verify_rcd(DBFILE f, int k, char *key)
85: {
86:     if (locate(treeno(f,k), key) == 0) {
87:         errno = D_NF;
88:         return ERROR;
89:     }
90:     return OK;
91: }
92:
93: /* ----- find the first indexed record in a file ----- */
94: int first_rcd(DBFILE f, int k, void *bf)
95: {
96:     RPTR ad;
97:
98:     if ((ad = firstkey(treeno(f,k))) == 0) {
99:         errno = D_EOF;
100:        return ERROR;
101:    }
102:    return getrcd(f, ad, bf);
103: }
104:
105: /* ----- find the last indexed record in a file ----- */
106: int last_rcd(DBFILE f, int k, void *bf)
107: {
108:     RPTR ad;
109:
110:    if ((ad = lastkey(treeno(f,k))) == 0) {
111:        errno = D_BOF;
112:        return ERROR;
113:    }
114:    return getrcd(f, ad, bf);

```

continued...

...from previous page (Listing 6.15)

```

115| }
116|
117| /* ----- find the next record in a file ----- */
118| int next_rcd(DBFILE f, int k, void *bf)
119| {
120|     RPTR ad;
121|
122|     if ((ad = nextkey(treeno(f, k))) == 0) {
123|         errno = D_EOF;
124|         return ERROR;
125|     }
126|     return getrcd(f, ad, bf);
127| }
128|
129| /* ----- find the previous record in a file ----- */
130| int prev_rcd(DBFILE f, int k, void *bf)
131| {
132|     RPTR ad;
133|
134|     if ((ad = prevkey(treeno(f,k))) == 0) {
135|         errno = D_BOF;
136|         return ERROR;
137|     }
138|     return getrcd(f, ad, bf);
139| }
140|
141| /* ----- return the current record to the data base ----- */
142| int rtn_rcd(DBFILE f, void *bf)
143| {
144|     int rtn;
145|
146|     if (curr_a [f] == 0) {
147|         errno = D_PRIOR;
148|         return ERROR;
149|     }
150|     if ((rtn = relate_rcd(f, bf)) != ERROR) {
151|         del_indexes(f, curr_a [f]);
152|         if ((rtn = add_indexes(f, bf, curr_a [f])) == OK)

```

continued...

..from previous page (Listing 6.15)

```

153         put_record(curr_fd [f], curr_a [f], bf);
154     else
155         errno = D_DUPL;
156     }
157     return rtn;
158 }
159
160 /* ----- delete the current record from the file ----- */
161 int del_rcd(DBFILE f)
162 {
163     if (curr_a [f]) {
164         del_indexes(f, curr_a [f]);
165         delete_record(curr_fd [f], curr_a [f]);
166         curr_a [f] = 0;
167         return OK;
168     }
169     errno = D_PRIOR;
170     return ERROR;
171 }
172
173 /* ----- find the current record in a file ----- */
174 int curr_rcd(DBFILE f, int k, void *bf)
175 {
176     RPTR ad;
177     if ((ad = currkey(treeno(f,k))) == 0)    {
178         errno = D_NF;
179         return ERROR;
180     }
181     getrcd(f, ad, bf);
182     return OK;
183 }
184
185 /* ----- get the next physical
186             sequential record from the file ----- */
187 int seqrcd(DBFILE f, void *bf)
188 {
189     RPTR ad;
190     int rtn;

```

continued...

..from previous page (Listing 6.15)

```
191;
192:     do  {
193:         ad = ++curr_a [f];
194:         if ((rtn = (rel_rcd(f,ad,bf)))==ERROR && errno=D_NF)
195:             break;
196:     }    while (errno == D_NF);
197:     return rtn;
198: }
199;
200: /* ----- close the data base ----- */
201: void db_cls(void)
202: {
203:     DBFILE f;
204:
205:     for (f = 0; f < MXFILS; f++)
206:         if (curr_fd [f] != -1) {
207:             file_close(curr_fd [f]);
208:             cls_index(f);
209:             free(bfs[f]);
210:             curr_fd [f] = -1;
211:         }
212:     db_opened = FALSE;
213: }
214;
215: /* ----- data base error routine ----- */
216: void dberror(void)
217: {
218:     static int fat [] = {0,1,0,0,0,1,1,1};
219:
220:     if (database_message)
221:         (*database_message)();
222:     if (fat [errno-1])
223:         exit(1);
224: }
225;
226: /* ----- compute file record length ----- */
227: int rlen(DBFILE f)
228: {
```

continued...

...from previous page (Listing 6.15)

```

229|     return epos(0, file_ele[f]);
230| }
231|
232| /* ----- initialize a file record buffer ----- */
233| void init_rcd(DBFILE f, void *bf)
234| {
235|     clrrcd(bf, file_ele[f]);
236| }
237|
238| /* ----- set a generic record buffer to blanks ----- */
239| void clrrcd(void *bf, const ELEMENT *els)
240| {
241|     int ln, i = 0, el;
242|     char *rb;
243|
244|     while ((*els + i)) {
245|         el = *(els + i);
246|         rb = (char *) bf + epos(el, els);
247|         ln = ellen[el - 1];
248|         while (ln--)
249|             *rb++ = ' ';
250|         *rb = '\0';
251|         i++;
252|     }
253| }
254|
255|
256| /* ----- move data from one record to another ----- */
257| void rcd_fill(const void *s, void *d,
258|               const ELEMENT *slist,
259|               const ELEMENT *dlist)
260| {
261|     const int *s1;
262|     const int *d1;
263|
264|     s1 = slist;
265|     while (*s1) {
266|         d1 = dlist;

```

continued...

...from previous page (Listing 6.15)

```

267     while (*d1) {
268         if (*s1 == *d1)
269             strcpy((char *)d+epos(*d1,dlist),
270                   (char *)s+epos(*s1,slist));
271         d1++;
272     }
273     s1++;
274   }
275 }
276
277 /* ----- compute relative position of
278      a data element within a record -----*/
279 int epos(ELEMENT el, const ELEMENT *list)
280 {
281     int len = 0;
282     const ELEMENT *els = list;
283
284     while (el != *els)
285         len += ellen [(*els++)-1] + 1;
286     return len;
287 }
288
289 /* ----- index management functions ----- */
290 /* ---- initialize the indices for a file ---- */
291 static void init_index(const char *path, const DBFILE f)
292 {
293     char xname [64];
294     int x = 0;
295
296     while (*index_ele [f] + x)) {
297         sprintf(xname, "%s%.8s.x%02d", path, dbfiles[f], x+1);
298         if ((bfd [f] [x++] = btree_init(xname)) == ERROR) {
299             printf("\n%s", xname);
300             errno = D_INDXC;
301             dberror();
302         }
303     }
304 }
```

continued...

...from previous page (Listing 6.15)

```

305|
306| /* ---- build the indices for a file ---- */
307| void build_index(char *path, DBFILE f)
308|
309| {
310|     char xname [64];
311|     int x = 0, x1;
312|     int len;
313|
314|     while (*(*index_ele [f] + x)) {
315|         sprintf(xname, "%s%.8s.x%02d", path, dbfiles[f], x+1);
316|         len = 0;
317|         x1 = 0;
318|         while (*(*(*index_ele [f] + x) + x1))
319|             len += ellen [*(*(*index_ele [f] + x) + (x1++))-1];
320|         build_b(xname, len);
321|         x++;
322|     }
323|
324| /* ----- close the indices for a file ----- */
325| static void cls_index(DBFILE f)
326|
327| {
328|     int x = 0;
329|
330|     while (*(*index_ele [f] + x)) {
331|         if (bfd [f] [x] != ERROR)
332|             btree_close(bfd [f] [x]);
333|         x++;
334|     }
335|
336| /* ---- add index values from a record to the indices ---- */
337| int add_indexes(DBFILE f, void *bf, RPTR ad)
338|
339| {
340|     int x = 0;
341|     int i;
342|     char key [MXKEYLEN];

```

continued...

..from previous page (Listing 6.15)

```

343|     while (*(index_ele [f] + x))    {
344|         *key = '\0';
345|         i = 0;
346|         while(*(*(index_ele [f] + x) + i))
347|             strcat(key,
348|                     (char *) bf +
349|                         epos(*(*(index_ele[f]+x)+(i++)),file_ele [f]));
350|             if (insertkey(bfd [f] [x], key, ad, ix) == ERROR)
351|                 return ERROR;
352|             x++;
353|         }
354|         return OK;
355|     }
356|
357| /* --- delete index values in a record from the indices --- */
358| static void del_indexes(DBFILE f, RPTR ad)
359| {
360|     char *bf;
361|     int x = 0;
362|     int i;
363|     char key [MXKEYLEN];
364|
365|     if ((bf = malloc(rlen(f))) == NULL) {
366|         errno = D_OOM;
367|         dberror();
368|     }
369|     get_record(curr_fd [f], ad, bf);
370|     while (*(index_ele [f] + x))    {
371|         *key = '\0';
372|         i = 0;
373|         while (*(*(index_ele [f] + x) + i))
374|             strcat(key,
375|                     bf +
376|                         epos(*(*(index_ele[f]+x)+(i++)), file_ele [f]));
377|             deletekey(bfd [f] [x++], key, ad);
378|     }
379|     free(bf);
380| }
```

continued...

...from previous page (Listing 6.15)

```

381| /* ---- validate the contents of a record where its file is
382|      related to another file in the data base ----- */
383| static int relate_rcd(DBFILE f, void *bf)
384| {
385|     int fx = 0, mx;
386|     const int *fp;
387|     static int ff[] = {0, -1};
388|     char *cp;
389|
390|     while (dbfiles [fx])    {
391|         if (fx != f && *(*(index_ele [fx]) + 1) == 0)    {
392|             mx = *(*(index_ele [fx]));
393|             fp = (int *) file_ele [f];
394|             while (*fp) {
395|                 if (*fp == mx)    {
396|                     cp = (char *)bf + epos(mx, file_ele [f]);
397|                     if (data_in(cp))    {
398|                         if (curr_fd[fx] == -1)    {
399|                             *ff = fx;
400|                             db_open(dbpath, ff);
401|                         }
402|                         if (verify_rcd(fx, 1, cp) == ERROR)
403|                             return ERROR;
404|                     }
405|                 }
406|                 break;
407|             }
408|             fp++;
409|         }
410|     }
411|     fx++;
412| }
413| return OK;
414| }
415|
416| /* ---- test a string for data. return TRUE if any ---- */
417| static int data_in(char *c)
418| {

```

continued...

..from previous page (Listing 6.15)

```
419:     while (*c == ' ')
420:         c++;
421:     return (*c != '\0');
422: }
423:
424: /* ----- get a record from a file ----- */
425: static int getrcd(DBFILE f, RPTR ad, void *bf)
426: {
427:     get_record(curr_fd [f], ad, bf);
428:     curr_a [f] = ad;
429:     return OK;
430: }
431:
432: extern FHEADER fh [];
433:
434: /* ----- find a record by relative record number ----- */
435: static int rel_rcd(DBFILE f, RPTR ad, void *bf)
436: {
437:     errno = 0;
438:     if (ad >= fh [curr_fd [f]].next_record) {
439:         errno = D_EOF;
440:         return ERROR;
441:     }
442:     getrcd(f, ad, bf);
443:     if (*(int *)bf == -1)    {
444:         errno = D_NF;
445:         return ERROR;
446:     }
447:     return OK;
448: }
```

Refer back to the diagram in Figure 6-4. Cdata.c is in the symbol that is labeled "Database Management Software."

File Manager (datafile.h and datafile.c)

Datafile.h, Listing 6.16, is the header file for the data file management functions. It defines the prototypes for the functions and the **fhdr** structure discussed earlier.

Listing 6.16 datafile.h

```
1| /* ----- datafile.h ----- */
2|
3| /* ----- data file prototypes ----- */
4| void file_create(char *, int);
5| int file_open(char *);
6| void file_close(int);
7| RPTR new_record(int, void *);
8| int get_record(int, RPTR, void *);
9| int put_record(int, RPTR, void *);
10| int delete_record(int, RPTR);
11|
12| /* ----- file header ----- */
13| typedef struct fhdr {
14|     RPTR first_record;
15|     RPTR next_record;
16|     int record_length;
17| } FHEADER;
```

Datafile.c, Listing 6.17, is the source file that contains the data file management functions for the database manager. It includes functions to create data files and to add, change, and delete records in data files.

Listing 6.17 datafile.c

```

1  /* ----- datafile.c ----- */
2
3 #include <stdio.h>
4 #include <errno.h>
5 #include <string.h>
6 #include <stdlib.h>
7 #include "cdata.h"
8 #include "datafile.h"
9
10#define flocate(r,l) ((long)(sizeof(FHEADER)+((r)-1)*(l)))
11
12static FILE *fp[MXFILS];
13FHEADER fh[MXFILS];
14
15/* ----- create a file ----- */
16void file_create(char *name, int len)
17{
18    FILE *fp;
19    FHEADER hd;
20
21    remove(name);
22    fp = fopen(name, "wb");
23    hd.next_record = 1;
24    hd.first_record = 0;
25    hd.record_length = len;
26    fwrite((char *) &hd, sizeof hd, 1, fp);
27    fclose(fp);
28}
29
30/* ----- open a file ----- */
31int file_open(char *name)
32{
33    int fno;
34
35    for (fno = 0; fno < MXFILS; fno++)
36        if (fp[fno] == NULL)
37            break;
38    if (fno == MXFILS)

```

continued...

..from previous page (Listing 6.17)

```

39|         return ERROR;
40|     if ((fp[fno] = fopen(name, "rb+")) == NULL)
41|         return ERROR;
42|     fseek(fp[fno], 0L, SEEK_SET);
43|     fread((char *) &fh[fno], sizeof(FHEADER), 1, fp[fno]);
44|     return fno;
45| }
46|
47| /* ----- close a file ----- */
48| void file_close(int fno)
49| {
50|     fseek(fp[fno], 0L, SEEK_SET);
51|     fwrite((char *) &fh[fno], sizeof(FHEADER), 1, fp[fno]);
52|     fclose(fp[fno]);
53|     fp[fno] = NULL;
54| }
55|
56| /* ----- create a new record ----- */
57| RPTR new_record(int fno, void *bf)
58| {
59|     RPTR rcdno;
60|     FHEADER *c;
61|
62|     if (fh[fno].first_record)    {
63|         rcdno = fh[fno].first_record;
64|         if ((c = malloc(fh[fno].record_length)) == NULL)    {
65|             errno = D_0M;
66|             dberror();
67|         }
68|         get_record(fno, rcdno, c);
69|         fh[fno].first_record = c->next_record;
70|         free(c);
71|     }
72|     else
73|         rcdno = fh[fno].next_record++;
74|     put_record(fno, rcdno, bf);
75|     return rcdno;
76| }
77|

```

continued...

...from previous page (Listing 6.17)

```

78| /* ----- retrieve a record ----- */
79| int get_record(int fno, RPTR rcdno, void *bf)
80{
81    if (rcdno >= fh[fno].next_record)
82        return ERROR;
83    fseek(fp[fno],
84          flocate(rcdno, fh[fno].record_length), SEEK_SET);
85    fread(bf, fh[fno].record_length, 1, fp[fno]);
86    return OK;
87}
88
89| /* ----- rewrite a record ----- */
90| int put_record(int fno, RPTR rcdno, void *bf)
91{
92    if (rcdno > fh[fno].next_record)
93        return ERROR;
94    fseek(fp[fno],
95          flocate(rcdno, fh[fno].record_length), SEEK_SET);
96    fwrite(bf, fh[fno].record_length, 1, fp[fno]);
97    return OK;
98}
99
100| /* ----- delete a record ----- */
101| int delete_record(int fno, RPTR rcdno)
102{
103    FHEADER *bf;
104
105    if (rcdno > fh[fno].next_record)
106        return ERROR;
107    if ((bf = (FHEADER *) 
108         malloc(fh[fno].record_length)) == NULL) {
109        errno = D_OOM;
110        dberror();
111    }
112    memset(bf, '\0', fh[fno].record_length);
113    bf->next_record = fh[fno].first_record;
114    bf->first_record = -1;
115    fh[fno].first_record = rcdno;
116    put_record(fno, rcdno, bf);
117    free(bf);
118    return OK;
119}

```

Refer back to the diagram in Figure 6-4. Datafile.c is in the symbol that is labeled "Data File Software." You might never have a reason to use the functions in datafile.c, but they are described here so that you can better understand your system.

Function file_create:

```
void file_create(
    char *name, /* file name */ 
    int len      /* record length */ 
)
```

This function is used to create a data file and is called from dbinit.c when the files for the database are initialized. The **name** pointer points to the DOS file name of the data file. The **len** integer is the file's record length.

Function file_open:

```
int file_open(
    char *name /* file name */ 
)
```

This function opens an existing data file, one that was previously created by **file_create**, and returns a logical file handle that Cdata uses for subsequent calls to data file functions. If the file does not exist, the function returns ERROR.

Function file_close:

```
void file_close(
    int fp      /* file handle */ 
)
```

This function closes a data file. You pass it the logical file handle that the **file_open** function returned.

Function new_record:

```
RPTR new_record(  
    int fp,          /* file handle */  
    void *bf         /* record buffer */  
)
```

This function adds a record to a data file, writing it in the next available record space in the file. It will write the record either at the end of the file or, if records have been deleted from the file, into a record position that was formerly occupied by a since-deleted record.

The RPTR return value is a **typedef** in cdata.h. It will be either an integer or a long integer, and it will represent the logical record position within the file that was assigned to the record. The RPTR value is relative to one.

Function get_record:

```
int get_record(  
    int fp,          /* file handle */  
    RPTR rcdno,     /* logical record number */  
    void *bf         /* record buffer */  
)
```

This function retrieves a record that was previously stored in the data file. The RPTR **rcdno** parameter is the logical record number where the record is known to be stored. The **bf** pointer points to the buffer into which the record will be read.

Function put_record:

```
int put_record(  
    int fp,          /* file handle */  
    RPTR rcdno,     /* logical record number */  
    void *bf         /* record buffer */  
)
```

This function rewrites a record to the file and is used when the record has been changed. The RPTR **rcdno** parameter is the logical record number where the record is to be stored. The **bf** pointer points to the buffer from which the record will be written.

Function delete_record:

```
int delete_record(
    int fp,          /* file handle           */
    RPTR rcdno      /* logical record number */ )
)
```

This function deletes the record located at logical record position **rcdno**. The record space is placed into a linked list for the file so that a later record addition can reuse the space.

Index Manager (btree.c)

Listing 6.18 is the source file btree.h. It defines the prototypes for the B-tree functions, the structures for the B-tree header record and nodes, and the MXTREES and NODE global variables, which define the maximum number of B-tree files that can be open at one time and the size in bytes of a B-tree node record.

Listing 6.18 btree.h

```
1| /* ----- btree.h ----- */
2|
3| #define MXTREES 20
4| #define NODE 512          /* length of a B-tree node */
5| #define ADR sizeof(RPTR)
6|
7| /* ----- btree prototypes ----- */
8| int btree_init(char *);
9| int btree_close(int);
10| void build_b(char *, int);
```

continued...

...from previous page (Listing 6.18)

```

11| RPTR locate(int, char *);
12| int deletekey(int, char *, RPTR);
13| int insertkey(int, char *, RPTR, int);
14| RPTR nextkey(int);
15| RPTR prevkey(int);
16| RPTR firstkey(int);
17| RPTR lastkey(int);
18| void keyval(int, char *);
19| RPTR currkey(int);
20|
21| /* ----- the btree node structure ----- */
22| typedef struct treenode {
23|     int nonleaf;      /* 0 if leaf, 1 if non-leaf          */
24|     RPTR prtnode;    /* parent node                      */
25|     RPTR lfsib;      /* left sibling node                */
26|     RPTR rtsib;      /* right sibling node               */
27|     int keyct;       /* number of keys                  */
28|     RPTR key0;       /* node # of keys < 1st key this node */
29|     char keyspace [NODE - ((sizeof(int) * 2) + (ADR * 4))];
30|     char spil [MXKEYLEN]; /* for insertion excess */
31| } BTREE;
32|
33| /* ---- the structure of the btree header node ----- */
34| typedef struct treehdr {
35|     RPTR rootnode;    /* root node number           */
36|     int keylength;    /* length of a key            */
37|     int m;           /* max keys/node             */
38|     RPTR rlshed_node; /* next released node        */
39|     RPTR endnode;    /* next unassigned node      */
40|     int locked;      /* if btree is locked        */
41|     RPTR leftmost;   /* left-most node            */
42|     RPTR rightmost;  /* right-most node           */
43| } HEADER;

```

Listing 6.19 is the source file btree.c. This group of functions manages the database B-tree index files. The functions are generic B-tree managers, and the index management functions in database.c use them. They include functions to create and search B-trees, add and delete keys in B-trees, and navigate B-trees in ascending or descending key sequence. The key entries in the B-tree files yield data file record addresses that point to the data file records that match the key values. As such, the B-tree mechanism supports multiple inverted indexes into the relational database flat files.

Listing 6.19 btree.c

```

1| /* ----- btree.c ----- */
2| #include <stdio.h>
3| #include <errno.h>
4| #include <string.h>
5| #include <stdlib.h>
6| #include "cdata.h"
7| #include "btree.h"
8|
9| #define KLEN bheader[trx].keylength
10| #define ENTLN (KLEN+ADR)
11|
12| HEADER bheader[MXTREES];
13| BTREE trnode;
14|
15| static FILE *fp[MXTREES];      /* file pointers to indexes */
16| static RPTR currnode[MXTREES]; /* node number of current key */
17| static int currkno[MXTREES];   /* key number of current key */
18| static int trx;                /* current tree */
19|
20| /* ----- local prototypes ----- */
21| static int btreescan(RPTR *, char *, char **);
22| static int nodescan(char *, char **);
23| static int compare_keys(char *, char *);
24| static RPTR fileaddr(RPTR, char *);
25| static RPTR leaflevel(RPTR *, char **, int *);
26| static void implode(BTREE *, BTREE *);
27| static void redist(BTREE *, BTREE *);
28| static void adopt(void *, int, RPTR);
29| static RPTR nextnode(void);

```

continued...

...from previous page (Listing 6.19)

```

30| static RPTR scannext(RPTR *, char **);
31| static RPTR scanprev(RPTR *, char **);
32| static char *childptr(RPTR, RPTR, BTREE *);
33| static void read_node(RPTR, void *);
34| static void write_node(RPTR, void *);
35| static void bseek(RPTR);
36| static void memerr(void);
37|
38| /* ----- initiate b-tree processing ----- */
39| int btree_init(char *ndx_name)
40{
41    for (trx = 0; trx < MXTREES; trx++)
42        if (fp[trx] == NULL)
43            break;
44    if (trx == MXTREES)
45        return ERROR;
46    if ((fp[trx] = fopen(ndx_name, "rb+")) == NULL)
47        return ERROR;
48    fread(&bheader[trx], sizeof(HEADER), 1, fp[trx]);
49    /* --- if this btree is locked, something is amiss --- */
50    if (bheader[trx].locked) {
51        fclose(fp[trx]);
52        fp[trx] = NULL;
53        return ERROR;
54    }
55    /* ----- lock the btree ----- */
56    bheader[trx].locked = TRUE;
57    fseek(fp[trx], 0L, SEEK_SET);
58    fwrite(&bheader[trx], sizeof(HEADER), 1, fp[trx]);
59    currnode[trx] = 0;
60    currkno[trx] = 0;
61    return trx;
62}
63|
64| /* ----- terminate b tree processing ----- */
65| int btree_close(int tree)
66{
67    if (tree >= MXTREES || fp[tree] == 0)

```

continued...

...from previous page (Listing 6.19)

```

68|         return ERROR;
69|     bheader[tree].locked = FALSE;
70|     fseek(fp[tree], 0L, SEEK_SET);
71|     fwrite(&bheader[tree], sizeof(HEADER), 1, fp[tree]);
72|     fclose(fp[tree]);
73|     fp[tree] = NULL;
74|     return OK;
75| }
76|
77| /* -----Build a new b-tree ----- */
78| void build_b(char *name, int len)
79| {
80|     HEADER *bhdp;
81|     FILE *fp;
82|
83|     if ((bhdp = malloc(NODE)) == NULL)
84|         memerr();
85|     memset(bhdp, '\0', NODE);
86|     bhdp->keylength = len;
87|     bhdp->m = ((NODE-((sizeof(int)*2)+(ADR*4)))/(len+ADR));
88|     bhdp->endnode = 1;
89|     remove(name);
90|     fp = fopen(name, "wb");
91|     fwrite(bhdp, NODE, 1, fp);
92|     fclose(fp);
93|     free(bhdp);
94| }
95|
96| /* ----- Locate key in the b-tree ----- */
97| RPTR Locate(int tree, char *k)
98| {
99|     int i, fnd = FALSE;
100|     RPTR t, ad;
101|     char *a;
102|
103|     trx = tree;
104|     t = bheader[trx].rootnode;

```

continued...

...from previous page (Listing 6.19)

```

105|     if (t) {
106|         read_node(t, &trnode);
107|         fnd = btreescan(&t, k, &a);
108|         ad = leaflevel(&t, &a, &i);
109|         if (i == trnode.keyct + 1) {
110|             i = 0;
111|             t = trnode.rtsib;
112|         }
113|         currnode[trx] = t;
114|         currkno[trx] = i;
115|     }
116|     return fnd ? ad : 0;
117| }
118|
119/* ----- Search tree ----- */
120static int btreescan(RPTR *t, char *k, char **a)
121{
122    int nl;
123    do {
124        if (nodescan(k, a)) {
125            while (compare_keys(*a, k) == FALSE)
126                if (scanprev(t, a) == 0)
127                    break;
128                if (compare_keys(*a, k))
129                    scannext(t, a);
130                return TRUE;
131            }
132            nl = trnode.nonleaf;
133            if (nl) {
134                *t = *((RPTR *) (*a - ADR));
135                read_node(*t, &trnode);
136            }
137        } while (nl);
138    return FALSE;
139}
140

```

continued...

...from previous page (Listing 6.19)

```

141 /* ----- Search node ----- */
142 static int nodescan(char *keyvalue, char **nodeadr)
143 {
144     int i;
145     int result;
146
147     *nodeadr = trnode.keyspace;
148     for (i = 0; i < trnode.keyct; i++) {
149         result = compare_keys(keyvalue, *nodeadr);
150         if (result == FALSE)
151             return TRUE;
152         if (result < 0)
153             return FALSE;
154         *nodeadr += ENTLN;
155     }
156     return FALSE;
157 }
158
159 /* ----- Compare keys ----- */
160 static int compare_keys(char *a, char *b)
161 {
162     int len = KLEN, cm;
163
164     while (len--)
165         if ((cm = (int) *a++ - (int) *b++) != 0)
166             break;
167     return cm;
168 }
169
170 /* ----- Compute current file address ----- */
171 static RPTR fileaddr(RPTR t, char *a)
172 {
173     RPTR cn, ti;
174     int i;
175
176     ti = t;
177     cn = leaflevel(&ti, &a, &i);
178     read_node(t, &trnode);

```

continued...

...from previous page (Listing 6.19)

```

179    return cn;
180 }
181
182 /* ----- Navigate down to leaf level ----- */
183 static RPTR Leaflevel(RPTR *t, char **a, int *p)
184 {
185     if (trnode.nonleaf == FALSE) { /* already at a leaf? */
186         *p = (*a - trnode.keyspace) / ENTLN + 1;
187         return *((RPTR *) (*a + KLEN));
188     }
189     *p = 0;
190     *t = *((RPTR *) (*a + KLEN));
191     read_node(*t, &trnode);
192     *a = trnode.keyspace;
193     while (trnode.nonleaf) {
194         *t = trnode.key0;
195         read_node(*t, &trnode);
196     }
197     return trnode.key0;
198 }
199
200 /* ----- Delete a key ----- */
201 int deletekey(int tree, char *x, RPTR ad)
202 {
203     BTREE *qp, *yp;
204     int rt_len, comb;
205     RPTR p, adr, q, *b, y, z;
206     char *a;
207
208     trx = tree;
209     if (trx >= MXTREES || fp[trx] == 0)
210         return ERROR;
211     p = bheader[trx].rootnode;
212     if (p == 0)
213         return OK;
214     read_node(p, &trnode);
215     if (btreescan(&p, x, &a) == FALSE)

```

continued...

..from previous page (Listing 6.19)

```

216:         return OK;
217:     adr = fileaddr(p, a);
218:     while (adr != ad) {
219:         adr = scannext(&p, &a);
220:         if (compare_keys(a, x))
221:             return OK;
222:     }
223:     if (trnode.nonleaf) {
224:         b = (RPTR *) (a + KLEN);
225:         q = *b;
226:         if ((qp = malloc(NODE)) == NULL)
227:             memerr();
228:         read_node(q, qp);
229:         while (qp->nonleaf) {
230:             q = qp->key0;
231:             read_node(q, qp);
232:         }
233:         /* Move the left-most key from the leaf
234:            to where the deleted key is */
235:         memmove(a, qp->keyspace, KLEN);
236:         write_node(p, &trnode);
237:         p = q;
238:         trnode = *qp;
239:         a = trnode.keyspace;
240:         b = (RPTR *) (a + KLEN);
241:         trnode.key0 = *b;
242:         free(qp);
243:     }
244:     currnode[trx] = p;
245:     currkno[trx] = (a - trnode.keyspace) / ENTLN;
246:     rt_len = (trnode.keyspace + (bheader[trx].m * ENTLN)) - a;
247:     memmove(a, a+ENTLN, rt_len);
248:     memset(a+rt_len, '\0', ENTLN);
249:     trnode.keyct--;
250:     if (currkno[trx] > trnode.keyct) {
251:         if (trnode.rtsib) {
252:             currnode[trx] = trnode.rtsib;

```

continued...

...from previous page (Listing 6.19)

```

253         currkno[trx] = 0;
254     }
255     else
256         currkno[trx]--;
257 }
258 while (trnode.keyct <= bheader[trx].m / 2 &&
259         p != bheader[trx].rootnode) {
260     comb = FALSE;
261     z = trnode.prntnode;
262     if ((yp = malloc(NODE)) == NULL)
263         memerr();
264     if (trnode.rtsib) {
265         y = trnode.rtsib;
266         read_node(y, yp);
267         if (yp->keyct + trnode.keyct <
268             bheader[trx].m && yp->prntnode == z) {
269             comb = TRUE;
270             implode(&trnode, yp);
271         }
272     }
273     if (comb == FALSE && trnode.lfsib) {
274         y = trnode.lfsib;
275         read_node(y, yp);
276         if (yp->prntnode == z) {
277             if (yp->keyct + trnode.keyct <
278                 bheader[trx].m) {
279                 comb = TRUE;
280                 implode(yp, &trnode);
281             }
282             else {
283                 redist(yp, &trnode);
284                 write_node(p, &trnode);
285                 write_node(y, yp);
286                 free(yp);
287                 return OK;
288             }
289         }
}

```

continued...

...from previous page (Listing 6.19)

```

290      }
291      if (comb == FALSE) {
292          y = trnode.rtsib;
293          read_node(y, yp);
294          redist(&trnode, yp);
295          write_node(y, yp);
296          write_node(p, &trnode);
297          free(yp);
298          return OK;
299      }
300      free(yp);
301      p = z;
302      read_node(p, &trnode);
303  }
304  if (trnode.keyct == 0) {
305      bheader[trx].rootnode = trnode.key0;
306      trnode.nonleaf = FALSE;
307      trnode.key0 = 0;
308      trnode.prntnode = bheader[trx].rlsed_node;
309      bheader[trx].rlsed_node = p;
310  }
311  if (bheader[trx].rootnode == 0)
312      bheader[trx].rightmost = bheader[trx].leftmost = 0;
313  write_node(p, &trnode);
314  return OK;
315 }
316
317 /* ----- Combine two sibling nodes. ----- */
318 static void implode(BTREE *left, BTREE *right)
319 {
320     RPTR lf, rt, p;
321     int rt_len, lf_len;
322     char *a;
323     RPTR *b;
324     BTREE *par;
325     RPTR c;
326     char *j;

```

continued...

...from previous page (Listing 6.19)

```

327|
328|     lf = right->lfsib;
329|     rt = left->rtsib;
330|     p = left->prntnode;
331|     if ((par = malloc(NODE)) == NULL)
332|         memerr();
333|     j = childptr(lf, p, par);
334|     /* --- move key from parent to end of left sibling --- */
335|     lf_len = left->keyct * ENTLN;
336|     a = left->keyspace + lf_len;
337|     memmove(a, j, KLEN);
338|     memset(j, '\0', ENTLN);
339|     /* --- move keys from right sibling to left --- */
340|     b = (RPTR *) (a + KLEN);
341|     *b = right->key0;
342|     rt_len = right->keyct * ENTLN;
343|     a = (char *) (b + 1);
344|     memmove(a, right->keyspace, rt_len);
345|     /* --- point lower nodes to their new parent --- */
346|     if (left->nonleaf)
347|         adopt(b, right->keyct + 1, lf);
348|     /* --- if global key pointers -> to the right sibling,
349|          change to -> left --- */
350|     if (currnode[trx] == left->rtsib) {
351|         currnode[trx] = right->lfsib;
352|         currkno[trx] += left->keyct + 1;
353|     }
354|     /* --- update control values in left sibling node --- */
355|     left->keyct += right->keyct + 1;
356|     c = bheader[trx].rlsed_node;
357|     bheader[trx].rlsed_node = left->rtsib;
358|     if (bheader[trx].rightmost == left->rtsib)
359|         bheader[trx].rightmost = right->lfsib;
360|     left->rtsib = right->rtsib;
361|     /* --- point the deleted node's right brother
362|          to this left brother --- */
363|     if (left->rtsib)    {

```

continued...

...from previous page (Listing 6.19)

```

364     read_node(left->rtsib, right);
365     right->lfsib = lf;
366     write_node(left->rtsib, right);
367 }
368 memset(right, '\0', NODE);
369 right->prntnode = c;
370 /* --- remove key from parent node --- */
371 par->keyct--;
372 if (par->keyct == 0)
373     left->prntnode = 0;
374 else {
375     rt_len = par->keyspace + (par->keyct * ENTLN) - j;
376     memmove(j, j+ENTLN, rt_len);
377 }
378 write_node(lf, left);
379 write_node(rt, right);
380 write_node(p, par);
381 free(par);
382 }
383
384 /* ----- Insert key ----- */
385 int insertkey(int tree, char *x, RPTR ad, int unique)
386 {
387     char k[MXKEYLEN + 1], *a;
388     BTREE *yp;
389     BTREE *bp;
390     int nl_flag, rt_len, j;
391     RPTR t, p, sv;
392     RPTR *b;
393     int lshft, rshft;
394
395     trx = tree;
396     if (trx >= MXTREES || fp[trx] == 0)
397         return ERROR;
398     p = 0;
399     sv = 0;
400     nl_flag = 0;

```

continued...

..from previous page (Listing 6.19)

```

401    memmove(k, x, KLEN);
402    t = bheader[trx].rootnode;
403    /* ----- Find insertion point ----- */
404    if (t) {
405        read_node(t, &trnode);
406        if (btreescan(&t, k, &a)) {
407            if (unique)
408                return ERROR;
409            else {
410                leaflevel(&t, &a, &j);
411                currkno[trx] = j;
412            }
413        }
414        else
415            currkno[trx] = ((a - trnode.keyspace) / ENTLN)+1;
416        currnode[trx] = t;
417    }
418    /* ----- Insert key into leaf node ----- */
419    while (t) {
420        nl_flag = 1;
421        rt_len = (trnode.keyspace+(bheader[trx].m*ENTLN))-a;
422        memmove(a+ENTLN, a, rt_len);
423        memmove(a, k, KLEN);
424        b = (RPTR *) (a + KLEN);
425        *b = ad;
426        if (trnode.nonleaf == FALSE) {
427            currnode[trx] = t;
428            currkno[trx] = ((a - trnode.keyspace) / ENTLN)+1;
429        }
430        trnode.keyct++;
431        if (trnode.keyct <= bheader[trx].m) {
432            write_node(t, &trnode);
433            return OK;
434        }
435        /* --- Redistribute keys between sibling nodes ---*/
436        lshft = FALSE;
437        rshft = FALSE;

```

continued...

...from previous page (Listing 6.19)

```

438     if ((yp = malloc(NODE)) == NULL)
439         memerr();
440     if (trnode.lfsib) {
441         read_node(trnode.lfsib, yp);
442         if (yp->keyct < bheader[trx].m &&
443             yp->prntnode == trnode.prntnode) {
444             lshft = TRUE;
445             redist(yp, &trnode);
446             write_node(trnode.lfsib, yp);
447         }
448     }
449     if (lshft == FALSE && trnode.rtsib) {
450         read_node(trnode.rtsib, yp);
451         if (yp->keyct < bheader[trx].m &&
452             yp->prntnode == trnode.prntnode) {
453             rshft = TRUE;
454             redist(&trnode, yp);
455             write_node(trnode.rtsib, yp);
456         }
457     }
458     free(yp);
459     if (lshft || rshft) {
460         write_node(t, &trnode);
461         return OK;
462     }
463     p = nextnode();
464     /* ----- Split node ----- */
465     if ((bp = malloc(NODE)) == NULL)
466         memerr();
467     memset(bp, '\0', NODE);
468     trnode.keyct = (bheader[trx].m + 1) / 2;
469     b = (RPTR *)
470         (trnode.keyspace+((trnode.keyct+1)*ENTLN)-ADR);
471     bp->key0 = *b;
472     bp->keyct = bheader[trx].m - trnode.keyct;
473     rt_len = bp->keyct * ENTLN;
474     a = (char *) (b + 1);

```

continued...

...from previous page (Listing 6.19)

```

475:         memmove(bp->keyspace, a, rt_len);
476:         bp->rtsib = trnode.rtsib;
477:         trnode.rtsib = p;
478:         bp->lfsib = t;
479:         bp->nonleaf = trnode.nonleaf;
480:         a -= ENTLN;
481:         memmove(k, a, KLEN);
482:         memset(a, '\0', rt_len+ENTLN);
483:         if (bheader[trx].rightmost == t)
484:             bheader[trx].rightmost = p;
485:         if (t == currnode[trx] &&
486:             currkno[trx]>trnode.keyct) {
487:             currnode[trx] = p;
488:             currkno[trx] -= trnode.keyct + 1;
489:         }
490:         ad = p;
491:         sv = t;
492:         t = trnode.prntnode;
493:         if (t)
494:             bp->prntnode = t;
495:         else {
496:             p = nextnode();
497:             trnode.prntnode = p;
498:             bp->prntnode = p;
499:         }
500:         write_node(ad, bp);
501:         if (bp->rtsib) {
502:             if ((yp = malloc(NODE)) == NULL)
503:                 memerr();
504:             read_node(bp->rtsib, yp);
505:             yp->lfsib = ad;
506:             write_node(bp->rtsib, yp);
507:             free(yp);
508:         }
509:         if (bp->nonleaf)
510:             adopt(&bp->key0, bp->keyct + 1, ad);
511:         write_node(sv, &trnode);

```

continued...

...from previous page (Listing 6.19)

```

512     if (t) {
513         read_node(t, &trinode);
514         a = trinode.keyspace;
515         b = &trinode.key0;
516         while (*b != bp->lfsib) {
517             a += ENTLN;
518             b = (RPTR *) (a - ADR);
519         }
520     }
521     free(bp);
522 }
523 /* ----- new root ----- */
524 if (p == 0)
525     p = nextnode();
526 if ((bp = malloc(NODE)) == NULL)
527     memerr();
528 memset(bp, '\0', NODE);
529 bp->nonleaf = nl_flag;
530 bp->prntnode = 0;
531 bp->rtsib = 0;
532 bp->lfsib = 0;
533 bp->keyct = 1;
534 bp->key0 = sv;
535 *((RPTR *) (bp->keyspace + KLEN)) = ad;
536 memmove(bp->keyspace, k, KLEN);
537 write_node(p, bp);
538 free(bp);
539 bheader[trx].rootnode = p;
540 if (nl_flag == FALSE) {
541     bheader[trx].rightmost = p;
542     bheader[trx].leftmost = p;
543     currnode[trx] = p;
544     currkno[trx] = 1;
545 }
546 return OK;
547 }
548 /* ----- redistribute keys in sibling nodes ----- */
549 static void redist(BTREE *left, BTREE *right)

```

continued...

...from previous page (Listing 6.19)

```

551| {
552|     int n1, n2, len;
553|     RPTR z;
554|     char *c, *d, *e;
555|     BTREE *zp;
556|
557|     n1 = (left->keyct + right->keyct) / 2;
558|     if (n1 == left->keyct)
559|         return;
560|     n2 = (left->keyct + right->keyct) - n1;
561|     z = left->prntnode;
562|     if ((zp = malloc(NODE)) == NULL)
563|         memerr();
564|     c = childptr(right->lfsib, z, zp);
565|     if (left->keyct < right->keyct) {
566|         d = left->keyspace + (left->keyct * ENTLN);
567|         memmove(d, c, KLEN);
568|         d += KLEN;
569|         e = right->keyspace - ADR;
570|         len = ((right->keyct - n2 - 1) * ENTLN) + ADR;
571|         memmove(d, e, len);
572|         if (left->nonleaf)
573|             adopt(d, right->keyct - n2, right->lfsib);
574|         e += len;
575|         memmove(c, e, KLEN);
576|         e += KLEN;
577|         d = right->keyspace - ADR;
578|         len = (n2 * ENTLN) + ADR;
579|         memmove(d, e, len);
580|         memset(d+len, '\0', e-d);
581|         if (right->nonleaf == 0 &&
582|             left->rtsib == currnode[trx])
583|             if (currkno[trx] < right->keyct - n2) {
584|                 currnode[trx] = right->lfsib;
585|                 currkno[trx] += n1 + 1;

```

continued...

..from previous page (Listing 6.19)

```

586:         }
587:         else
588:             currkno[trx] -= right->keyct - n2;
589:         }
590:     else {
591:         e = right->keyspace + ((n2-right->keyct)*ENTLN)-ADR;
592:         memmove(e, right->keyspace-ADR,
593:                 (right->keyct * ENT LN) + ADR);
594:         e -= KLEN;
595:         memmove(e, c, KLEN);
596:         d = left->keyspace + (n1 * ENT LN);
597:         memmove(c, d, KLEN);
598:         memset(d, '\0', KLEN);
599:         d += KLEN;
600:         len = ((left->keyct - n1 - 1) * ENT LN) + ADR;
601:         memmove(right->keyspace-ADR, d, len);
602:         memset(d, '\0', len);
603:         if (right->nonleaf)
604:             adopt(right->keyspace - ADR,
605:                   left->keyct - n1, left->rtsib);
606:         if (left->nonleaf == FALSE)
607:             if (right->lfsib == currnode[trx] &&
608:                     currkno[trx] > n1) {
609:                 currnode[trx] = left->rtsib;
610:                 currkno[trx] -= n1 + 1;
611:             }
612:             else if (left->rtsib == currnode[trx])
613:                 currkno[trx] += left->keyct - n1;
614:         }
615:         right->keyct = n2;
616:         left->keyct = n1;
617:         write_node(z, zp);
618:         free(zp);
619:     }
620:
621: /* ----- assign new parents to child nodes ----- */
622: static void adopt(void *ad, int kct, RPTR newp)

```

continued...

...from previous page (Listing 6.19)

```

623| {
624|     char *cp;
625|     BTREE *tmp;
626|
627|     if ((tmp = malloc(NODE)) == NULL)
628|         memerr();
629|     while (kct--) {
630|         read_node(*((RPTR *)ad, tmp);
631|         tmp->prntnode = newp;
632|         write_node(*((RPTR *)ad, tmp);
633|         cp = ad;
634|         cp += ENTLN;
635|         ad = cp;
636|     }
637|     free(tmp);
638| }
639|
640| /* ----- compute node address for a new node -----*/
641| static RPTR nextnode(void)
642| {
643|     RPTR p;
644|     BTREE *nb;
645|
646|     if (bheader[trx].rlsed_node) {
647|         if ((nb = malloc(NODE)) == NULL)
648|             memerr();
649|         p = bheader[trx].rlsed_node;
650|         read_node(p, nb);
651|         bheader[trx].rlsed_node = nb->prntnode;
652|         free(nb);
653|     }
654|     else
655|         p = bheader[trx].endnode++;
656|     return p;
657| }
658|
659| /* ----- next sequential key ----- */

```

continued...

...from previous page (Listing 6.19)

```

660 RPTR nextkey(int tree)
661 {
662     trx = tree;
663     if (currnode[trx] == 0)
664         return firstkey(trx);
665     read_node(currnode[trx], &trnode);
666     if (currkno[trx] == trnode.keyct)  {
667         if (trnode.rtsib == 0)  {
668             return 0;
669         }
670         currnode[trx] = trnode.rtsib;
671         currkno[trx] = 0;
672         read_node(trnode.rtsib, &trnode);
673     }
674     else
675         currkno[trx]++;
676     return *((RPTR *)
677                 (trnode.keyspace+(currkno[trx]*ENTLN)-ADR));
678 }
679
680 /* ----- previous sequential key ----- */
681 RPTR prevkey(int tree)
682 {
683     trx = tree;
684     if (currnode[trx] == 0)
685         return lastkey(trx);
686     read_node(currnode[trx], &trnode);
687     if (currkno[trx] == 0)  {
688         if (trnode.lfsib == 0)
689             return 0;
690         currnode[trx] = trnode.lfsib;
691         read_node(trnode.lfsib, &trnode);
692         currkno[trx] = trnode.keyct;
693     }
694     else
695         currkno[trx]--;

```

continued...

...from previous page (Listing 6.19)

```

696:     return *((RPTR *) (trnode.keyspace + (currkno[trx] * ENTLN) - ADR));
697: }
698: }
699: /* ----- first key ----- */
700: RPTR firstkey(int tree)
701: {
702:     trx = tree;
703:     if (bheader[trx].leftmost == 0)
704:         return 0;
705:     read_node(bheader[trx].leftmost, &trnode);
706:     currnode[trx] = bheader[trx].leftmost;
707:     currkno[trx] = 1;
708:     return *((RPTR *) (trnode.keyspace + KLEN));
709: }
710: }
711: /* ----- last key ----- */
712: RPTR lastkey(int tree)
713: {
714:     trx = tree;
715:     if (bheader[trx].rightmost == 0)
716:         return 0;
717:     read_node(bheader[trx].rightmost, &trnode);
718:     currnode[trx] = bheader[trx].rightmost;
719:     currkno[trx] = trnode.keyct;
720:     return *((RPTR *)
721:             (trnode.keyspace + (trnode.keyct * ENTLN) - ADR));
722: }
723: }
724: /* ----- scan to the next sequential key ----- */
725: static RPTR scannext(RPTR *p, char **a)
726: {
727:     RPTR cn;
728:     if (trnode.nonleaf) {
729:         *p = *((RPTR *) (*a + KLEN));
730:         read_node(*p, &trnode);
731:         while (trnode.nonleaf) {
732:             *p = *((RPTR *) (trnode.keyspace + (currkno[trx] * ENTLN) - ADR));
733:             read_node(*p, &trnode);
734:             currkno[trx]++;
735:         }
736:     }
737: }

```

continued...

...from previous page (Listing 6.19)

```

734:         *p = trnode.key0;
735:         read_node(*p, &trnode);
736:     }
737:     *a = trnode.keyspace;
738:     return *((RPTR *) (*a + KLEN));
739: }
740: *a += ENTLN;
741: while (-1) {
742:     if ((trnode.keyspace + (trnode.keyct)
743:          * ENTLN) != *a)
744:         return fileaddr(*p, *a);
745:     if (trnode.prtnode == 0 || trnode.rtsib == 0)
746:         break;
747:     cn = *p;
748:     *p = trnode.prtnode;
749:     read_node(*p, &trnode);
750:     *a = trnode.keyspace;
751:     while ((*((RPTR *) (*a - ADR)) != cn)
752:            *a += ENTLN;
753:     }
754:     return 0;
755: }
756
757: /* ---- scan to the previous sequential key ---- */
758: static RPTR scanprev(RPTR *p, char **a)
759: {
760:     RPTR cn;
761:
762:     if (trnode.nonleaf) {
763:         *p = *((RPTR *) (*a - ADR));
764:         read_node(*p, &trnode);
765:         while (trnode.nonleaf) {
766:             *p = *((RPTR *)
767:                  (trnode.keyspace+(trnode.keyct)*ENTLN-ADR));
768:             read_node(*p, &trnode);
769:         }
770:         *a = trnode.keyspace + (trnode.keyct - 1) * ENTLN;

```

continued...

...from previous page (Listing 6.19)

```

771      return *((RPTR *) (*a + KLEN));
772  }
773  while (-1) {
774      if (trnode.keyspace != *a) {
775          *a -= ENTLN;
776          return fileaddr(*p, *a);
777      }
778      if (trnode.prntnode == 0 || trnode.lfsib == 0)
779          break;
780      cn = *p;
781      *p = trnode.prntnode;
782      read_node(*p, &trnode);
783      *a = trnode.keyspace;
784      while ((*((RPTR *) (*a - ADR)) != cn)
785             *a += ENTLN;
786     )
787     return 0;
788 }
789 /* ----- locate pointer to child ---- */
790 static char *childptr(RPTR left, RPTR parent, BTREE *btp)
791 {
792     char *c;
793     read_node(parent, btp);
794     c = btp->keyspace;
795     while ((*((RPTR *) (c - ADR)) != left)
796            c += ENTLN;
797     return c;
798 }
799 /* ----- current key value ----- */
800 void keyval(int tree, char *ky)
801 {
802     RPTR b, p;
803     char *k;
804     int i;

```

continued...

...from previous page (Listing 6.19)

```

808|
809|     trx = tree;
810|     b = currnode[trx];
811|     if (b) {
812|         read_node(b, &trnode);
813|         i = currkno[trx];
814|         k = trnode.keyspace + ((i - 1) * ENTLN);
815|         while (i == 0) {
816|             p = b;
817|             b = trnode.prntnode;
818|             read_node(b, &trnode);
819|             for (; i <= trnode.keyct; i++) {
820|                 k = trnode.keyspace + ((i - 1) * ENTLN);
821|                 if (*((RPTR *) (k + KLEN)) == p)
822|                     break;
823|             }
824|         }
825|         memmove(ky, k, KLEN);
826|     }
827| }
828|
829| /* ----- current key ----- */
830| RPTR currkey(int tree)
831| {
832|     RPTR f = 0;
833|
834|     trx = tree;
835|     if (currnode[trx]) {
836|         read_node(currnode[trx], &trnode);
837|         f = *((RPTR *)
838|                 (trnode.keyspace+(currkno[trx]*ENTLN)-ADR));
839|     }
840|     return f;
841| }
842|
843| /* ----- read a btree node ----- */
844| static void read_node(RPTR nd, void *bf)

```

continued...

..from previous page (Listing 6.19)

```

845 {
846     bseek(nd);
847     fread(bf, NODE, 1, fp[trx]);
848     if (ferror(fp[trx])) {
849         errno = D_IOERR;
850         dberror();
851     }
852 }
853
854 /* ----- write a btree node ----- */
855 static void write_node(RPTR nd, void *bf)
856 {
857     bseek(nd);
858     fwrite(bf, NODE, 1, fp[trx]);
859     if (ferror(fp[trx])) {
860         errno = D_IOERR;
861         dberror();
862     }
863 }
864
865 /* ----- seek to the b-tree node ----- */
866 static void bseek(RPTR nd)
867 {
868     if (fseek(fp[trx],
869               (long) (NODE+((nd-1)*NODE)), SEEK_SET) == ERROR) {
870         errno = D_IOERR;
871         dberror();
872     }
873 }
874
875 /* ----- out of memory error ----- */
876 static void memerr(void)
877 {
878     errno = D_0M;
879     dberror();
880 }

```

Refer to the diagram in Figure 6-4. Btree.c is in the symbol that is labeled "B-Tree Index Software."

The B-tree index management functions are valuable utilities that you might find useful in applications other than your database. The functions in the B-tree library that you can call are described here.

Function build_b:

```
void build_b(
    char *name, /* B-tree file name */
    int len      /* key length */
)
```

When you want to establish a new B-tree, you will use this function. The **name** parameter points to the name of the B-tree file. The **len** integer is the length of the key value that will be indexed by the B-tree.

Function btree_init:

```
int btree_init(
    char *name /* B-tree file name */
)
```

This function initializes processing for an existing B-tree index file. The parameter points to the DOS file name of the B-tree file. The function returns an integer that is used for subsequent calls to the B-tree management functions. That integer is called the tree number in the descriptions of the functions that follow.

Function btree_close:

```
int btree_close(
    int tree     /* tree number */
)
```

This function closes a B-tree that was opened by **btree_init**. A program that uses the B-tree functions must call this function when it is completed. The **btree_init**

function marks B-trees with an in-use indicator. If your program terminates before calling **btree_close**, the in-use indicator will remain set, and subsequent attempts to open the B-tree will fail.

Function insertkey:

```
int insertkey(
    int tree,      /* tree number          */
    char *key,     /* key value            */
    RPTR ad,       /* key's leaf           */
    int unique)   /* TRUE = unique key   */
```

This function adds a key to the B-tree. The **key** pointer points to the key value to be added. The **ad** RPTR value will be stored with the key and is the value that is returned when the key is located in a search. The **unique** flag tells the function whether or not the B-tree should accept duplicate values. If the flag is non-zero and the value is already in the tree, then the key is not added. If the value is zero, then the key is added, but only if none of the other occurrences of the same key value are stored with an identical **ad** RPTR value.

Function locate:

```
RPTR locate(
    int tree,      /* tree number          */
    char *key,     /* key value            */
    )
```

To find a key value in a B-tree, call the **locate** function. The **tree** parameter is the integer that was returned by **btree_init**. The **key** pointer points to the value for which you are searching.

If the **key** value is in the B-tree, the RPTR value that was associated with the key when it was added to the B-tree is returned. If the key value is one of a series of identical values in the B-tree, the RPTR is the one that occurs first in the B-tree file.

If the key value is not in the B-tree file, a zero RPTR value is returned.

Function deletekey:

```
int deletekey(
    int tree,      /* tree number          */
    char *key,     /* key value            */
    RPTR ad       /* key's leaf           */
)
```

This function deletes a key from the B-tree. The **key** pointer points to the value you want to delete. The **ad** R PTR value must match the one that was stored with the key when it was added to the B-tree.

Function firstkey:

```
RPTR firstkey(
    int tree      /* tree number          */
)
```

Once a key value has been inserted or when one has been located, the software maintains pointers to the current key location in the B-tree. You can retrieve B-tree keys at random or in ascending or descending sequence. The sequential movement through the B-tree can proceed from any key location in the B-tree structure.

The **firstkey** function returns the R PTR value associated with the first key in the collating sequence of the key values of the B-tree. If no keys are stored, the function returns a zero value.

Function lastkey:

```
RPTR lastkey(
    int tree      /* tree number          */
)
```

This function returns the R PTR value associated with the last key in the collating sequence of the B-tree. If no keys are stored, the function returns a zero value.

Function nextkey:

```
RPTR nextkey(
    int tree      /* tree number           */
)
```

If you have already positioned the B-tree pointers by an insertion, deletion, or search, this function retrieves the R PTR value associated with the next higher logical key in the collating sequence of the B-tree. If no B-tree position has been established, this function calls the **firstkey** function.

Function prevkey:

```
RPTR prevkey(
    int tree      /* tree number           */
)
```

If you have already positioned the B-tree pointers by an insertion, deletion, or search, this function retrieves the R PTR value associated with the next lower logical key in the collating sequence of the B-tree. If no B-tree position has been established, this function calls the **lastkey** function.

Function keyval:

```
void keyval(
    int tree,      /* tree number           */
    char *key     /* key value            */
)
```

This function retrieves the key value associated with the current key pointer as positioned by the insertion, deletion, or search functions. The retrieved key value is copied into the location pointed to by the **key** pointer. If no key position has been established, no copy is performed.

Function currkey:

```
RPTR currkey(
    int tree      /* tree number          */
)
```

This function returns the RPTR value associated with the current key pointer as positioned by the insertion, deletion, or search functions. If no key position has been established, a zero RPTR value is returned.

SUMMARY

This chapter described the Cdata Database Management System, which is a complete DBMS that includes a Data Definition Language and a Data Manipulation Language. To support this DBMS, Chapter 7 presents a package of utility programs. These programs will provide general purpose database maintenance functions and data management functions that you can call from your programs for record display and reporting.

Breakfast

1 slice whole-wheat bread
1/2 cup orange juice

Lunch or Snack

1/2 cup cottage cheese
1/2 cup fruit cocktail

For a more filling meal, add a small bowl of cereal or a few slices of ham or bacon to your breakfast. If you prefer a light meal, add a few slices of cheese or a hard-boiled egg.

Snack or dinner

VITAMINS

Mid-morning and mid-afternoon snacks are important because they help keep energy levels up. Between meals, it's especially important to eat something nutritious. Snacks can be as simple as a banana or an apple, or as elaborate as a sandwich made with whole-wheat bread, cheese, and lettuce. You can also add a few slices of ham or bacon to your sandwich.

Supper

1/2 cup rice or pasta
1/2 cup vegetables
1/2 cup fruit

DRINKS

For a healthy diet, keep your alcohol intake to a minimum. If you do drink, limit yourself to one or two glasses of beer, wine, or liquor per day. The alcohol in beer, wine, and liquor contains many calories, so if you're trying to lose weight, it's best to avoid them.

Chapter 7

Cdata Utility Programs

Chapter 6 presented the Cdata database management system. With it you can design and build a complete software system integrated with a relational database. To make that task easier, this chapter presents several utility programs that provide access to the database in ways that are common to most database environments.

In Chapter 6, you learned to include the database schema's header file in your application program when you compiled it because the application program refers to the components of the database by using the global identifiers that are a part of the schema. You compile the schema's arrays into a separate object module and link them with your programs. The utility programs use any database schema. They deal with a generic database and use the arrays of data elements and file names to access

the database records. Therefore, it is not necessary to include a particular database's header file in the utility programs and recompile them each time you develop a new schema. You must, however, relink each utility program with the schema's object module for each new schema.

With this technique, you have a separate copy of each utility program to accompany each new database.

CDATA AND THE SCREEN AND KEYBOARD

This chapter contains keyboard and screen drivers that support the Cdata utility programs. They are not inexorably associated with Cdata. You can integrate the Cdata DBMS into an application that uses any screen, keyboard, and mouse drivers. The Cdata functions concern themselves with database management and do not conflict with video and keyboard software in any way. The Cdata utility programs in this chapter use the screen and keyboard functions that are also in this chapter. You can port them to other user interface environments if you wish.

KEYBOARD HEADER FILE (KEYS.H)

Keys.h, shown in Listing 7.1, contains a set of #define statements that equate global symbols to the function keys on the IBM PC keyboard.

the database records. Therefore, it is not necessary to include a particular database's header file in the utility programs and recompile them each time you develop a new schema. You must, however, relink each utility program with the schema's object module for each new schema.

With this technique, you have a separate copy of each utility program to accompany each new database.

CDATA AND THE SCREEN AND KEYBOARD

This chapter contains keyboard and screen drivers that support the Cdata utility programs. They are not inexorably associated with Cdata. You can integrate the Cdata DBMS into an application that uses any screen, keyboard, and mouse drivers. The Cdata functions concern themselves with database management and do not conflict with video and keyboard software in any way. The Cdata utility programs in this chapter use the screen and keyboard functions that are also in this chapter. You can port them to other user interface environments if you wish.

KEYBOARD HEADER FILE (KEYS.H)

Keys.h, shown in Listing 7.1, contains a set of #define statements that equate global symbols to the function keys on the IBM PC keyboard.

Listing 7.1 keys.h

```
1| /* ----- keys.h ----- */
2|
3| #ifndef KEYS_H
4| #define KEYS_H
5|
6| #define RUBOUT    8
7| #define HT        9
8| #define ESC       27
9|
10| #define F1        187
11| #define F2        188
12| #define F3        189
13| #define F4        190
14| #define F5        191
15| #define F6        192
16| #define F7        193
17| #define F8        194
18| #define F9        195
19| #define F10       196
20|
21| #define HOME      199
22| #define UP        200
23| #define PGUP      201
24| #define BS        203
25| #define FWD       205
26| #define END       207
27| #define DN        208
28| #define PGDN      209
29| #define INS       210
30| #define DEL       211
31|
32| #endif
```

SYSTEM SUBROUTINES (SYS.C)

Listings 7.2 and 7.3 are sys.h and sys.c, a collection of system-specific functions that are isolated in this source file to separate the compiler, operating system, and hardware dependencies from the rest of the software.

Listing 7.2 sys.h

```
1| /* ----- sys.h ----- */
2|
3| #ifndef SYS_H
4| #define SYS_H
5|
6| void put_char(int);
7| void cursor(int, int);
8| void clear_screen(void);
9| int get_char(void);
10|
11| #endif
```

Listing 7.3 sys.c

```

1  /* ----- sys.c ----- */
2
3  #include <stdio.h>
4  #include <conio.h>
5  #include "keys.h"
6  #include "sys.h"
7  #include "cdata.h"
8
9  /* =====
10   The following functions are keyboard and screen drivers
11   for the PC with ANSI.SYS installed.
12   ===== */
13
14  extern int FieldChar, screen_displayed;
15
16  /* ----- write a character to the screen ----- */
17  void put_char(int c)
18  {
19      switch (c) {
20          case FWD: printf("\033[C");
21                      break;
22          case UP:  printf("\033[A");
23                      break;
24          default:  putchar(c == ' ' ? FieldChar : c);
25      }
26      fflush(stdout);
27  }
28
29  /* ----- set the cursor position ----- */
30  void cursor(int x, int y)
31  {
32      printf("\033[%d;%dH",y+1, x+1);
33      fflush(stdout);
34  }
35
36  /* ----- clear the screen ----- */

```

continued...

...from previous page (Listing 7.3)

```

37| void clear_screen(void)
38| {
39|     screen_displayed = 0;
40|     printf("\033[2J");
41|     fflush(stdout);
42| }
43|
44| /* ----- get a keyboard character ----- */
45|
46| int get_char(void)
47| {
48|     int c;
49|
50|     if ((c = getch()) == 0)
51|         c = getch() | 128;
52|     return c & 255;
53| }
```

Function **get_char**:

```
int get_char()
```

The **get_char** function returns a character from the keyboard without the usual keyboard enhancements included when the standard C function **getchar** is used. It also handles the translation of function keys into the integer values as defined in **keys.h**.

Function **put_char**:

```

void put_char(
    int c      /* the character to write */
)
```

The **put_char** function is the output equivalent of **get_char**. It uses the standard C function named **putchar**, but it allows you to send it the global values **FWD** and **UP** to move the cursor forward one character position or up one line. It also translates

the space character to the value recorded in the global integer variable named **FieldChar**.

Function clear_screen:

```
void clear_screen()
```

The **clear_screen** function clears the screen.

Function cursor:

```
void cursor(  
    int x,      /* cursor column coordinate */  
    int y      /* cursor row coordinate */  
)
```

The **cursor** function positions the cursor at specified character x and y coordinates.

The functions in sys.c use the protocols of the ANSI.SYS terminal driver for moving the cursor and clearing the screen. There are faster ways to perform these actions. The IBM PC includes low-level functions in its ROM BIOS that an application program can call for screen and keyboard manipulation. No C language standard exists for using these calls, so the compilers differ in their treatment of the problem. You can use the features of your compiler to access these functions, and you will see an improvement in the performance of these programs' display features. But, in doing so, you will sacrifice a measure of portability.

THE DATABASE SIZE CALCULATOR (DBSIZE.C)

After designing a database, you need to know how much disk storage it requires. You might be thinking about storing it on a diskette or a small capacity hard disk. You could be sharing space with other software and data. Whatever the reason, you need a method for estimating the storage requirements. Knowing the data and index file formats and guessing at the data file volumes, you could sit down with a calculator and figure it out yourself. But there is a better way. Since the storage required for a Cdata database is a function of the number of records in each file and the number of indexes into each file, a program is included that calculates the disk space from estimates of the file sizes. The program is named dbsize.c and is shown in Listing 7.4.

Listing 7.4 dbsize.c

```

1| /* ----- dbsize.c ----- */
2| /* Compute the character size of a database */
3|
4| #include <stdio.h>
5| #include "cdata.h"
6| #include "btree.h"
7| #include "datafile.h"
8|
9| static void index_m(int, int *);
10|
11 void main(void)
12 {
13     int f, x;
14     long rct [MXFILS];          /* number of records/file */
15     long fsizes [MXFILS];       /* file sizes */
16     long dsize = 0;             /* data base sizes */
17     int m [MXINDEX+1];          /* btree m values */
18     long xsizes [MXFILS] [MXINDEX]; /* index sizes */
19|
20     for (f = 0; dbfiles [f]; f++) {
21         printf("\nEnter record count for %-10s: ",dbfiles[f]);
22         fflush(stdout);
23         scanf("%ld", &rct [f]);
24         fsizes [f] = rct [f] * rlen(f) + sizeof(FHEADER);
25         printf("File size: %10ld",
26                fsizes [f]);
27         dsize += fsizes [f];
28         index_m(f, m);
29         for (x = 0; m [x]; x++) {
30             xsizes [f] [x] = (2 + (rct [f] / m [x])) * NODE;
31             dsize += xsizes [f] [x];
32             printf(
33                 "\nIndex %d (m=%2d) size: %10ld",
34                 x+1,m[x],xsizes[f][x]);
35         }
36     }
37     printf("\n-----");
38     printf("\nData base size: %10ld", dsize);

```

continued...

..from previous page (Listing 7.4)

```

39 }
40
41 /* --- compute the btree m values
42      for the indices of a data base file --- */
43 static void index_m(int f, int *m)
44 {
45     int x, x1;
46     int len;
47
48     for (x = 0; x < MXINDEX; x++) {
49         if (index_ele [f] [x] == NULL)
50             break;
51         len = 0;
52         for (x1 = 0; index_ele [f] [x] [x1]; x1++)
53             len += ellen [index_ele [f] [x] [x1] - 1];
54         *m++ = ((NODE-(sizeof(int)*2)
55                  -sizeof(RPTR)*4))/(len+sizeof(RPTR));
56     }
57     *m = 0;
58 }
```

To run the program, type its name: dbsize. The program will then ask you about the number of records in each file. Dbsize uses your answers to calculate the data file and index file storage requirements. Screen 7.1 shows this program being run against the CBS schema.

```
C>dbsize

Enter record count for CLIENTS : 12
File size: 1450
Index 1 (m=54) size: 1024
Enter record count for PROJECTS : 25
File size: 1210
Index 1 (m=54) size: 1024
Enter record count for CONSULTANTS: 10
File size: 330
Index 1 (m=54) size: 1024
Enter record count for ASSIGNMENTS: 35
File size: 640
Index 1 (m=35) size: 1536
Index 2 (m=54) size: 1024
Index 3 (m=54) size: 1024
-----
Data base size: 10286
```

Screen 7.1. Dbize Session for CBS Schema.

DATABASE INITIALIZATION (DBINIT.C)

After you have designed a database schema and before your applications programs can access the database, you must build initial copies of the data and index files with no records in them. The program dbinit.c performs this task for you. When run, it initializes the entire database. If a copy of the database already exists, dbinit erases it and replaces the copy with the new, empty database. The dbinit.c program is shown in Listing 7.5.

Listing 7.5 dbinit.c

```
1| /* ----- dbinit.c ----- */
2|
3| /* This program is used to build the initial data base.
4| It constructs all files and indexes.
5| There is no data loaded,
6| and any existing files are deleted.      */
7|
8| #include <stdio.h>
9| #include "cdata.h"
10| #include "datafile.h"
11|
12| void main(void)
13| {
14|     int f = 0;
15|     extern int rlen();
16|     char fname [13];
17|
18|     while (dbfiles [f]) {
19|         sprintf(fname, "%.8s.dat", dbfiles [f]);
20|         file_create(fname, rlen(f));
21|         printf("\nCreating file %s with length %d",
22|                           fname, rlen(f));
23|         build_index("", f);
24|         f++;
25|     }
26| }
```

To run dbinit, type its name. Screen 7.2 shows a dbinit session that uses the CBS database schema to initialize the CBS database.

```
C>dbinit  
Creating file CLIENTS.dat with length 120  
Creating file PROJECTS.dat with length 48  
Creating file CONSULTA.dat with length 32  
Creating file ASSIGNME.dat with length 18
```

Screen 7.2. Dbinit Session for CBS Data Base.

DATA ENTRY AND QUERY (QD.C)

Except for the invoice program that served as an example in Chapter 6, no software has been developed to access the data in the database files. Later, you will write some custom programs, but, for now, you just want to store, look at, change, and delete some records in files.

When you develop an on-line application, you code data entry and retrieval programs. Many of these programs are logically similar but differ in the files accessed and the data elements involved. The user enters the primary key data element value for the record desired, and the program displays the record. The user moves the cursor around the screen and changes data element values. Then the user tells the program to store that record and get another one. Maybe the user deletes a record or adds a new one. The process is universal; most DBMS packages include a program that allows the user to browse the database, doing all the things just mentioned. Cdata has such a program. It is called qd.c and is shown in Listing 7.6.

Listing 7.6 qd.c

```

1| /* ----- qd.c ----- */
2| /* A query program. Enter the name of the data base file and
3| * (optionally) a list of data elements. A screen is built
4| * and the file is updated based upon the data entered.
5| */
6| #include <stdio.h>
7| #include <stdlib.h>
8| #include <string.h>
9| #include "cdata.h"
10| #include "screen.h"
11| #include "keys.h"
12|
13| static int file;
14| static int existing_record;
15| static char *rb;           /* record buffer      */
16| static char *hb;           /* hold buffer        */
17| static char *sc;           /* screen buffer      */
18| static int len;            /* record length      */
19| static int fl[] = {0, -1};
20| static int iplist [MXELE+1];
21| static const int *els;
22|
23| /* ----- prototypes ----- */
24| static void query(void);
25| static void clear_record(void);
26| static void rcdin(void);
27| static void rcdout(void);
28| static int empty(char *, int);
29| static int same(void);
30| static void set_trap(void);
31| static int key_entry(char *);
32|
33| void main(int argc, char *argv[])
34| {
35|     if (argc > 1)  {
36|         if ((file = filename(argv[1])) != ERROR)   {
37|             if (argc == 2)  {

```

continued...

...from previous page (Listing 7.6)

```

38:           len = rlen(file);
39:           els = (int *) file_ele [file];
40:       }
41:   else if (ellist(argc-2, argv+2, iplist) == OK) {
42:       len = epos(0, iplist);
43:       els = iplist;
44:   }
45:   else
46:       exit(1);
47:   sc = malloc(len);          /* screen buffer */
48:   rb = malloc(rlen(file));  /* record buffer */
49:   hb = malloc(rlen(file));  /* hold buffer */
50:   if (sc == NULL || rb == NULL || hb == NULL) {
51:       printf("\nOut of memory");
52:       exit(1);
53:   }
54:   init_rcd(file, rb);
55:   init_rcd(file, hb);
56:   init_screen(argv[1], els, sc);
57:   query();
58:   free(hb);
59:   free(rb);
60:   free(sc);
61: }
62: }
63: }
64: /* ----- process the query ----- */
65: static void query(void)
66: {
67:     int term = 0;
68:     *fl = file;
69:     db_open("", fl);
70:     clrrcd(sc, els);
71:     set_trap();
72:     while (term != ESC) {
73:         ...
74:     }

```

continued...

...from previous page (Listing 7.6)

```

75      term = data_entry();
76      switch (term) {
77          /* ----- GO ----- */
78          case F1:    rcdout();
79              break;
80          /* ----- First record ----- */
81          case HOME:  rcdout();
82              if (first_rcd(file, 1, rb) == ERROR)
83                  post_notice("Empty file");
84              else
85                  rcdin();
86              break;
87          /* ----- First record ----- */
88          case END:   rcdout();
89              if (last_rcd(file, 1, rb) == ERROR)
90                  post_notice("Empty file");
91              else
92                  rcdin();
93              break;
94          /* ----- Previous record ----- */
95          case PGUP:  rcdout();
96              if (prev_rcd(file, 1, rb) == ERROR) {
97                  post_notice("Beginning of file");
98                  if (first_rcd(file, 1, rb) ==
99                      ERROR) {
100                     post_notice("Empty file");
101                     break;
102                 }
103             }
104             rcdin();
105             break;
106            /* ----- Next record ----- */
107            case PGDN: rcdout();
108            if (next_rcd(file, 1, rb) == ERROR) {
109                post_notice("At end of file");
110                if (last_rcd(file, 1, rb) ==
111                    ERROR) {

```

continued...

..from previous page (Listing 7.6)

```
112                     post_notice("Empty file");
113                     break;
114                 }
115             }
116             rcdin();
117             break;
118         /* ----- Delete record ----- */
119         case F7:    if (spaces(rb) == 0) {
120                     post_notice("Verify w/F7");
121                     if (get_char() == F7)  {
122                         del_rcd(file);
123                         clear_record();
124                     }
125                     clear_notice();
126                 }
127                 break;
128         case ESC:   if (spaces(sc))
129                     break;
130                     clear_record();
131                     term = 0;
132                     break;
133         default:   break;
134     }
135 }
136 clear_screen();
137 db_cls();
138 }
139
140 /* ----- clear out the record area ----- */
141 static void clear_record(void)
142 {
143     int i = 0;
144     while (index_ele[file][0][i])
145         protect(index_ele[file][0][i++], FALSE);
146     clrrcd(sc, els);
147     existing_record = FALSE;
148 }
149
150 /* ----- get the data base file record ----- */
continued...
```

...from previous page (Listing 7.6)

```

151| static void rcdin(void)
152| {
153|     int i = 0;
154|
155|     if (empty(rb, rlen(file)) == 0) {
156|         rcd_fill(rb, sc, file_ele[file], els);
157|         memmove(hb, rb, rlen(file));
158|         existing_record = TRUE;
159|         while (index_ele[file][0][i])
160|             protect(index_ele[file][0][i++],TRUE);
161|     }
162| }
163|
164| /* ----- add or update the data base file record ----- */
165| static void rcdout(void)
166| {
167|     if (empty(sc, len) == 0)      {
168|         rcd_fill(sc, rb, els, file_ele[file]);
169|         if (existing_record)      {
170|             if (same() == 0)        {
171|                 post_notice("Returning record");
172|                 rtn_rcd(file, rb);
173|             }
174|         }
175|         else          {
176|             post_notice("New record added");
177|             if (add_rcd(file, rb) == ERROR)
178|                 dberror();
179|         }
180|         clear_record();
181|     }
182| }
183|
184| /* ----- test for an empty record buffer ----- */
185| static int empty(char *b, int l)

```

continued...

...from previous page (Listing 7.6)

```
186: {
187:     while (l--)
188:         if (*b && *b != ' ')
189:             return FALSE;
190:         else
191:             b++;
192:     return TRUE;
193: }
194:
195: /* ----- test two record buffers for equality ----- */
196: static int same(void)
197: {
198:     int ln = rlen(file);
199:
200:     while (--ln)
201:         if (*(rb + ln) != *(hb + ln))
202:             break;
203:     return (*(rb + ln) == *(hb + ln));
204: }
205:
206: /* ----- set the query screen's key element trap ----- */
207: static void set_trap(void)
208: {
209:     int i = 0;
210:
211:     while (index_ele [file] [0] [i])
212:         i++;
213:     edit(index_ele [file] [0] [i-1], key_entry);
214: }
215:
216: /* --- come here when the primary key has been entered ---- */
217: static int key_entry(char *s)
218: {
219:     char key [MXKEYLEN];
220:     int i;
221:
222:     if (spaces(s))
```

continued...

...from previous page (Listing 7.6)

```
223|     return OK;
224|     *key = '\0';
225|     i = 0;
226|     while (index_ele [file] [0] [i])    {
227|         protect(index_ele[file][0][i],TRUE);
228|         strcat(key, sc + epos(index_ele[file][0][i++], els));
229|     }
230|     if (find_rcd(file, 1, key, rb) == ERROR)    {
231|         post_notice("New record");
232|         existing_record = FALSE;
233|         return OK;
234|     }
235|     rcdin();
236|     tally();
237|     return OK;
238| }
```

The qd program allows you to specify a file and, optionally, a list of data elements. It constructs a data entry screen, displays it, and waits for you to key in the primary key data element. You can use a list of data elements on the command line when you execute qd, but you must include the primary key data element as the first one.

Screen 7.3 shows a session using qd with the CLIENTS file in the CBS database. It shows (a) the command line that tells qd to display the client number, phone number, and amount due from the file of clients. Then it shows (b) the screen template built by qd. In the next screen (c), the user has entered the client number 00015. The last screen (d) shows the record that qd has retrieved.

a.	C>qd clients client_no client_name amt_due										
b.	<table><thead><tr><th></th><th style="text-align: right;">--- clients ---</th></tr></thead><tbody><tr><td>CLIENT NO</td><td>_____</td></tr><tr><td>CLIENT NAME</td><td>_____</td></tr><tr><td>PHONE</td><td>(____) ____-____</td></tr><tr><td>AMT DUE</td><td>\$ ____.-__</td></tr></tbody></table>		--- clients ---	CLIENT NO	_____	CLIENT NAME	_____	PHONE	(____) ____-____	AMT DUE	\$ ____.-__
	--- clients ---										
CLIENT NO	_____										
CLIENT NAME	_____										
PHONE	(____) ____-____										
AMT DUE	\$ ____.-__										
c.	<table><thead><tr><th></th><th style="text-align: right;">--- clients ---</th></tr></thead><tbody><tr><td>CLIENT NO</td><td>00015</td></tr><tr><td>CLIENT NAME</td><td>_____</td></tr><tr><td>PHONE</td><td>(____) ____-____</td></tr><tr><td>AMT DUE</td><td>\$ ____.-__</td></tr></tbody></table>		--- clients ---	CLIENT NO	00015	CLIENT NAME	_____	PHONE	(____) ____-____	AMT DUE	\$ ____.-__
	--- clients ---										
CLIENT NO	00015										
CLIENT NAME	_____										
PHONE	(____) ____-____										
AMT DUE	\$ ____.-__										
d.	<table><thead><tr><th></th><th style="text-align: right;">--- clients ---</th></tr></thead><tbody><tr><td>CLIENT NO</td><td>00015</td></tr><tr><td>CLIENT NAME</td><td>Ace_Wrecking_Yard_____</td></tr><tr><td>PHONE</td><td>(305)123-4567</td></tr><tr><td>AMT DUE</td><td>\$ __1500.00</td></tr></tbody></table>		--- clients ---	CLIENT NO	00015	CLIENT NAME	Ace_Wrecking_Yard_____	PHONE	(305)123-4567	AMT DUE	\$ __1500.00
	--- clients ---										
CLIENT NO	00015										
CLIENT NAME	Ace_Wrecking_Yard_____										
PHONE	(305)123-4567										
AMT DUE	\$ __1500.00										

Screen 7.3. A qd Session with the CLIENTS File.

Once qd has retrieved a record, you can change its data by moving the cursor among the data element fields and changing the data values. When you are satisfied with the record's new contents, press the function key F1 to return the changed record to the database. The qd program displays an empty template, and you can start again. To reject any changes you have made to a record, press the [Esc] key rather than F1.

You can display the previous or the next record on the screen by using function keys PgUp and PgDn. Pressing the PgUp key at the beginning of the file causes qd to display the first record in the key sequence. Pressing the PgDn key at the end of the file displays the last record. If you use the PgUp or PgDn keys after you have changed data values in the current record, qd writes the new record to the database in place of the original and then retrieves the previous or next record. Use the Home and End keys to display the first and last record in the file.

You can use qd to add and delete records from the database. If you enter a primary key that does not exist, qd displays the message, "Adding new record." If you did not mean to add a new record—perhaps you entered the wrong primary key value—then press the [Esc] key to reject the addition. If you did mean to add a record, you can enter data values into the data element fields. Press the F1 key when the data values are correct.

To delete a record using qd, enter its key value and review the record that qd retrieves. Then press the F7 key (the delete function key). Qd displays a message that says, "Verify w/F7." This message tells you to verify your delete request by pressing the F7 key a second time. Qd deletes the record and displays an empty template. If you press something other than the F7 key, qd rejects the delete request.

The cursor arrow keys, the Enter key, and the Tab key move the cursor around the screen. Use the Ins key to toggle the Insert mode. In the Insert mode, you can insert characters into a field. Each keystroke shifts the characters on the right of the cursor one position to the right. When the Insert mode is off, the new keystrokes overwrite the characters under the cursor. Use the Del key to delete the character under the cursor. Use the Backspace key to delete the character to the left of the cursor.

To exit qd, press the [Esc] key when the template is empty.

Table 7-1 is a summary of the keyboard functions for qd.

Table 7-1. Keyboard Functions for Qd.

F1	Write a changed/new record to the data base.
F7	Delete a record from the data base.
PgUp	View the previous record.
PgDn	View the next record.
Home	View the first record.
End	View the last record.
Esc	(When record is visible) Reject any additions/changes. Clear the template. (When template is empty) Exit from qd.
Ins	Toggle keyboard Insert mode.
Tab	Move cursor to next field.
Enter	" " " "
Down Arrows	" " " "
Up Arrow	Move cursor to previous field.
Right Arrow	Move cursor right one character.
Left Arrow	Move cursor left one character.
Backspace <-	Delete character to left of cursor.
Del	Delete character under cursor.

DATABASE FILE REPORT PROGRAM (DS.C)

Most automated systems include printed reports taken from the data in the database files. As you develop applications software, you address the requirements for specific reports. But before you can satisfy those requirements, you need a report program that lists the contents of the data files. As with the qd program, the report program should be general enough to allow you to specify a file and the data elements that you want to show on the report.

To meet this reporting requirement, a program called ds.c is provided as shown in Listing 7.7.

Listing 7.7 ds.c

```

1  /* ----- ds.c ----- */
2
3  /* Display the data of a file on stdout. Type the filename
4  * and a list of data element names. The query response is
5  * sent to the standard output device. If no element list
6  * is typed, the query uses the format of the file record.
7  */
8  #include <stdio.h>
9  #include <ctype.h>
10 #include <string.h>
11 #include "cdata.h"
12
13 void main(int argc, char *argv[])
14 {
15     DBFILE f;
16     ELEMENT iplist [MXELE+1];
17     static DBFILE fl[] = {0, -1};
18     FILE *fp = stdout;
19
20     if (argc > 1) {
21         if ((f = filename(argv[1])) != ERROR) {
22             *fl = f;
23             db_open("", fl);
24             /* -- test for file spec on the command line -- */
25             if (strcmp(argv[2], "-f") == 0) {
26                 fp = fopen(argv[3], "w");
27                 argv += 2;
28                 argc -= 2;
29             }
30             if (fp != NULL) {
31                 /* --- test for a data element list --- */
32                 if (argc == 2)
33                     dblist(fp, f, 1, file_ele [f]);
34                 else if (ellist(argc-2, argv+2, iplist)==OK)
35                     dblist(fp, f, 0, iplist);
36                 db_cls();
37                 if (fp != stdout)

```

continued...

..from previous page (Listing 7.7)

```
38:             fclose(fp);
39:         }
40:     else
41:         fprintf(stderr, "\nCannot open %s", argv[2]);
42:     }
43:     else fprintf(stderr,
44:                  "\n%s is not a data base file", argv[1]);
45:   }
46: else
47:   fprintf(stderr,
48:           "\nUsage: ds <data base file> [ -f <output file> ] "
49:           "[ data element list ]");
50: }
```

You run the ds program by using the same command line format as in the qd program. You may redirect the output to the printer or display it on the screen. Use the DOS I/O redirection convention to send the report to the printer. (Type ">prn" on the command line along with the parameters. See your DOS User's Guide for details on I/O redirection.) The program senses which output you've chosen and acts accordingly; it causes a page eject for the printer or pauses and waits for a keystroke when the screen is full. The program also restricts a screen display to an 80-column format, truncating at the data element level. Reports directed to the printer may be up to 136 characters long. Screen 7.4 shows the command line and resulting report when ds reports the PROJECTS file from the CBS database. Even though all the data elements from the file are shown, their arrangement on the report line is different from their arrangement in the file, which shows how you can use the command line to select specific data elements for the report.

Screen 7.4			
C>ds projects client_no project_no project_name amt_expended			
Filename: PROJECTS			
CLIENT_NO	PROJECT_NO	PROJECT_NAME	AMT_EXPENDED
00022	00001	Payroll	\$ 12250.00
00022	00002	Inventory	\$ 10295.00
00022	00003	General Ledger	\$ 7501.00
Records: 3			

Screen 7.4. General Report Program (ds).

If you do not specify data elements on the command line, the ds program uses all the data elements in the file's schema and reports the records in the sequence of the primary key. If you specify selected data elements, the report is in the physical order of the records in the file and displays only the data elements you specify.

If the report line does not fit on a display line (80 columns on the screen, 132 columns on the printer), ds truncates the rightmost data elements. The ds program calls the function named dblist to prepare the display or report output. Because dblist is a useful tool for applications programs, it is described and listed separately.

INDEX BUILDER (INDEX.C)

You use this utility program when you have had a problem, which occurs most often during testing. After the system is operational, you use the index program mainly to recover from power failures, but when you are testing and blowing up programs, you use it often.

The inverted index files of Cdata use B-tree algorithms. If a program terminates without bringing the indexes to an orderly close, the system locks the B-trees. This measure is correct because the trees are unreliable if they have not been properly closed. The next time you try to use them, Cdata will sense this condition.

The only correct measure is to rebuild the indexes by running the program named index.c, as shown in Listing 7.8. Make sure that index.exe is available on the default DOS subdirectory whenever one of your Cdata applications programs is running.

Listing 7.8 index.c

```

1| /* ----- index.c ----- */
2| #include <stdio.h>
3| #include <process.h>
4| #include <stdlib.h>
5| #include <string.h>
6| #include "cdata.h"
7|
8| extern RPTR curr_a[]; /* current record file address */
9|
10 void main(int argc, char *argv[])
11 {
12     int f = 1, i;
13     static int fs [MXFILS+1];
14     char *bf, *path;
15
16     if (argc < 2) {
17         printf("\nFile name or 'all' required");
18         exit(1);
19     }
20     if (strcmp("all", argv [1]) == 0) /* index all */
21         for (f = 0; dbfiles [f]; f++) /* put files in list */
22             fs [f] = f;
23     else if ((fs = filename(argv[1])) == ERROR)
24         exit(1);
25     fs [f] = (-1); /* terminator (file,file,...,-1) */
26     path = argc > 2 ? argv[2] : "";
27     /* delete and rebuild indexes
28         in the data base files being indexed */
29     for (i = 0; (f = fs [i]) != (-1); i++)
30         build_index(path, f);
31     /* Open the data base files. */
32     db_open(path, fs);
33     for (i = 0; (f = fs [i]) != (-1); i++) {
34         printf("\nIndexing %s", dbfiles [f]);
35         if ((bf = malloc(rlen(f))) != NULL) {
36             while (seqrcd(f, bf) != ERROR)
37                 add_indexes(f, bf, curr_a [f]);

```

continued...

...from previous page (Listing 7.8)

```

38|           free(bf);
39|       }
40|   }
41|   db_cls();
42| }
```

You can run this program anytime you are in doubt about the integrity of an index against a particular file in the database or against all of the files. To run index, enter its name followed either by the word "all" or by a list of database file names as shown here:

```
C>index all
C>index projects clients
```

The index program calls the function named filename to parse the file names that you enter on the command line. You will find this function discussed in Chapter 6.

SOME USEFUL UTILITY FUNCTIONS

The utility programs in this chapter use several functions that are described here. They are separated from the programs because you can use them in the development of applications programs. This aspect qualifies them as tools in the toolset and, as such, they are maintained apart from the programs that use them.

Parse Command Line Data Element List (ellist.c)

Function ellist:

```

int ellist(
    int count,          /* number of names in the list */
    char *names[],      /* the names */
    int *list           /* the resulting list */
)
```

Remember in Chapter 6 that Cdata describes a record by using an array of data element global integers terminated with an integer of zero value. Remember also that the names of the data elements are defined in an array of pointers to ASCII strings. The function named **ellist** is in ellist.c, Listing 7.9. It converts a list of ASCII data

element names into an integer array similar to a Cdata record definition array. This function allows you to specify queries and reports with a selected set of data elements on the command line. The utility programs *qd* and *ds* both use this function.

Listing 7.9 ellist.c

```

1  /* ----- ellist.c ----- */
2
3  /*
4   * Construct list of ELEMENTs from list of names
5   * such as might be entered on a command line.
6   * Return OK, or ERROR if one of the names is not in the
7   * dictionary.
8   */
9
10 #include <stdio.h>
11 #include <string.h>
12 #include "cdata.h"
13
14 int ellist(int count, char *names[], ELEMENT *list)
15 {
16     char elname[31];
17     ELEMENT el, el1;
18
19     for (el = 0; el < count; el++) {
20         for (el1 = 0; ; el1++) {
21             if (denames[el1] == NULL) {
22                 fprintf(stderr,
23                         "\nNo such data element as %s", elname);
24                 return ERROR;
25             }
26             name_cvt(elname, names[el]);
27             if (strcmp(elname, denames[el1]) == 0)
28                 break;
29         }
30         *list++ = el1 + 1;
31     }
32     *list = 0;
33     return OK;
34 }
```

When you call **ellist**, you pass it the number of names in the list and the address of an array of character pointers that point to the names. Also, pass it the address of an integer array where the resulting record-defining array will be built.

Elist will return OK if the list is properly built and ERROR if one of the names in the strings is not a valid data element name. If **ellist** finds an error, it displays an error message on the console before it returns.

Screen Manager (screen.c)

Listing 7.10 is screen.h, the header file that defines the prototypes for the screen manager functions.

Listing 7.10 screen.h

```
1| /* ----- screen.h ----- */
2|
3| #ifndef SCREEN_H
4| #define SCREEN_H
5|
6| #include "sys.h"
7|
8| /* ----- screen driver function definitions ----- */
9| void init_screen(char *, const ELEMENT *, char *);
10| void protect(ELEMENT, int);
11| void edit(ELEMENT, int (*())());
12| void display_template(void);
13| int data_entry(void);
14| int spaces(char *);
15| void tally(void);
16| void put_field(ELEMENT);
17| void error_message(char *);
18| void clear_notice(void);
19| void post_notice(char *);
20|
21| #endif
```

The screen manager functions (screen.c, Listing 7.11) are simple enough that they do not require a lot of memory or processing time, yet they are sufficient to manage the simple screen input and output requirements of the Cdata utility programs. The screen functions are the mainstay of the qd program, and you will see that they do quite a bit of work, considering their size. They use the record definition conventions of Cdata to manage data entry into a screen template.

Listing 7.11 screen.c

```
1| /* ----- screen.c ----- */
2|
3| #include <stdio.h>
4| #include <string.h>
5| #include <ctype.h>
6| #include <stdlib.h>
7| #include <errno.h>
8| #include "cdata.h"
9| #include "screen.h"
10| #include "keys.h"
11|
12| int FieldChar = '_';           /* field filler character */
13|
14| /* ----- prototypes ----- */
15| static int elp(ELEMENT);
16| static int no flds(void);
17| static void data_coord(ELEMENT);
18| static int read_element
19|     (const char,const char *,char *);
20| static void right_justify(char *);
21| static void right_justify_zero_fill(char *);
22| static int validate_date(char *);
23| static void disp_element(char *, const char *);
24| static void insert_status(void);
25| static void database_error(void);
26| static int endstroke(int);
27|
28| /* ----- data element structure ----- */
29| static struct {
```

continued...

...from previous page (Listing 7.11)

```

30:     int prot;                      /* element protected = TRUE */
31:     int (*edits)();                 /* custom edit function      */
32: } sb [MXELE];
33:
34: static const ELEMENT *elist;      /* data element list         */
35: static char *bf;                  /* data entry buffer         */
36: static char *tname;               /* screen template name      */
37:
38: static int notice_posted = 0;    /* notice on screen = TRUE   */
39: static int insert_mode = FALSE;  /* insert mode, TRUE/FALSE   */
40: static int prev_col, prev_row;  /* current cursor location   */
41: int screen_displayed;           /* template displayed flag   */
42:
43: /* ----- initialize the screen process ----- */
44: void init_screen(char *name, const ELEMENT *els, char *bfr)
45{
46:     tname = name;
47:     elist = els;
48:     bf = bfr;
49:     database_message = database_error;
50}
51:
52: /* ----- set the protect flag for a screen field ----- */
53: void protect(ELEMENT el, int tf)
54{
55:     sb[elp(el)].prot = tf;
56}
57:
58: /* ---- set the field edit function for a screen field --- */
59: void edit(ELEMENT el, int (*func)())
60{
61:     sb[elp(el)].edits = func;
62}
63:
64: /* ---- compute the relative position
65:      of an element on a screen ----- */
66: static int elp(ELEMENT el)
67{

```

continued...

...from previous page (Listing 7.11)

```

68:     int i;
69:
70:     for (i = 0; *(elist + i); i++)
71:         if (el == *(elist + i))
72:             break;
73:     return i;
74: }
75:
76: /* ----- Display the crt template ----- */
77: void display_template(void)
78: {
79:     int i, ct;
80:     ELEMENT el;
81:     char detag[16], *cp2;
82:     const char *cp1;
83:
84:     clear_screen();
85:     screen_displayed = TRUE;
86:     ct = no flds();
87:     printf("\n                                --- %s ---\n", tname);
88:     for (i = 0; i < ct; i++)    {
89:         el = *(elist + i) - 1;
90:         cp1 = denames[el];
91:         cp2 = detag;
92:         while (*cp1 && cp2 < detag + sizeof detag - 1)  {
93:             *cp2++ = *cp1 == '_' ? ' ' : *cp1;
94:             cp1++;
95:         }
96:         *cp2 = '\0';
97:         printf("\n%16.16s %s", detag, elmask[el]);
98:     }
99:     printf("\n");
100:    insert_status();
101: }
102:
103: /* ----- Process data entry for a screen template. ----- */
104: int data_entry(void)
105: {
106:     int (*validfunct)(char *, int);

```

continued...

..from previous page (Listing 7.11)

```

107|     int field_ctr, exitcode;
108|     ELEMENT el;
109|     int field_ptr = 0, done = FALSE, isvalid;
110|     char *bfptr;
111|
112|     if (screen_displayed == 0)
113|         display_template();
114|     tally();
115|     field_ctr = no flds();
116|     /* ---- collect data from keyboard into screen ---- */
117|     while (done == FALSE) {
118|         bfptr = bf + epos(list[field_ptr], elist);
119|         el = *(elist + field_ptr) - 1;
120|         validfunct = sb[field_ptr].edits;
121|         data_coord(el + 1);
122|         if (sb[field_ptr].prot == FALSE) {
123|             exitcode =
124|                 read_element(eltype[el], elmask[el], bfptr);
125|             isvalid = (exitcode != ESC && validfunct) ?
126|                         (*validfunct)(bfptr,exitcode) : OK;
127|         }
128|         else {
129|             exitcode = FWD;
130|             isvalid = OK;
131|         }
132|         if (isvalid == OK)
133|             switch (exitcode) { /* passed edit */
134|                 case DN:           /* cursor down key */
135|                 case '\r':        /* enter/return */
136|                 case '\t':        /* horizontal tab */
137|                 case FWD:          /* -> */
138|                     if (field_ptr+1 == field_ctr)
139|                         field_ptr = 0;
140|                     else
141|                         field_ptr++;
142|                     break;
143|                 case UP:           /* cursor up key */
144|                 case BS:           /* back space */

```

continued...

...from previous page (Listing 7.11)

```
145:             if (field_ptr == 0)
146:                 field_ptr = field_ctr - 1;
147:             else
148:                 field_ptr--;
149:             break;
150:         default:
151:             done = endstroke(exitcode);
152:             break;
153:         }
154:     }
155:     return (exitcode);
156: }
157:
158: /* ----- Compute the number of fields on a template ----- */
159: static int no flds(void)
160: {
161:     int ct = 0;
162:
163:     while (*(elist + ct))
164:         ct++;
165:     return ct;
166: }
167:
168: /* ----- compute data element field coordinates ----- */
169: static void data_coord(ELEMENT el)
170: {
171:     prev_col = 17;
172:     prev_row = elp(el) + 3;
173: }
174:
175: /* ----- read data element from keyboard ----- */
176: static int read_element
177:     (const char type,const char *msk,char *bfr)
178: {
179:     const char *mask = msk;
180:     char *buff = bfr;
181:     int done = FALSE, c, column = prev_col;
182:
183:     while (*mask != FieldChar) {
```

continued...

..from previous page (Listing 7.11)

```

184     prev_col++;
185     mask++;
186 }
187 while (TRUE) {
188     cursor(prev_col, prev_row);
189     c = get_char();
190     clear_notice();
191     switch (c) {
192         case '\b':
193         case BS:
194             if (buff == bfr) {
195                 done = c == BS;
196                 break;
197             }
198             --buff;
199             do {
200                 --mask;
201                 --prev_col;
202             } while (*mask != FieldChar);
203             if (c == BS)
204                 break;
205         case DEL:
206             memmove(buff, buff+1, strlen(buff));
207             *(buff+strlen(buff)) = ' ';
208             cursor(prev_col, prev_row);
209             disp_element(buff,mask);
210             break;
211         case FWD:
212             do {
213                 prev_col++;
214                 mask++;
215             } while (*mask && *mask != FieldChar);
216             buff++;
217             break;
218         case INS:
219             insert_mode ^= TRUE;
220             insert_status();
221             break;

```

continued...

...from previous page (Listing 7.11)

```

222|         case '.':
223|             if (type == 'C')    {
224|                 if (*mask++ && *buff == ' ')    {
225|                     *buff++ = '0';
226|                     if (*mask++ && *buff == ' ')
227|                         *buff++ = '0';
228|                 }
229|                 right_justify(bfr);
230|                 cursor(column, prev_row);
231|                 disp_element(bfr, msk);
232|                 prev_col = column + strlen(msk)-2;
233|                 mask = msk + strlen(msk)-2;
234|                 buff = bfr + strlen(bfr)-2;
235|                 break;
236|             }
237|             default:
238|                 if (endstroke(c))   {
239|                     done = TRUE;
240|                     break;
241|                 }
242|                 if (type != 'A' && !isdigit(c)) {
243|                     error_message("Numbers only");
244|                     break;
245|                 }
246|                 if (insert_mode)   {
247|                     memmove(buff+1, buff, strlen(buff)-1);
248|                     disp_element(buff,mask);
249|                 }
250|                 *buff++ = c;
251|                 put_char(c);
252|                 do {
253|                     prev_col++;
254|                     mask++;
255|                 } while (*mask && *mask != FieldChar);
256|                 if (!*mask)
257|                     c = FWD;
258|                 break;
259|             }

```

continued...

...from previous page (Listing 7.11)

```

260    if (!*mask)
261        done = TRUE;
262    if (done) {
263        if (type == 'D' &&
264            c != ESC &&
265            validate_date(bfr) != OK)
266            return -1;
267        break;
268    }
269 }
270 if (c != ESC && type != 'A') {
271     if (type == 'C') {
272         if (*mask++ && *buff == ' ')
273             *buff++ = '0';
274         if (*mask++ && *buff == ' ')
275             *buff++ = '0';
276     }
277     if (type == 'Z' || type == 'D')
278         right_justify_zero_fill(bfr);
279     else
280         right_justify(bfr);
281     cursor(column, prev_row);
282     disp_element(bfr,msk);
283 }
284 return (c);
285 }
286 }
287
288 /* ----- right justify, space fill ----- */
289 static void right_justify(char *s)
290 {
291     int len;
292
293     len = strlen(s);
294     while (*s == ' ' || *s == '0' && len) {
295         len--;
296         *s++ = ' ';
297     }

```

continued...

...from previous page (Listing 7.11)

```
298|     if (len)
299|         while (*(s+(len-1)) == ' ') {
300|             memmove(s+1, s, len-1);
301|             *s = ' ';
302|         }
303|     }
304|
305| /* ----- right justify, zero fill ----- */
306| static void right_justify_zero_fill(char *s)
307|
308|     int len;
309|
310|     if (spaces(s))
311|         return;
312|     len = strlen(s);
313|     while (*(s + len - 1) == ' ')
314|         memmove(s+1, s, len-1);
315|         *s = '0';
316|     }
317| }
318|
319| /* ----- test for spaces ----- */
320| int spaces(char *c)
321|
322|     while (*c == ' ')
323|         c++;
324|     return !*c;
325| }
326|
327| /* ----- validate a date (DDMMYY) ----- */
328| static int validate_date(char *s)
329|
330|     static int days_in_month[] =
331|         { 31,28,31,30,31,30,31,31,30,31,30,31 };
332|     char date [7];
333|     int day, month, year;
334|
335|     strcpy(date, s);
```

continued...

...from previous page (Listing 7.11)

```

336|     if (spaces(date))
337|         return OK;
338|     /* ----- extract the year ----- */
339|     year = atoi(date+4);
340|     /* ----- extract the month ----- */
341|     *(date + 4) = '\0';
342|     month = atoi(date+2)-1;
343|     /* ----- extract the day ----- */
344|     *(date+2) = '\0';
345|     day = atoi(date)-1;
346|     /* ----- leap year adjustment ----- */
347|     if ((year % 4) == 0)
348|         days_in_month[1] = 29;
349|     else
350|         days_in_month[1] = 28;
351|     if (month < 12)
352|         if (day < days_in_month [month])
353|             return OK;
354|         error_message("Invalid date");
355|     return ERROR;
356}
357
358/* ---- display all the fields on a screen ----- */
359void tally(void)
360{
361    const ELEMENT *els = elist;
362
363    while (*els)
364        put_field(*els++);
365}
366
367/* ----- Write a data element on the screen ----- */
368void put_field(ELEMENT el)
369{
370    data_coord(el);
371    cursor(prev_col, prev_row);
372    disp_element(bf + epos(el, elist), elmask[el - 1]);
373}

```

continued...

..from previous page (Listing 7.11)

```
374: /* ----- display a data element ----- */
375: static void disp_element(char *b, const char *msk)
376{
377    while (*msk) {
378        int c = *msk != FieldChar ? *msk : *b++;
379        put_char(c);
380        msk++;
381    }
382    cursor(prev_col,prev_row);
383}
384
385
386/* ----- display insert mode status ----- */
387static void insert_status(void)
388{
389    cursor(65,24);
390    printf(insert_mode ? "[INS]" : "      ");
391    cursor(prev_col,prev_row);
392}
393
394/* ----- error message ----- */
395void error_message(char *s)
396{
397    putchar('\a');
398    post_notice(s);
399}
400
401/* ----- clear notice line ----- */
402void clear_notice(void)
403{
404    int i;
405
406    if (notice_posted) {
407        cursor(0,24);
408        for (i = 0; i < 50; i++)
409            putchar(' ');
410        notice_posted = FALSE;
411        cursor(prev_col, prev_row);
412}
```

continued...

...from previous page (Listing 7.11)

```

412    }
413 }
414
415 /* ----- post a notice ----- */
416 void post_notice(char *s)
417 {
418     clear_notice();
419     cursor(0,24);
420     while (*s)
421         putchar(isprint(*s) ? *s : '.');
422     s++;
423 }
424 cursor(prev_col, prev_row);
425 notice_posted = TRUE;
426 }
427
428 /* ----- specific data base error ----- */
429 static void database_error(void)
430 {
431     static char *ers [] = {
432         "Record not found",
433         "No prior record",
434         "End of file",
435         "Beginning of file",
436         "Record already exists",
437         "Not enough memory",
438         "Index corrupted",
439         "Disk i/o error"
440     };
441     error_message(ers [errno-1]);
442 }
443
444 /* ----- test c for an ending keystroke ----- */
445 static int endstroke(int c)
446 {
447     switch (c) {
448         /* ----- implementation-dependent values ----- */
449
450         case F1:      /* Function keys */

```

continued...

..from previous page (Listing 7.11)

```
451|         case F2:
452|         case F3:
453|         case F4:
454|         case F5:
455|         case F6:
456|         case F7:
457|         case F8:
458|         case F9:
459|         case F10:
460|
461|         case PGUP:      /* Page scrolling keys */
462|         case PGDN:
463|         case HOME:
464|         case END:
465|
466|         case UP:        /* Cursor movement keys */
467|         case DN:
468|
469|         case '\r':      /* standard ASCII values */
470|         case '\n':
471|         case '\t':
472|         case ESC:
473|             return TRUE;
474|         default:
475|             return FALSE;
476|     }
477| }
```

The screen functions build temporary screen templates for ad hoc data entry and query programs. The data elements are displayed one per line. See the programs posstime.c and payments.c in Chapter 8 for examples of how applications programs can use the screen functions.

Each function uses a caller-supplied list of data elements and a data entry collection buffer.

Following is a description of the functions in the Screen Management toolset that applications programs can call.

Function init_screen:

```
init_screen(
    char *name,           /* template name */
    const ELEMENT *elist, /* data element list */
    char *bf              /* data entry buffer */
)
```

The **init_screen** function initializes the screen processing software for a particular entry screen format. The name string pointer points to a string that **init_screen** will display at the top of the screen. **Elist** points to a null-terminated array of data element integers. The screen manager uses the array to select the names and entry masks for the template from the database schema. **Bf** points to a data buffer into which the screen manager will collect the data values. The buffer must be long enough to hold all the data elements in the list. Data entry processes will write data elements into the buffer in the sequence of the list.

Function protect:

```
protect(
    ELEMENT el,           /* field data element */
    int tf                /* true to protect */
)
```

The screen manager treats data element fields on the template as protected or unprotected. You can enter data into an unprotected field but not into a protected field. To protect or unprotect a field, you call this function before any data entry

occurs and after you have called **init_screen**. Use the data element global name as the first parameter and either a true value or a false value to tell the function to protect or unprotect the field.

Function edit:

```
edit(
    ELEMENT el,      /* field data element */
    int (*func)()    /* edit function to call */
)
```

You can tell the screen manager to call a custom function when the data entry is complete for a specific data element, which will allow you to validate the entry or to use a key element to retrieve a record from the database. Use the data element global integer name as the first parameter and a pointer to your custom function as the second. Your function must return the integer value OK if the data value is correct or ERROR if it is not. The header file cdata.h defines these values.

Function display_template:

```
display_template()
```

Call this function to display the template on the screen. Since the **data_entry** function described next calls **display_template** automatically, you will need this call only if you have changed the screen display.

Function data_entry:

```
int data_entry()
```

Call this function to allow the user to enter data into the screen template and, therefore, into the buffer you identified when you called **init_screen**. The function will call **display_template** and **tally** (described next) to display the template and any data element values in the buffer before data entry begins.

Function tally:

```
tally()
```

This function displays data values that are in the buffer on the template. You can use this call in your custom edit function. First, retrieve the record that matches a key data element value. Next, move the record's data element values into the screen buffer. Then call **tally**.

Function put_field:

```
put_field(
    ELEMENT el      /* field data element */
)
```

Call **put_field** to display the current value of a specified data element on the screen template. You will use this function to display any data element that you have computed or changed from within your program.

File Listing Utilities

Function dblist:

```
void dblist(
    FILE *fd,           /* output file */
    DBFILE f,           /* data base file number */
    int k,              /* key number */
    const ELEMENT *list /* element list */
)
```

The ds program calls the **dblast** function (dbl.c, Listing 7.12) to prepare a listing, file, or screen display of the contents of the file named on the command line. The **dblast** function manages the production of the output. You can call the same function from within an applications program. When you do, pass **stdprn**, **stdout**, or the **FILE** pointer of an open file as the first parameter. The second parameter is the **DBFILE** file number of the file you want to list. The third parameter is a key number, which determines the sequence of the output. If the key number is zero, the output will be in the physical sequence of the records in the file; otherwise, the key number specifies the primary key (1) or one of the secondary keys (2, 3, and so on), and the output will be in the sequence of the chosen key index. The last parameter is the **ELEMENT** list of data element integers that you want selected from the file.

Listing 7.12 dblist.c

```

1| /* ----- dblist.c ----- */
2| #include <stdio.h>
3| #include <errno.h>
4| #include <stdlib.h>
5| #include "cdata.h"
6|
7| void dblist(FILE *fd, DBFILE f, int k, const ELEMENT *list)
8| {
9|     char *bf;
10|    int rcdct = 0;
11|    if ((bf = malloc(rlen(f))) != NULL) {
12|        errno = 0;
13|        if (k)
14|            first_rcd(f, k, bf);
15|        while (errno != D_EOF) {
16|            if (k) {
17|                clist(fd,file_ele[f],list,bf,dbfiles[f]);
18|                rcdct++;
19|                next_rcd(f, k, bf);
20|            }
21|            else if (seqrcd(f, bf) != ERROR) {
22|                clist(fd,file_ele[f],list,bf,dbfiles[f]);
23|                rcdct++;
24|            }
25|        }
26|        test_eop(fd, dbfiles [f], list);
27|        fprintf(fd, "\nRecords: %d\n", rcdct);
28|        free(bf);
29|    }
30| }

```

Function clist:

```

void clist(
    FILE *fd,           /* output file          */
    const ELEMENT *fl,  /* list of elements in buffer */
    const ELEMENT *pl,  /* list of elements to print */
    char *bf,            /* input buffer address */
    const char *fn      /* file name for report header */
)

```

The **dblist** function, described earlier, calls the **clist** function (*clist.c*, Listing 7.13) to list or display individual records. It can be useful in applications programs as well. The **dblist** function produces a display of a record as described by the **ELEMENT pl** parameter. The function extracts data elements from the buffer pointed to by the **bf** pointer. The **ELEMENT fl** parameter points to a list of the data elements that are in the buffer. The function produces a header line of data element names along with a title taken from the **fn** parameter. It knows whether output is going to the screen or a file or printer, and it issues a page eject at the end of a printed page or pauses at the bottom of a screen.

Listing 7.13 clist.c

```

1| /* ----- clist.c ----- */
2|
3| #include <stdio.h>
4| #include <stdlib.h>
5| #include <ctype.h>
6| #include <string.h>
7| #include "cdata.h"
8| #include "screen.h"
9|
10| #define SCRNLINES 20
11| #define PRNTLINES 55
12|
13| int lct = 99;
14| int clip;
15|
16| static void oflow(FILE *, const char *, const ELEMENT *);
17|
18| /* ----- list a record ----- */
19|
20| /* Send 2 lists; a list of elements in buffer and a list of
21| * elements to be listed. Also send address of the buffer.
22| * Can be used to list a data base file or an extract file.
23| */
24|
25| static int hdlen(ELEMENT el);
26|
27| #define isconsole(fd) (fd == stdout)

```

continued...

...from previous page (Listing 7.13)

```

28|
29| void clist(FILE *fd,const ELEMENT *fl,const ELEMENT *pl,
30|             void *bf,const char *fn)
31|
32|     char *ln, *cp;
33|     const char *mp;
34|     int width;
35|     int lw = 0;
36|     void test_eop();
37|
38|     if ((ln = malloc(epos(0, pl) + 1)) != NULL) {
39|         clip = (isconsole(fd) ? 79 : 136);
40|         test_eop(fd, fn, pl);
41|         rcd_fill(bf, ln, fl, pl);
42|         cp = ln;
43|         if (*pl) {
44|             putc('\n', fd);
45|             putc('\r', fd);
46|             lct++;
47|         }
48|         while (*pl) {
49|             mp = elmask [(*pl) - 1];
50|             width = hdlen(*pl++);
51|             lw += width + 1;
52|             if (lw >= clip)
53|                 break;
54|             while (width--) {
55|                 if (*mp && *mp != '_')
56|                     putc(*mp, fd);
57|                 else if (*cp) {
58|                     putc(isprint(*cp) ? *cp : '?', fd);
59|                     cp++;
60|                 }
61|                 else
62|                     putc(' ', fd);
63|                 if (*mp)
64|                     mp++;
65|             }
}

```

continued...

...from previous page (Listing 7.13)

```

66        if (*pl)
67            putc(' ', fd);
68        cp++;
69    }
70    free(ln);
71 }
72 }
73
74 /* ----- test for end of page/screen ----- */
75 void test_eop(FILE *fd, const char *fn, const ELEMENT *pl)
76 {
77     if (lct >= (isconsole(fd) ? SCRNLINES : PRNTLINES))
78         oflow(fd, fn, pl);
79 }
80
81 /* ----- top of page/screen ----- */
82 static void oflow(FILE *fd, const char *fn, const ELEMENT *pl)
83 {
84     int width;
85     const int *ll;
86     int ow = 0;
87     char msk [80];
88
89     clip = (isconsole(fd) ? 79 : 136);
90     ll = pl;
91     if (lct < 99)  {
92         if (isconsole(fd))  {
93             printf("\n<cr> to continue...");
94             while (getchar() != '\r')
95                 ;
96         }
97         else
98             printf("\r\f");
99     }
100    lct = 0;

```

continued...

...from previous page (Listing 7.13)

```
101 if (isconsole(fd))
102     clear_screen();
103     fprintf(fd, "Filename: %s\n", fn);
104     while (*pl) {
105         width = hdlen(*pl);
106         ow += width + 1;
107         if (ow >= clip)
108             break;
109         sprintf(msk, "%%-%d.%ds ", width, width);
110         fprintf(fd, msk, denames [(*pl++) - 1]);
111     }
112     ow = 0;
113     putc('\n', fd);
114     putc('\r', fd);
115     while (*ll) {
116         width = hdlen(*ll++);
117         ow += width + 1;
118         if (ow >= clip)
119             break;
120         while (width--)
121             putc('-', fd);
122         putc(' ', fd);
123     }
124 }
125
126 static int hdlen(ELEMENT el)
127 {
128     el--;
129     return strlen(elmask [el]) < strlen(denames [el]) ?
130             strlen(denames [el]) :
131             strlen(elmask [el]);
132 }
```

You can use this function to display the result of any kind of data selection as long as all the data elements in the record are in the Cdata data element dictionary.

Parse File Name

Function filename:

```
DBFILE filename(
    char *fn          /* file name to convert */
)
```

Just as you must convert data element names to their integer equivalents, so must you convert file names. The qd, ds, and index utility programs all use the file name as you enter it on the command line when you execute the program. They then call the **filename** function (filename.c, Listing 7.14) to get the file integer that the Cdata DML functions recognize. **Filename** returns the integer if the file name is a valid one and the value **ERROR** if it is not. If you pass an incorrect file name, **filename** will display an error message on the console before it returns.

Listing 7.14 filename.c

```
1| /* ----- filename.c ----- */
2|
3| #include <stdio.h>
4| #include <string.h>
5| #include <ctype.h>
6| #include "cdata.h"
7|
8| /*
9|  * Convert a file name into its file token.
10|  * Return the token,
11|  * or ERROR if the file name is not in the schema.
12|  */
13| DBFILE filename(char *fn)
14| {
15|     char fname[32];
16|     DBFILE f;
17|     void name_cvt();
18| }
```

continued...

..from previous page (Listing 7.14)

```
19|     name_cvt(fname, fn);
20|     for (f = 0; dbfiles [f]; f++)
21|         if (strcmp(fname, dbfiles [f]) == 0)
22|             break;
23|     if (dbfiles [f] == 0)    {
24|         fprintf(stderr, "\nNo such file as %s", fname);
25|         return ERROR;
26|     }
27|     return f;
28| }
29|
30| /* ----- convert a name to upper case ----- */
31| void name_cvt(char *c2, char *c1)
32| {
33|     while (*c1) {
34|         *c2 = toupper(*c1);
35|         c1++;
36|         c2++;
37|     }
38|     *c2 = '\0';
39| }
```

Function name_cvt:

```
int name_cvt(
    char *c2, /* destination for converted string */
    char *c1 /* string to convert */
```

The **name_cvt** function is also in Listing 7.14. It converts a string to uppercase letters. The first parameter is the address of the string for the converted data. The second parameter is the address of the string to be converted. This function is mentioned here because several utility programs use it, and even though the function is small, it is useful.

SORTING

The Cdata sort utility functions support sorting records or subsets of records that your applications programs extract from the database. The sort system is to be used in applications programs that need to display or report data records in sequences other than the ones that the indexes support. The sort functions operate as an in-line sort utility where your programs pass data records to be sorted and then retrieve the records in the specified sorted sequence.

Listing 7.15 is sort.h, the header file that describes the Cdata sort system. The MOSTMEM and LEASTMEM values control memory allocation for sorting. MOSTMEM is the amount the program tries for, and LEASTMEM is the least amount it will accept. As published, the values work within a small-model program on the PC.

Listing 7.15 sort.h

```

1| /* ----- sort.h ----- */
2|
3| #define MOSTMEM 50000U /* most memory for sort buffer      */
4| #define LEASTMEM 10240 /* least memory for sort buffer   */
5| #define MAXSORTS 3    /* maximum fields in a sort sequence */
6|
7| int init_sort(
8|     const ELEMENT *, /* element list of record to be sorted */
9|     const ELEMENT *, /* element list of sort sequence */
10|     int             /* TRUE = ascending sort           */
11| );
12|
13| void sort(void *); /* sort records           */
14| void *sort_op(void); /* retrieve sorted records */

```

The Sort Functions

Function init_sort:

```
int init_sort(
    ELEMENT *list,      /* element list of record to be sorted */
    ELEMENT *seq,       /* element list of sort sequence */
    int direction       /* TRUE = ascending sort */
)
```

You call the **init_sort** function to initialize the sorting process and tell it the format of the record and the data element list to sort. The **list** parameter points to a data element list that describes the record being sorted. The **seq** parameter points to the data element list that describes the sort fields. If the **direction** parameter is TRUE, the sort occurs in ascending sequence. Otherwise it occurs in descending sequence.

If the sort may proceed—if enough memory is available—the function returns zero. Otherwise it returns -1.

Function sort:

```
void sort(
    void *rcd
)
```

You call the **sort** function once for each record, passing a pointer to the record to be sorted. The **sort** function will make a copy of the record and insert it into the sort process. After you have passed the last record, call the function and pass a NULL pointer. This tells the **sort** function to finish the sort and prepare to return sorted records.

Function sort_op:

```
char *sort_op()
```

To retrieve sorted records, call the **sort_op** function. Each call to it returns a pointer to the next record in the sorted sequence. After the last record comes back, the **sort_op** returns a NULL pointer.

The Sort Source Code

Listing 7.16 is sort.c, the program that contains the sorting functions. The first function is `init_sort`, which you call to initiate sorting. It calls `appr_mem` to allocate a block of memory in which to sort, initializes the sorting parameters, and returns.

Listing 7.16 sort.c

```

1| /* ----- sort.c ----- */
2|
3| #include <stdio.h>
4| #include <stdlib.h>
5| #include <string.h>
6| #include "cdata.h"
7| #include "sort.h"
8|
9| static int sort_direction; /* true = ascending sort      */
10| static int rc_len;        /* record length          */
11| static int f_pos[MAXSORTS];
12|
13| static unsigned totrcd;   /* total records sorted    */
14| static int no_seq;        /* counts sequences       */
15| static int no_seq1;
16| static unsigned bspace;   /* available buffer space  */
17| static int nr cds;        /* # of records in sort buffer */
18| static int nr cds1;
19| static char *bf, *bfl;    /* points to sort buffer   */
20| static int inbf;          /* variable records in sort buffer */
21| static char **sptr;       /* -> array of buffer pointers */
22| static char *init_sptr;   /* pointer to appropriated buffer */
23| static int rcds_seq;      /* rcds / sequence in merge buffer */
24| static FILE *fp1, *fp2;    /* sort work file fds      */
25| static char fname[15];    /* sort work name           */
26| static char f2name[15];   /* sort work name           */
27|
28| static int comp(const void *, const void *);
29| static char *appr_mem(unsigned *);
30| static FILE *wopen(char *, int);
31| static void dumpbuff(void);
32| static void merge(void);
33| static void prep_merge(void);

```

continued...

...from previous page (Listing 7.16)

```

34;
35: struct bp { /* one for each sequence in merge buffer */
36:     char *rc; /* -> record in merge buffer */
37:     int rbuf; /* records left in buffer this sequence */
38:     int rdsk; /* records left on disk this sequence */
39: };
40;
41: /* ----- initialize sort global variables----- */
42: int init_sort(
43:     const ELEMENT *list, /* element list of record to be sorted */
44:     const ELEMENT *seq, /* element list of sort sequence */
45:     int direction /* TRUE = ascending sort */
46: )
47: {
48:     char *appr_mem();
49:     int i;
50:
51:     if ((bf = appr_mem(&bspace)) != NULL) {
52:         sort_direction = direction;
53:         rc_len = epos(0, list);
54:         for (i = 0; i < MAXSORTS; i++) {
55:             if (*seq == 0)
56:                 break;
57:             f_pos[i] = epos(*seq++, list);
58:         }
59:         while (i < MAXSORTS)
60:             f_pos[i++] = -1;
61:         nrcds1 = nrcds =
62:             bspace / (rc_len + sizeof(char *));
63:         init_sptr = bf;
64:         sptr = (char **) bf;
65:         bf += nrcds * sizeof(char *);
66:         fp1 = fp2 = NULL;
67:         totrcd = no_seq = inbf = 0;
68:         return 0;
69:     }
70:     else
71:         return -1;

```

continued...

...from previous page (Listing 7.16)

```

72: }
73:
74: /* ----- Function to accept records to sort ----- */
75: void sort(void *s_rcd)
76: {
77:     if (inbf == nrcds) { /* if the sort buffer is full */
78:         qsort(init_sptr, inbf, sizeof (char *), comp);
79:         if (s_rcd) { /* if there are more records to sort */
80:             dumpbuff(); /* dump the buffer to a sort work file */
81:             no_seq++; /* count the sorted sequences */
82:         }
83:     }
84:     if (s_rcd != NULL) {
85:         /* --- this is a record to sort --- */
86:         totrcd++;
87:         /* --- put the rcd addr in the pointer array --- */
88:         *sptr = bf + inbf * rc_len;
89:         inbf++;
90:         /* --- move the rcd to the buffer --- */
91:         memmove(*sptr, s_rcd, rc_len);
92:         sptr++; /* point to next array entry */
93:     }
94:     else { /* null pointer means no more rcds */
95:         if (inbf) { /* any records in the buffer? */
96:             qsort(init_sptr, inbf,
97:                   sizeof (char *), comp);
98:             if (no_seq) /* if this isn't the only sequence */
99:                 dumpbuff(); /* dump the buffer to a work file */
100:                no_seq++; /* count the sequence */
101:            }
102:            no_seq1 = no_seq;
103:            if (no_seq > 1) /* if there is more than 1 sequence */
104:                prep_merge(); /* prepare for the merge */
105:        }
106:    }
107:
108: /* ----- Prepare for the merge ----- */
109: static void prep_merge()

```

continued...

..from previous page (Listing 7.16)

```

110: {
111:     int i;
112:     struct bp *rr;
113:     unsigned n_bfsz;
114:
115:     memset(init_sptr, '\0', bspace);
116:     /* ----- merge buffer size ----- */
117:     n_bfsz = bspace - no_seq * sizeof(struct bp);
118:     /* ----- # rcds/seq in merge buffer ----- */
119:     rcds_seq = n_bfsz / no_seq / rc_len;
120:     if (rcds_seq < 2) {
121:         /* ---- more sequence blocks than will fit in buffer,
122:            merge down ---- */
123:         fp2 = wopen(f2name, 2);      /* open a sort work file */
124:         while (rcds_seq < 2) {
125:             FILE *hd;
126:             merge();                  /* binary merge */
127:             hd = fp1;                 /* swap fds */
128:             fp1 = fp2;
129:             fp2 = hd;
130:             nrcds *= 2;
131:             /* ----- adjust number of sequences ----- */
132:             no_seq = (no_seq + 1) / 2;
133:             n_bfsz = bspace - no_seq * sizeof(struct bp);
134:             rcds_seq = n_bfsz / no_seq / rc_len;
135:         }
136:     }
137:     bfl = init_sptr;
138:     rr = (struct bp *) init_sptr;
139:     bfl += no_seq * sizeof(struct bp);
140:     bf = bfl;
141:
142:     /* fill the merge buffer with records from all sequences */
143:
144:     for (i = 0; i < no_seq; i++) {
145:         fseek(fp1, (long) i * ((long) nrcds * rc_len),
146:               SEEK_SET);
147:         /* ----- read them all at once ----- */
148:         fread(bf1, rcds_seq * rc_len, 1, fp1);
149:         rr->rc = bf1;
150:         /* --- the last seq has fewer rcds than the rest --- */

```

continued...

...from previous page (Listing 7.16)

```

151|     if (i == no_seq-1) {
152|         if (totrcd % nrcds > rcds_seq) {
153|             rr->rbuf = rcds_seq;
154|             rr->rdsk = (totrcd % nrcds) - rcds_seq;
155|         }
156|         else {
157|             rr->rbuf = totrcd % nrcds;
158|             rr->rdsk = 0;
159|         }
160|     }
161|     else {
162|         rr->rbuf = rcds_seq;
163|         rr->rdsk = nrcds - rcds_seq;
164|     }
165|     rr++;
166|     bfl += rcds_seq * rc_len;
167| }
168|
169|
170| /* ----- Merge the work file down
171|    This is a binary merge of records from sequences
172|    in fp1 into fp2. ----- */
173| static void merge()
174| {
175|     int i;
176|     int needy, needx; /* true = need a rcd from (x/y) */
177|     int xcnt, ycnt; /* # rcds left each sequence */
178|     int x, y; /* sequence counters */
179|     long adx, ady; /* sequence record disk addresses */
180|
181|     /* --- the two sets of sequences are x and y ----- */
182|     fseek(fp2, 0L, SEEK_SET);
183|     for (i = 0; i < no_seq; i += 2) {
184|         x = y = i;
185|         y++;

```

continued...

...from previous page (Listing 7.16)

```

186      ycnt =
187      y == no_seq ? 0 : y == no_seq - 1 ?
188          totrcd % nrcds : nrcds;
189      xcnt = y == no_seq ? totrcd % nrcds : nrcds;
190      adx = (long) x * (long) nrcds * rc_len;
191      ady = adx + (long) nrcds * rc_len;
192      needy = needx = 1;
193      while (xcnt || ycnt) {
194          if (needx && xcnt) { /* need a rcd from x? */
195              fseek(fp1, adx, SEEK_SET);
196              adx += (long) rc_len;
197              fread(init_sptr, rc_len, 1, fp1);
198              needx = 0;
199          }
200          if (needy && ycnt) { /* need a rcd from y? */
201              fseek(fp1, ady, SEEK_SET);
202              ady += rc_len;
203              fread(init_sptr+rc_len, rc_len, 1, fp1);
204              needy = 0;
205          }
206          if (xcnt || ycnt) { /* if anything is left */
207              /* ---- compare the two sequences --- */
208              if (!ycnt || (xcnt &&
209                  (comp(&init_sptr, &init_sptr + rc_len))
210                  < 0)) {
211                  /* ----- record from x is lower ----- */
212                  fwrite(init_sptr, rc_len, 1, fp2);
213                  --xcnt;
214                  needx = 1;
215              }
216              else if (ycnt) { /* record from y is lower */
217                  fwrite(init_sptr+rc_len,
218                         rc_len, 1, fp2);
219                  --ycnt;
220                  needy = 1;
221              }
222          }
223      }

```

continued...

...from previous page (Listing 7.16)

```

224    }
225  }
226
227  /* ----- Dump the sort buffer to the work file ----- */
228  static void dumpbuff()
229  {
230    int i;
231
232    if (fp1 == NULL)
233      fp1 = wopen(fdname, 1);
234    sptr = (char **) init_sptr;
235    for (i = 0; i < inbf; i++) {
236      fwrite(*(sptr + i), rc_len, 1, fp1);
237      *(sptr + i) = 0;
238    }
239    inbf = 0;
240  }
241
242  /* ----- Open a sort work file ----- */
243  static FILE *wopen(char *name, int n)
244  {
245    FILE *fp;
246    strcpy(name, "sortwork.000");
247    name[strlen(name) - 1] += n;
248    if ((fp = fopen(name, "wb+")) == NULL) {
249      fprintf(stderr, "\nFile error");
250      exit(1);
251    }
252    return fp;
253  }
254
255  /* ----- Function to get sorted records ----- */
256  This is called to get sorted records after the sort is done.
257  It returns pointers to each sorted record.
258  Each call to it returns one record.
259  When there are no more records, it returns NULL. ----- */
260
261  void *sort_op(void)

```

continued...

...from previous page (Listing 7.16)

```

262: {
263:     int j = 0;
264:     int nrd, i, k, l;
265:     struct bp *rr;
266:     static int r1 = 0;
267:     char *rtn;
268:     long ad, tr;
269:
270:     sptr = (char **) init_sptr;
271:     if (no_seq < 2) {
272:         /* -- with only 1 sequence, no merge has been done -- */
273:         if (r1 == totrcd) {
274:             free(init_sptr);
275:             fp1 = fp2 = NULL;
276:             r1 = 0;
277:             return NULL;
278:         }
279:         return *(sptr + r1++);
280:     }
281:     rr = (struct bp *) init_sptr;
282:     for (i = 0; i < no_seq; i++)
283:         j |= (rr + i)->rbuf | (rr + i)->rdsk;
284:
285:     /* -- j will be true if any sequence still has records - */
286:     if (!j) {
287:         fclose(fp1);           /* none left */
288:         remove(fdname);
289:         if (fp2) {
290:             fclose(fp2);
291:             remove(f2name);
292:         }
293:         free(init_sptr);
294:         fp1 = fp2 = NULL;
295:         r1 = 0;
296:         return NULL;
297:     }
298:     k = 0;
299:
300:     /* --- find the sequence in the merge buffer

```

continued...

...from previous page (Listing 7.16)

```

301|           with the lowest record --- */
302|   for (i = 0; i < no_seq; i++)
303|     k = ((comp( &(rr + k)->rc, &(rr + i)->rc) < 0) ? k : i);
304|
305| /* --- k is an integer sequence number that offsets to the
306|    sequence with the lowest record ---- */
307|
308|   (rr + k)->rbuf--;           /* decrement the rcd counter */
309|   rtn = (rr + k)->rc;          /* set the return pointer */
310|   (rr + k)->rc += rc_len;
311|   if ((rr + k)->rbuf == 0) {
312|     /* ---- the sequence got empty ---- */
313|     /* --- so get some more if there are any --- */
314|     rtn = bf + k * rcds_seq * rc_len;
315|     memmove(rtn, (rr + k)->rc - rc_len, rc_len);
316|     (rr + k)->rc = rtn + rc_len;
317|     if ((rr + k)->rdsk != 0) {
318|       l = ((rcds_seq-1) < (rr+k)->rdsk) ?
319|         rcds_seq-1 : (rr+k)->rdsk;
320|       nrd = k == no_seq - 1 ? totrcd % rcds : rcds;
321|       tr = (long) ((k * rcds + (nrd - (rr + k)->rdsk)));
322|       ad = tr * rc_len;
323|       fseek(fp1, ad, SEEK_SET);
324|       fread(rtn + rc_len, l * rc_len, 1, fp1);
325|       (rr + k)->rbuf = l;
326|       (rr + k)->rdsk -= l;
327|     }
328|     else
329|       memset((rr + k)->rc, 127, rc_len);
330|   }
331|   return rtn;
332| }
333|
334| /* ----- appropriate available memory ----- */
335| static char *appr_mem(unsigned *h)
336| {
337|   char *buff = NULL;
338|
339|   *h = (unsigned) MOSTMEM + 1024;
340|   while (buff == NULL && *h > LEASTMEM) {
341|     *h -= 1024;
342|     buff = malloc(*h);
343|   }

```

continued...

...from previous page (Listing 7.16)

```
344|     return buff;
345| }
346|
347| /* ----- compare function for sorting, merging ----- */
348| static int comp(const void *a, const void *b)
349|
350| {
351|     int i, k;
352|
353|     if (**(char **)a == 127 || **(char **)b == 127)
354|         return (int) **(char **)a - (int) **(char **)b;
355|     for (i = 0; i < MAXSORTS; i++) {
356|         if (f_pos[i] == -1)
357|             break;
358|         if ((k = strcmp((*(char **)a)+f_pos[i],
359|                         (*(char **)b)+f_pos[i]))) != 0)
360|             return (isort_direction) ? -k : k;
361|     }
362| }
```

The **sort** function accepts records to sort from the calling application. At first it simply copies them into the buffer, one after the other. When the buffer is full, the **sort** function calls the standard **qsort** function to sort the records in the buffer. This becomes a block of sorted records. The **dumpbuff** function writes them to the sort work file. The sort work file will contain multiple blocks of sorted records.

When the calling application sends a NULL pointer to the **sort** function, it sorts the final block, writes it to the sort work file, and calls **prep_merge** to prepare for when the caller wants sorted records. The merge divides the sort work buffer into many smaller buffers, one for each sequence that was generated by the **sort** function. These buffers contain blocks of two or, usually, more records. If and while the number of blocks is high enough to prevent the buffer from being segmented this way, **prep_merge** uses the **merge** function to merge groups of two blocks into one, halving the number of blocks and doubling their sizes.

When the number of blocks is low enough that each one can contribute at least two records in the sort buffer, the **merge** and **prep_merge** functions are done.

The calling application calls **sort_op** to get sorted records. The **sort_op** function looks at the first record in each buffer segment to find the lowest record. It bumps that segment's record pointer and returns a pointer to the record it found. When it exhausts a buffer segment, it reads more records from the associated sequence on the sort work file. When all the segments are empty, the application has read the last sorted record, and the sort program signals that it is done by returning a NULL pointer.

The **comp** function compares records for the sort system. It follows the format of the compare function for the standard C **qsort** function. It uses the data element list that the using application sent to the **init_sort** function to determine what to compare. It uses functions from the Cdata library to compute the position in the buffer of each data element. The **comp** function does a straight case-insensitive string compare. The **strcmp** function that it uses is not defined by Standard C, but all the compilers include it.

A SORTING EXAMPLE: THE CLIENT ROSTER

Listing 7.17 is roster.c, a program that reads the records from the CLIENTS file, uses the Cdata sort system to sort the records into client name sequence, and writes a report showing all clients and their phone numbers. This program, like invoice.c in Chapter 6, is one of the four custom application programs for the Consultant's Billing System.

Listing 7.17 roster.c

```
1: /* ----- roster.c ----- */
2:
3: /*
4:  * Sort clients and print a roster
5:  */
6:
7: #include <stdio.h>
8: #include <stdlib.h>
9: #include "cbs.h"
10: #include "sort.h"
11:
12: struct clients cl;
13:
14: void main(void)
15: {
16:     static DBFILE fl[] = [CLIENTS, -1];
17:
18:     static const ELEMENT els[] = [CLIENT_NAME, 0];
19:     static const ELEMENT elp[] = [CLIENT_NAME, PHONE, 0];
20:     struct clients *cls;
21:
22:     db_open("", fl);
23:     init_sort(file_ele[CLIENTS],els,TRUE);
24:     while (TRUE)    {
25:         if (next_rcd(CLIENTS, 1, &cl) == ERROR)
26:             break;
27:         sort(&cl);
28:     }
29:     sort(NULL);
30:     while ((cls = sort_op()) != NULL)
31:         clist(stdout, file_ele[CLIENTS], elp, cls, "Client Roster");
32:     db_cls();
33: }
```

SUMMARY

You have now finished building the Cdata database management system, complete with data definition language, data manipulation language, and utility functions. The software described in these chapters is real; it is used in many installations. Large, complex applications systems have been built with no more tools than these programs, an editor, and a compiler. To illustrate the flexibility and usability of this approach, Chapter 8 presents a small application, the Consultant's Billing System. This simple but functional application uses the database design presented in this book for examples of the Cdata approach.

Chapter 8

An Application: The Consultant's Billing System

This chapter presents the primary objective of this book: to develop a software system to support an application. This exercise involves using the database designed in earlier chapters—a database that supports a billing system for a consulting firm.

Begin by considering the requirements for such a system. In actual practice, the requirements analysis is the first step in the design of a computer system. This book postpones the requirements discussion until this chapter—after the discussion of the software that surrounds the database is completed.

CONSULTANT'S BILLING SYSTEM REQUIREMENTS

A Consultant's Billing System must maintain records of consultants, clients, and projects. Since consultants work on and charge time to the projects of clients, the system will relate consultants to projects and projects to clients. A consultant might charge different rates for different projects, so the system records a rate for each consultant-to-project assignment. The system also provides for hourly charges and other direct expenses against a project, posts payments when they occur, and prepares invoices.

CONSULTANT'S BILLING SYSTEM SOFTWARE DESIGN

Figure 8-1 shows the structure chart of a likely Consultant's Billing System. This system seems simple. The database design in the previous chapters is sufficient to support the requirements of the Consultant's Billing System; all it needs are some programs to process the data. With the Cdata DBMS, you already have most of the programs written. The Consultant's Billing System uses the utility programs from Chapter 7, the Cdata functions from Chapter 6, and three custom applications programs. There are only four new programs, and two of them were written in Chapter 6—the invoice program used to illustrate how you build applications programs around Cdata and the roster program that illustrated the sort system.. To understand how a complete application can be built with only four custom programs, consider the individual processes a user of a Consultant's Billing System will require.

Consultant's Billing System Data Entry

To enter data into the Consultant's Billing System, you need a database. At first, the database is empty; it has no records. Before you can begin, you must build the empty database. The dbinit utility program initializes a database that you describe in the Cdata format. That program builds an empty database. Then, you need a way to enter data into each of the files. You must add clients, consultants, and projects to the database, and you must also assign consultants to projects. Occasionally, you will want to change the data values of existing records or delete records from the database. You want to do what the qd utility program does; therefore, most of your data entry requirements are supported by this existing program, the qd utility.

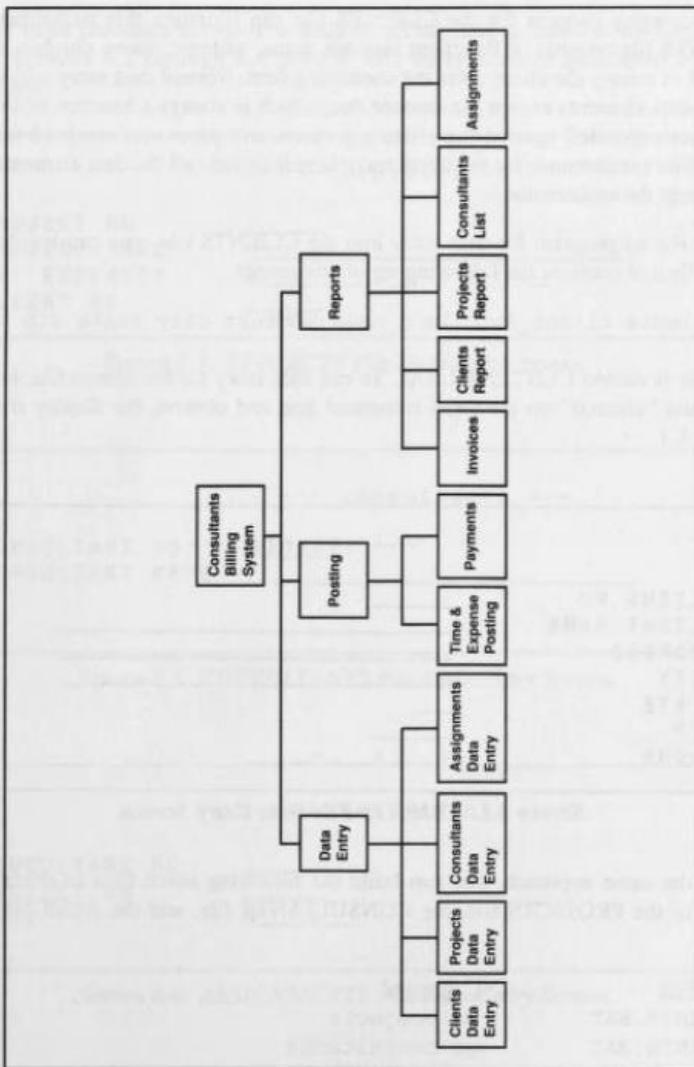


Figure 8-1. Consultant's Billing System Structure Chart.

8 An Application: The Consultant's Billing System

The data entry process for the CLIENTS file can illustrate this technique. Each CLIENTS file consists of the client number, name, address, phone number, and the amount of money the client owes the consulting firm. Normal data entry will support all the data elements except the amount due, which is always a function of time and expenses expended against the client's projects and payments received from the client. File maintenance for the client records will include all the data elements in the file except the amount due.

To use the *qd* program for data entry into the CLIENTS file, you can build a DOS batch file that contains the following set of commands:

```
qd clients client_no client_name address city state zip phone
```

This file is named CLNTENTR.BAT. To run data entry for the clients file, enter the command "clntentr" on the DOS command line and observe the display shown in Screen 8.1.

--- clients ---	
CLIENT NO	_____
CLIENT NAME	_____
ADDRESS	_____
CITY	_____
STATE	_____
ZIP	_____
PHONE	(____) ____ - ____

Screen 8.1. CLIENTS File Data Entry Screen.

Using the same approach, you can build the following batch files to execute data entry for the PROJECTS file, the CONSULTANTS file, and the ASSIGNMENTS file:

<u>Batch File</u>	<u>Command</u>
PROJENTR.BAT	qd projects
CONSENTR.BAT	qd consultants
ASGNENTR.BAT	qd assignments

Each of these processes allows data entry on all the data elements in the file that qd reads. Screens 8.2 through 8.4 are the data entry screens generated by these commands:

--- projects ---	
PROJECT NO	_____
PROJECT NAME	_____
AMT EXPENDED	\$ _____ .__
CLIENT NO	_____

Screen 8.2. PROJECTS File Data Entry Screen.

--- consultants ---	
CONSULTANT NO	_____
CONSULTANT NAME	_____

Screen 8.3. CONSULTANTS File Data Entry Screen.

--- assignments ---	
CONSULTANT NO	_____
PROJECT NO	_____
RATE	\$ ____ .__

Screen 8.4. ASSIGNMENTS File Data Entry Screen.

Consultant's Billing System Reports

Just as the qd utility program manages routine data entry into the Consultant's Billing System, the ds utility program supports most of the reporting requirements. The Consultant's Billing System contains five basic reports. The first four are reports for the contents of the four files and the last is a report for the preparation of invoices. The discussion of the invoice program will be deferred until later; for now, concentrate on the listings of the file contents. Each of these listings is produced by the ds utility program. By using the batch files shown here, you can produce the reports shown in Listings 8.1 through 8.4:

<u>Batch File</u>	<u>Command</u>
CLNREPT.BAT	ds clients client_no client_name phone amt_due >pm
PROJREPT.BAT	ds projects >prn
CONSREPT.BAT	ds consultants >prn
ASGNREPT.BAT	ds assignments >prn

Listing 8.1 Client Report

Filename: CLIENTS		PHONE	AMT_DUE
CLIENT_NO	CLIENT_NAME		
00001	Smith Engineering	(703) 555-6453	\$ 1980.00
00002	Jones Dairy	(703) 555-2345	\$ 500.00
00003	Harry's Auto Mart	(703) 555-9876	\$.
Records: 3			

Listing 8.2 Project Report

PROJECT_NO	PROJECT_NAME	AMT_EXPENDED	CLIENT_NO
00001	Payroll Analysis	\$ 2500.00	00001
00002	General Ledger	\$ 1480.00	00001
00003	Inventory System	\$ 1000.00	00002
00004	Hardware Market Survey	\$.	00002
Records: 4			

Listing 8.3 Consultant Report

```
Filename: CONSULTANTS
CONSULTANT_NO CONSULTANT_NAME
```

CONSULTANT_NO	CONSULTANT_NAME
00001	George Martin
00002	Sam Spade
00003	Phil Napolean
00004	Barney Fife

Records: 4

Listing 8.4 Assignments Report

```
Filename: ASSIGNMENTS
CONSULTANT_NO PROJECT_NO RATE
```

CONSULTANT_NO	PROJECT_NO	RATE
00001	00001	\$ 40.00
00001	00002	\$ 40.00
00001	00004	\$ 40.00
00002	00001	\$ 35.00
00002	00004	\$ 35.00
00003	00002	\$ 37.00
00003	00003	\$ 37.00
00004	00003	\$ 25.00

Records: 8

Notice that you direct the reports to the printer by using the input/output redirection facility of DOS. You can display these same reports on the screen by eliminating the expression ">pm" from each of the commands.

The Data Posting Programs

Three kinds of data values are posted to the Consultant's Billing System database. You distinguish data entry (discussed earlier) and data posting in this manner; data entry involves the nonroutine maintenance of data in the database, and data posting implies routine transactions that support the purpose of the system. When you change a client's phone number, that is data entry. When you add a consultant to the file, that is data entry. Data entry appends data values to the database or replaces data

8 An Application: The Consultant's Billing System

values that already exist. Data posting adjusts existing data values in the database, often with addition or subtraction. The three data values to be posted to the database are listed here:

- Labor hours charged to projects
- Expenses charged to projects
- Payments received from clients

These processes are beyond the abilities of the Cdata utility programs and require algorithms not implied by the schema, so you need custom programs to support them. A program named posttime.c (Listing 8.5) manages the posting of hours and expenses. This program illustrates the use of multiple files in a database. When you enter a consultant number, the program reads the CONSULTANTS file to find the consultant's name. Then, when you enter a project number, the program retrieves the project name from the PROJECTS file. The program displays each of these names just after retrieving it. If you have entered a valid consultant and a valid project, the program reads the ASSIGNMENTS file to determine if the consultant is assigned to the project. If not, an error message appears; otherwise, the program retrieves the hourly rate for the selected consultant on the selected project. After you have entered hours and expenses, the program uses the client number in the projects file to find the associated CLIENTS record. The amount to charge is computed as:

$$(\text{hours} \times \text{rate} + \text{expenses})$$

This amount is added to the amount due in the client record and the amount expended in the project record.

Listing 8.5 posttime.c

```

1| /* ----- posttime.c ----- */
2|
3| #include <stdio.h>
4| #include <stdlib.h>
5| #include <string.h>
6| #include "cbs.h"
7| #include "screen.h"
8| #include "keys.h"
9|
10| static char *sc;
11| static DBFILE fl[] =
12|     {CLIENTS, CONSULTANTS, PROJECTS, ASSIGNMENTS, -1};
13| static ELEMENT els[] =
14|     {CONSULTANT_NO, CONSULTANT_NAME,
15|      PROJECT_NO, PROJECT_NAME, HOURS, EXPENSE, 0};
16| struct clients cl;
17| struct consultants cs;
18| struct projects pr;
19| struct assignments as;
20|
21| void main(void)
22| {
23|     int term = '\0';
24|     int edcons(), edproj();
25|     long atol();
26|     long exp, rt, time_chg;
27|     int hrs;
28|
29|     db_open("", fl);
30|     sc = malloc(epos(0, els));
31|     clrrcd(sc, els);
32|     init_screen("Time & Expenses", els, sc);
33|     edit(CONSULTANT_NO, edcons); /* consultant number */
34|     protect(CONSULTANT_NAME, TRUE); /* consultant name */
35|     edit(PROJECT_NO, edproj); /* project number */
36|     protect(PROJECT_NAME, TRUE); /* project name */
37|     while (term != ESC) {
38|         term = data_entry();

```

continued...

8 An Application: The Consultant's Billing System

..from previous page (Listing 8.5)

```
39|         switch (term)  {
40|             /* ----- GO ----- */
41|             case F1:
42|                 hrs = atoi(sc + epos(HOURS, els));
43|                 exp = atol(sc + epos(EXPENSE, els));
44|                 rt = atol(as.rate);
45|                 time_chg = rt * hrs;
46|                 clrrcd(sc, els);
47|                 sprintf(cl.amt_due, "%8ld",
48|                         atol(cl.amt_due)+time_chg+exp);
49|                 rtn_rcd(CLIENTS, &cl);
50|                 sprintf(pr.amt_expended, "%9ld",
51|                         atol(pr.amt_expended)+time_chg+exp);
52|                 rtn_rcd(PROJECTS, &pr);
53|                 break;
54|             case ESC:
55|                 if (spaces(sc))
56|                     break;
57|                 clrrcd(sc, els);
58|                 term = '\0';
59|                 break;
60|             default:
61|                 break;
62|         }
63|     }
64|     clear_screen();
65|     free(sc);
66|     db_cls();
67| }
68|
69| /* ----- consultant number ----- */
70| int edcons(char *s)
71| {
72|     if (find_rcd(CONSULTANTS, 1, s, &cs) == ERROR)  {
73|         dberror();
74|         return ERROR;
75|     }
76|     rcd_fill(&cs, sc, file_ele[CONSULTANTS], els);
```

continued...

...from previous page (Listing 8.5)

```
77|     put_field(CONSULTANT_NAME);
78|     return OK;
79| }
80|
81| /* ----- project number ----- */
82| int edproj(char *s)
83| {
84|     char consproj[11];
85|
86|     if (find_rcd(PROJECTS, 1, s, &pr) == ERROR) {
87|         dberror();
88|         return ERROR;
89|     }
90|     rcd_fill(&pr, sc, file_ele[PROJECTS], els);
91|     put_field(PROJECT_NAME);
92|     strcpy(consproj, cs.consultant_no);
93|     strcat(consproj, pr.project_no);
94|     if (find_rcd(ASSIGNMENTS, 1, consproj, &as) == ERROR) {
95|         error_message("Consultant not assigned to project");
96|         return ERROR;
97|     }
98|     find_rcd(CLIENTS, 1, pr.client_no, &c1);
99|     return OK;
100| }
```

The payments.c program, Listing 8.6, posts payments to the CLIENTS file. It retrieves the client record by using the client number that you enter. The payment amount is compared to the amount due. If the payment is greater than the amount due, an overpayment error message appears; otherwise, the program subtracts the payment amount from the amount due in the client record.

Listing 8.6 payments.c

```

1  /* ----- payments.c ----- */
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <errno.h>
6  #include "cbs.h"
7  #include "screen.h"
8  #include "keys.h"
9
10 char *sc;
11 static DBFILE fl[] = {CLIENTS, -1};
12 static ELEMENT els[] =
13     {CLIENT_NO, CLIENT_NAME, AMT_DUE, PAYMENT, DATE_PAID, 0};
14 struct clients cl;
15 int len;
16
17 void main(void)
18 {
19     int term = '\0';
20     int edcl();
21     long atol();
22
23     db_open("", fl);
24     len = epos(0, els);
25     sc = malloc(len);
26     clrrcd(sc, els);
27     init_screen("Payments", els, sc);
28     edit(CLIENT_NO, edcl);
29     protect(CLIENT_NAME, TRUE);
30     protect(AMT_DUE, TRUE);
31     while (term != ESC) {
32         term = data_entry();
33         switch (term) {
34             case F1:
35                 if (atol(cl.amt_due) <
36                     atol(sc + epos(PAYMENT, els))) {
37                     error_message("Overpayment");
38                     continue;
39                 }

```

continued...

..from previous page (Listing 8.6)

```

40:         sprintf(cl.amt_due,"%8ld",
41:             atol(cl.amt_due)
42:             -atol(sc+epos(PAYMENT,els)));
43:         clrrcd(sc, els);
44:         rtn_rcd(CLIENTS, &cl);
45:         break;
46:     case ESC:
47:         if (spaces(sc))
48:             break;
49:         clrrcd(sc, els);
50:         term = '\0';
51:         break;
52:     default:
53:         break;
54:     }
55: }
56: clear_screen();
57: free(sc);
58: db_cls();
59: }
60:
61: /* ----- client number ----- */
62: int edcl(char *s)
63: {
64:     if (find_rcd(CLIENTS, 1, s, &cl) == ERROR) {
65:         dberror();
66:         return ERROR;
67:     }
68:     rcd_fill(&cl, sc, file_ele[CLIENTS], els);
69:     tally();
70:     return OK;
71: }
```

The Invoice Program

This last program completes the Consultant's Billing System and generates invoices for clients. The invoice.c program (with its source code) is in Chapter 6 as an example of applications programs that use a Cdata database. When you type its name on the command line, the program prepares statements showing the current amount due from each client. Listing 8.7 is an example of the statements produced by the invoice program.

Listing 8.7 Invoice Statements

Invoice for Services Rendered
Smith Engineering
1212 Mecca Ave
Vienna , VA 22101

Amount Due: \$1980.00

Invoice for Services Rendered

Jones Dairy
Bristow , VA 22111

Amount Due: \$500.00

Invoice for Services Rendered

Harry's Auto Mart
123 King St
Alexandria , VA 22333

Amount Due: \$0.00

SUMMARY

The Consultant's Billing System is not a complete accounting system. It lacks the accounting detail and control mechanisms that a medium-sized or large consulting firm requires for adequate records maintenance. It works for a small consulting firm that limits its practice to a few clients and projects. It does, however, provide a compact example of the use of the Cdata approach to database management. Do not let this small example lead you to think that Cdata is applicable only to small or simple applications. Cdata is routinely used in much larger and more complex systems.

Chapter 9

Building the Software

Chapter 9 explains how you can build the software published in this book by using one of the compilers. The procedure for each compiler accompanies an appropriate **makefile** to build the toolset functions and the example applications.

PREPARING YOUR SYSTEM

These programs will compile and run on an IBM PC or compatible with one of the five compilers discussed in this chapter. Because the code is Standard C, the programs should port to other environments with a minimum of modification. The code does not use display or hardware characteristics that are unique to the PC.

There are a few things that you must consider when building and running these programs. The next paragraphs address these considerations.

ANSI.SYS

The programs use the ANSI standard terminal commands to clear the screen and position the cursor. In a PC, you must have the ANSI.SYS device driver installed. Otherwise the screen will display jibberish character sequences with little arrows and strings such as “[2J,” and the data displays will not appear in the correct positions on the screen.

The ANSI.SYS device driver comes with DOS. You must, however, tell DOS to include it when you boot your computer. To do this, you add an entry to a file named CONFIG.SYS that must appear in the root directory of the disk drive from which you boot DOS. The CONFIG.SYS file must contain this statement:

```
DEVICE=ANSI.SYS
```

Copy the ANSI.SYS file from your DOS directory or the DOS distribution diskette into the root directory as well. You could leave the file in the DOS directory and make the CONFIG.SYS entry something like this:

```
DEVICE=C:\DOS\ANSI.SYS
```

File Handles

DOS has a default limit of eight file handles. That means that only eight files can be open at one time unless you change the default. Most software installation procedures—including those of the C compilers discussed below—tell you to increase the limit to at least 20 by adding this entry to CONFIG.SYS:

```
FILES=20
```

That value increases the maximum number of files that can be open at one time to twenty. The highest value you can assign is 255, but that value does not automatically accrue to programs. The value in the FILES= entry is the maximum number of open files that DOS can maintain. Individual programs have a limit of twenty regardless of the FILES= value.

The number of files that can be open is a function of the number of DOS file handles that a program can own. The structure for a normal DOS program includes room for only twenty handles. Five of these are assigned to `stdin`, `stdout`, `stderr`, `stdprn`, and `stdaux`, so the typical program can really have only fifteen handles.

Fifteen open files would seem to be enough until you consider the structure of a database. A database can have multiple data files, and each data file can have multiple index files. With the DOS limit of twenty files open per program, and five of them dedicated to standard DOS files, a database management program could run out of handles if it tried to open all the data files and index files in the database. You can control this problem from your applications program by specifying fewer DBFILE entries in your call to the `db_open` function. If an application needs to gain access to more data and index files than the twenty-file limit will handle, you can do one of two things. You can alternately open and close subsets of the database, or you can use a feature of DOS 3.3 and greater that allows you to increase the maximum number of file handles that a program can have. A DOS function call exists in 3.3 and greater that will increase the number of file handles. Using the features of your compiler that allow you to call DOS, put the value `0x67` in the `AH` register, the number of handles you want in the `BX` register, and perform an interrupt to vector `0x20`, the DOS interrupt.

If you want to increase a program's maximum number of handles for versions of DOS prior to 3.3, read *Extending Turbo C Professional*, Al Stevens, 1989, MIS Press. The book explains a trick for doing it and provides the necessary C code.

THE COMPILERS

This section discusses the five compilers that will compile the programs in this book. The fact that the code ports between these compilers without changes gives it a good chance to be compatible with other Standard C compilers on other computers.

You must have installed your compiler successfully with all the installation options in place. The compiler must be in the DOS path, and, for some compilers, you must set `INCLUDE`, `LIB`, and other DOS environment variables.

Microsoft C 5.1

Listing 9.1 is makefile.msc, the file that the Microsoft Make utility program uses to build projects. The command line is:

```
make makefile.msc
```

If you are using version 6.0 or greater of Microsoft C, the Nmake utility program is different than the earlier Make. Use this command line:

```
nmake /MAKE /f makefile.msc
```

The /MAKE option causes Nmake to emulate the behavior of the earlier Make. Because many programmers still use and swear by Microsoft C 5.1, this book uses the format that works with that version.

Listing 9.1 makefile.msc

```
1| # -----
2| # Microsoft C 5.1 MAKEFILE to build Cdata into an application
3| # -----
4|
5| # -----
6| # Application Name
7| # -----
8| CDATA_APPL    = cbs
9|
10| #
11| # An implicit rule to build .OBJ files from .C files
12| # -----
13|
14| .c.obj:
15|     cl -c $*.c
16|
17| #
18| # An implicit rule to build .EXE files from .OBJ files
19| # -----
20|
21| .obj.exe:
22|     link $(STARTUP) $* $(CDATA_APPL).obj, $*,,,cdata;
23|
```

continued...

..from previous page (Listing 9.1)

```
24| # -----
25| # CDATA.LIB
26| # -----
27|
28| ellist.obj : ellist.c cdata.h
29|
30| cdata.obj : cdata.c cdata.h datafile.h btree.h keys.h
31|
32| screen.obj : screen.c cdata.h screen.h keys.h
33|
34| btree.obj : btree.c cdata.h btree.h
35|
36| datafile.obj : datafile.c cdata.h datafile.h
37|
38| dblist.obj : dblist.c cdata.h
39|
40| clist.obj : clist.c cdata.h screen.h
41|
42| sort.obj : sort.c cdata.h sort.h
43|
44| sys.obj : sys.c cdata.h sys.h
45|
46| filename.obj : filename.c cdata.h
47|
48| cdata.lib : ellist.obj \
49|             cdata.obj \
50|             screen.obj \
51|             btree.obj \
52|             datafile.obj \
53|             dblist.obj \
54|             clist.obj \
55|             filename.obj \
56|             sort.obj \
57|             sys.obj
58|     del cdata.lib
59|     lib cdata @cdata.bld
60|
61| # -----
62| # The Data Base Schema
63| # -----
64| 
```

continued...

9 Building the Software

...from previous page (Listing 9.1)

```
65: schema.obj    : schema.c cdata.h
66:
67: schema.exe    : schema.obj
68:           link $(STARTUP) $*, $*,,,cdata;
69:
70: $(CDATA_APPL).c : $(CDATA_APPL).sch schema.exe
71:           schema $(CDATA_APPL)
72:
73: $(CDATA_APPL).obj : $(CDATA_APPL).c
74:
75: # -----
76: # Cdata Utility Programs
77: # -----
78:
79: qd.obj        : qd.c cdata.h screen.h keys.h
80:
81: ds.obj        : ds.c cdata.h
82:
83: index.obj     : index.c cdata.h
84:
85: dbsize.obj    : dbsize.c cdata.h btree.h datafile.h
86:
87: dbinit.obj    : dbinit.c cdata.h datafile.h
88:
89: qd.exe        : qd.obj $(CDATA_APPL).obj cdata.lib
90:
91: ds.exe        : ds.obj $(CDATA_APPL).obj cdata.lib
92:
93: index.exe     : index.obj $(CDATA_APPL).obj cdata.lib
94:
95: dbsize.exe    : dbsize.obj $(CDATA_APPL).obj cdata.lib
96:
97: dbinit.exe    : dbinit.obj $(CDATA_APPL).obj cdata.lib
98:
99: # -----
100: # Application-specific (CBS) programs
```

continued...

...from previous page (Listing 9.1)

```

101; # -----
102;
103; posttime.obj : posttime.c cbs.h screen.h keys.h
104;
105; payments.obj : payments.c cbs.h screen.h keys.h
106;
107; invoice.obj : invoice.c cbs.h
108;
109; roster.obj : roster.c cbs.h sort.h
110;
111; posttime.exe : posttime.obj cbs.obj cdata.lib
112;
113; payments.exe : payments.obj cbs.obj cdata.lib
114;
115; invoice.exe : invoice.obj cbs.obj cdata.lib
116;
117; roster.exe : roster.obj cbs.obj cdata.lib

```

Listing 9.2 is cdata.bld, the command file that the Microsoft Lib program uses to build the cdata.lib object library.

Listing 9.2 cdata.bld

```

1; +ellist.obj      &
2; +cdata.obj       &
3; +screen.obj      &
4; +btree.obj       &
5; +datafile.obj    &
6; +dblist.obj      &
7; +clist.obj       &
8; +filename.obj    &
9; +sort            &
10; +sys.obj

```

Vendor:

Microsoft Corporation
 16011 NE 36th Way
 Redmond WA 98073-9717
 (800)541-1261

TopSpeed C 1.04

The TopSpeed compiler does not use the traditional makefile. Instead, it uses project files to build individual programs and libraries. Listing 9.3 is maketsc.bat, a DOS batch file that runs the TopSpeed compiler for each of the project files. To build the software, type the name of the batch file on the command line.

Listing 9.3 maketsc.bat

```
1| echo off
2| rem ----- mk.bat
3| rem  TopSpeed C batch file to build Cdata into an application
4|
5| rem ----- build the library
6| tsc CDATA /M
7|
8| rem ----- build the schema
9| tsc SCHEMA /M
10|
11| rem ----- Cdata utility programs
12| tsc DBINIT /M
13| tsc INDEX /M
14| tsc DBSIZE /M
15| tsc DS /M
16| tsc QD /M
17|
18| rem ----- Consultant's Billing System application programs
19| tsc INVOICE /M
20| tsc POSTTIME /M
21| tsc PAYMENTS /M
22| tsc ROSTER /M
```

Listings 9.4 through 9.14 are the project files that build the cdata.lib library, the cdata utility programs, and the Consultant's Billing System application programs.

Listing 9.4 cdata.prj

```
1| -- cdata.prj
2| make LIBRARY
3| option +case -map +pack -vid +fixed
4| model Small
5| include eelist
6| include cdata
7| include screen
8| include btree
9| include datafile
10| include dblist
11| include clist
12| include filename
13| include sort
14| include sys
```

Listing 9.5 schema.prj

```
1| -- schema.prj
2| make DOS EXE
3| option +case +map +pack -vid
4| model Small
5| include initexe %n
6| import %0%_clib
7| run schema cbs
```

Listing 9.6 dbinit.prj

```
1| -- dbinit.prj
2| make DOS EXE
3| option +case +map +pack -vid
4| model Small
5| include initexe %n cbs
6| import cdata
7| import %0%_clib
```

9 Building the Software

Listing 9.7 index.prj

```
1| -- index.prj
2| make DOS EXE
3| option +case +map +pack -vid
4| model Small
5| include initexe %n cbs
6| import cdata
7| import %0%_clib
```

Listing 9.8 dbsize.prj

```
1| -- dbsize.prj
2| make DOS EXE
3| option +case +map +pack -vid
4| model Small
5| include initexe %n cbs
6| import cdata
7| import %0%_clib
```

Listing 9.9 qd.prj

```
1| -- qd.prj
2| make DOS EXE
3| option +case +map +pack -vid
4| model Small
5| include initexe %n cbs
6| import cdata
7| import %0%_clib
```

Listing 9.10 ds.prj

```
1| -- ds.prj
2| make DOS EXE
3| option +case +map +pack -vid
4| model Small
5| include initexe %n cbs
6| import cdata
7| import %0%M_clib
```

Listing 9.11 invoice.prj

```
1| -- invoice.prj
2| make DOS EXE
3| option +case +map +pack -vid
4| model Small
5| include initexe %n cbs
6| import cdata
7| import %0%M_clib
```

Listing 9.12 roster.prj

```
1| -- roster.prj
2| make DOS EXE
3| option +case +map +pack -vid
4| model Small
5| include initexe %n cbs
6| import cdata
7| import %0%M_clib
```

Listing 9.13 posttime.prj

```
1| -- posttime.prj
2| make DOS EXE
3| option +case +map +pack -vid
4| model Small
5| include initexe %n cbs
6| import cdata
7| import %0%M_clib
```

Listing 9.14 payments.prj

```
1| -- payments.prj
2| make DOS EXE
3| option +case +map +pack -vid
4| model Small
5| include initexe %n cbs
6| import cdata
7| import %0%M_clib
```

Vendor:

Jensen & Partners International, Inc.
1101 San Antonio Road, Suite 301
Mountain View, CA 94043
(415) 967-3200

Turbo C 2.0 and Turbo C++ 1.0

The programs in this book compile with Turbo C 2.0 and Turbo C++ 1.0. The procedures are the same. Listing 9.15 is makefile.tc. Change the TPATH macro in makefile.tc if you have installed the Turbo compiler somewhere other than on the C drive in the \tc subdirectory. The Turbo compilers use the cdata.bld librarian command file from Listing 9.2 above.

To build the software, enter this command:

```
make -fmakefile.tc
```

Listing 9.15 makefile.tc

```
1| # -----
2| # Turbo C MAKEFILE to build Cdata into an application
3| #
4|
5| #
6| # Where Turbo C is installed
7| #
8| TCPATH = c:\tc
9| #
10| # Application Name
11| #
12| CDATA_APPL = cbs
13| #
14| # Memory Model
15| #
16| MODEL = s
17|
18| STARTUP = $(TCPATH)\lib\c0$(MODEL)
19| LIBS = cdata $(TCPATH)\lib\c$(MODEL)
20|
21| #
22| # An implicit rule to build .OBJ files from .C files
23| #
24|
25| .c.obj:
26|     tcc -c $*
27|
28| #
29| # An implicit rule to build .EXE files from .OBJ files
30| #
31|
32| .obj.exe:
33|     tlink $(STARTUP) $* $(CDATA_APPL).obj, $*,,$(LIBS)
34|
35| all : cdata.lib      \
36|       schema.exe    \
37|       qd.exe        \
38|       ds.exe        \
```

continued...

...from previous page (Listing 9.15)

```
39|         index.exe      \
40|         dbsize.exe    \
41|         dbinit.exe    \
42|         postime.exe   \
43|         payments.exe  \
44|         invoice.exe   \
45|         roster.exe
46|
47| # -----
48| # CDATA.LIB
49| # -----
50|
51| cdata.lib      : ellist.obj      \
52|                   cdata.obj      \
53|                   screen.obj     \
54|                   btree.obj      \
55|                   datafile.obj   \
56|                   dblist.obj     \
57|                   clist.obj      \
58|                   filename.obj  \
59|                   sort.obj       \
60|                   sys.obj
61|     del cdata.lib
62|     tlib cdata @cdata.bld
63|
64| ellist.obj     : ellist.c cdata.h
65|
66| cdata.obj      : cdata.c cdata.h datafile.h btree.h keys.h
67|
68| screen.obj     : screen.c cdata.h screen.h keys.h
69|
70| btree.obj      : btree.c cdata.h btree.h
71|
72| datafile.obj   : datafile.c cdata.h datafile.h
73|
74| dblist.obj     : dblist.c cdata.h
75|
76| clist.obj      : clist.c cdata.h screen.h
```

continued...

...from previous page (Listing 9.15)

```
77
78 sort.obj      : sort.c cdata.h sort.h
79
80 sys.obj       : sys.c cdata.h sys.h
81
82 filename.obj : filename.c cdata.h
83
84 # -----
85 # The Data Base Schema
86 # -----
87
88 schema.exe    : schema.obj
89         tlink $(STARTUP) $*, $*,,$(LIBS)
90
91 schema.obj   : schema.c
92
93 $(CDATA_APPL).c : $(CDATA_APPL).sch schema.exe
94         schema.$(CDATA_APPL)
95
96 $(CDATA_APPL).obj : $(CDATA_APPL).c
97
98 # -----
99 # Cdata Utility Programs
100 #
101
102 qd.exe        : qd.obj $(CDATA_APPL).obj cdata.lib
103
104 ds.exe        : ds.obj $(CDATA_APPL).obj cdata.lib
105
106 index.exe    : index.obj $(CDATA_APPL).obj cdata.lib
107
108 dbsize.exe   : dbsize.obj $(CDATA_APPL).obj cdata.lib
109
110 dbinit.exe   : dbinit.obj $(CDATA_APPL).obj cdata.lib
111
112 qd.obj       : qd.c cdata.h screen.h keys.h
113
114 ds.obj       : ds.c cdata.h
```

continued...

...from previous page (Listing 9.15)

```
115: index.obj      : index.c cdata.h
116: dbsize.obj    : dbsize.c cdata.h btree.h datafile.h
117:
118: dbinit.obj    : dbinit.c cdata.h datafile.h
119:
120: # -----
121: # Application-specific (CBS) programs
122: # -----
123:
124:
125:
126: posstime.exe   : posstime.obj cbs.obj cdata.lib
127:
128: payments.exe   : payments.obj cbs.obj cdata.lib
129:
130: invoice.exe    : invoice.obj cbs.obj cdata.lib
131:
132: roster.exe     : roster.obj cbs.obj cdata.lib
133:
134: posstime.obj   : posstime.c cbs.h screen.h keys.h
135:
136: payments.obj   : payments.c cbs.h screen.h keys.h
137:
138: invoice.obj    : invoice.c cbs.h
139:
140: roster.obj     : roster.c cbs.h sort.h
```

Vendor:

Borland International
1800 Green Hills Road
Scotts Valley, CA 95066-0001
(800)331-0877

WATCOM C 8.0

Listing 9.16 is makefile.wat, the file that the WATCOM Wmake utility uses to build the software in this book. Listing 9.17 is watcdata.bld, the WATCOM librarian's command file. To build the software, enter this command:

```
wmake /f makefile.wat
```

Listing 9.16 makefile.wat

```
1: # -----
2: # Watcom C MAKEFILE to build Cdata into an application
3: #
4:
5: #
6: # Application Name
7: #
8: CDATA_APPL    = cbs
9:
10:#
11:# An implicit rule to build .OBJ files from .C files
12:#
13:
14:.c.obj:
15:    wcl /c /w3 $*
16:
17:#
18:# An implicit rule to build .EXE files from .C files
19:#
20:
21:.c.exe:
22:    wcl /w3 $* $(CDATA_APPL).obj cdata.lib
23:
24:all : cdata.lib      &
25:      schema.exe   &
26:      qd.exe        &
27:      ds.exe        &
28:      index.exe     &
29:      dbsize.exe    &
```

continued...

9 Building the Software

...from previous page (Listing 9.16)

```
30:      dbinit.exe    &
31:      posttime.exe  &
32:      payments.exe  &
33:      invoice.exe   &
34:      roster.exe    &
35:      echo All done
36:
37: # -----
38: # CDATA.LIB
39: # -----
40:
41: cdata.lib     : ellist.obj    &
42:                  cdata.obj    &
43:                  screen.obj   &
44:                  btree.obj    &
45:                  datafile.obj &
46:                  dblist.obj   &
47:                  clist.obj    &
48:                  filename.obj &
49:                  sort.obj    &
50:                  sys.obj
51:      del cdata.lib
52:      wlib cdata @watcdata.bld
53:
54: ellist.obj    : ellist.c cdata.h
55:
56: cdata.obj     : cdata.c cdata.h datafile.h btree.h keys.h
57:
58: screen.obj    : screen.c cdata.h screen.h keys.h
59:
60: btree.obj     : btree.c cdata.h btree.h
61:
62: datafile.obj  : datafile.c cdata.h datafile.h
63:
64: dblist.obj    : dblist.c cdata.h
65:
66: clist.obj     : clist.c cdata.h screen.h
67:
```

continued...

...from previous page (Listing 9.16)

```
68 sort.obj      : sort.c cdata.h sort.h
69
70 sys.obj       : sys.c cdata.h sys.h
71
72 filename.obj : filename.c cdata.h
73
74 # -----
75 # The Data Base Schema
76 #
77
78 schema.exe     : schema.c
79   wcl /w3 schema
80
81 $(CDATA_APPL).c    : $(CDATA_APPL).sch schema.exe
82   schema $(CDATA_APPL)
83
84 $(CDATA_APPL).obj : $(CDATA_APPL).c
85
86 #
87 # Cdata Utility Programs
88 #
89
90 qd.exe        : qd.c cdata.h screen.h keys.h &
91   $(CDATA_APPL).obj cdata.lib
92
93 ds.exe        : ds.c cdata.h $(CDATA_APPL).obj cdata.lib
94
95 index.exe     : index.c cdata.h $(CDATA_APPL).obj cdata.lib
96
97 dbsize.exe    : dbsize.c cdata.h btree.h datafile.h &
98   $(CDATA_APPL).obj cdata.lib
99
100 dbinit.exe   : dbinit.c cdata.h datafile.h &
```

continued...

..from previous page (Listing 9.16)

```
101;           $(CDATA_APPL).obj cdata.lib
102;
103; # -----
104; # Application-specific (CBS) programs
105; #
106;
107; posttime.exe : posttime.c cbs.h screen.h keys.h &
108;                 cbs.obj cdata.lib
109;
110; payments.exe : payments.c cbs.h screen.h keys.h &
111;                 cbs.obj cdata.lib
112;
113; invoice.exe  : invoice.c cbs.h cbs.obj cdata.lib
114;
115; roster.exe   : roster.c cbs.h sort.h cbs.obj cdata.lib
```

Listing 9.17 watcdata.bld

```
1; +cdata.obj
2; +ellist.obj
3; +screen.obj
4; +btree.obj
5; +datafile.obj
6; +dblist.obj
7; +clist.obj
8; +filename.obj
9; +sort
10; +sys.obj
```

Vendor:

WATCOM Products Inc.
415 Phillip Street
Waterloo, Ontario
CANADA N2L 3X2
(800)265-4555

COMPILER COMPARISON

Table 9-1 is for those who are interested in benchmarks. It compares the performance of the compilers with respect to the time it takes to build the software in this book and the sizes of the executable programs. I ran the tests on a 33 Mhz 386 with cache. I made no attempt to optimize the compiler environments by using RAM disks or the like. I installed the compilers directly out of the box, used the default compiler options, and the makefiles shown in this chapter.

Table 9-1. Compiler Comparision

Compiler:	Microsoft	TopSpeed	Turbo C	Turbo C++	WATCOM C
Build Time:	2:49	3:10	0:53	1:42	3:00
File Sizes					
schema.exe	14857	14855	14688	14270	12634
qd.exe	29633	26524	29598	29214	24190
ds.exe	29717	15315	29668	29300	24116
index.exe	25043	17400	23354	22744	20290
dbsize.exd	27323	11899	25552	24708	22946
dbinit.exe	24265	9963	22624	21738	19626
posttime.exe	28285	22978	28362	27670	22986
payments.exe	28001	22663	28064	27372	22720
invoice.exe	24449	12935	23274	22388	20064
roster.exe	31611	16418	31510	30834	25334

HOW MAY YOU USE THIS SOFTWARE?

I get frequent inquiries about the first edition of this book asking the extent to which a reader may use the code. The readers are concerned that because the code appears in a copyrighted book, their rights to use it might somehow limit them. Here are the particulars.

You may develop any program you want by using the source code in this book. You may sell or give away the executable programs or their object modules, including the object modules that you compile from the code in this book. You may modify this code to your heart's content for your own use.

You may not publish this source code as your own work even if you have modified it. You may not publish it as my work unless you get my permission.

If you are developing a program for a client who requires that you deliver source code, buy them a copy of the book along with the diskette. If you are selling a program to the general public and you want to include source code, then you must notify me to work out an arrangement.

HOW CAN YOU GET HELP?

I will always respond to questions about and problems with this code on the CompuServe on-line service. My identification code is 71101.1262. That is a convenient way for me to work with you because I can do it when my schedule permits. I cannot promise to answer letters or be available for telephone support as I have in the past. The number of calls and letters I get has become unmanageable.

If you are having problems, call MIS Press, the publisher of this book first to see if there are updates to the diskette. Then get in touch with me. If you have typed the code in, do not expect much help. Virtually every time I fielded a problem from someone who typed the code in from the first edition, the problem turned out to be a typographical error. If you are serious about using the code in this book for a software project, begin by getting the diskette.

Appendix

Cdata Reference Guide

This appendix is a reference guide to the Cdata programs, functions, and data structures. It provides a brief summary of each item and refers you to the page where the book discusses the item and the listing where the item occurs.

CDATA FUNCTIONS

These are the high-level database management functions.

```
void db_open(
    const char *path, /* DOS path to database */
    const DBFILE *fl  /* list of files to open */
)
```

Open a set of files in the database. Page 119, Listing 6.15, cdata.c.

```
int add_rcd(
    DBFILE f, /* file number */ 
    void *bf /* buffer containing record */ 
)
```

Add a record to the database. Page 119. Listing 6.15, cdata.c.

```
int find_rcd(
    DBFILE f, /* file number */ 
    int k, /* key number */ 
    char *key, /* key value */ 
    void *bf /* buffer for record */ 
)
```

Find a record with a key. Page 120. Listing 6.15, cdata.c.

```
int verify_rcd(
    DBFILE f, /* file number */ 
    int k, /* key number */ 
    char *key, /* key value */ 
)
```

Verify that a record exists with the specified key. Page 120. Listing 6.15, cdata.c.

```
int first_rcd(
    DBFILE f, /* file number */ 
    int k, /* key number */ 
    void *bf /* buffer for record */ 
)
```

Retrieve the first record in the key sequence. Page 121. Listing 6.15, cdata.c.

```
int last_rcd(
    DBFILE f, /* file number */ 
    int k, /* key number */ 
    void *bf /* buffer for record */ 
)
```

Retrieve the last record in the key sequence. Page 121. Listing 6.15, cdata.c.

```
int next_rcd(
    DBFILE f, /* file number */ 
    int k, /* key number */ 
    void *bf /* buffer for record */ 
)
```

Retrieve the next record in the key sequence. Page 122. Listing 6.15, cdata.c.

```
int prev_rcd(
    DBFILE f,           /* file number          */
    int k,              /* key number          */
    void *bf            /* buffer for record  */
)
```

Retrieve the previous record in the key sequence. Page 122. Listing 6.15, cdata.c.

```
int rtn_rcd(
    DBFILE f,           /* file number          */
    void *bf            /* buffer for record  */
)
```

Return an updated record to the database. Page 123. Listing 6.15, cdata.c.

```
int del_rcd(
    DBFILE f           /* file number          */
)
```

Delete a record previously retrieved. Page 123. Listing 6.15, cdata.c.

```
int curr_rcd(
    DBFILE f,           /* file number          */
    int k,              /* key number          */
    void *bf            /* buffer for record  */
)
```

Retrieve the current record in the key sequence. Page 124. Listing 6.15, cdata.c.

```
int seqrcd(
    DBFILE f,           /* file number          */
    void *bf            /* buffer for record  */
)
```

Retrieve the next physically sequential record in the file. Page 124. Listing 6.15, cdata.c.

db_cls()

Close the database. Page 125. Listing 6.15, cdata.c.

dberror()

Post a database error. Page 125. Listing 6.15, cdata.c.

```
void rcd_fill(
    const void *s,          /* source record buffer      */
    void *d,                /* destination record buffer */
    const ELEMENT *slist,   /* source data element list */
    const ELEMENT *dlist   /* destination data element list */
)
```

Logically move the data elements in one buffer to the corresponding data element positions in another buffer. Page 125. Listing 6.15, cdata.c.

```
int epos(
    ELEMENT el,            /* data element              */
    const ELEMENT *list    /* element list              */
)
```

Return an offset to a data element position in a buffer. Page 126. Listing 6.15, cdata.c.

```
int rlen(
    DBFILE f               /* file number              */
)
```

Return the length of a database file record. Page 127. Listing 6.15, cdata.c.

```
init_rcd(
    DBFILE f,              /* file number              */
    void *bf                /* buffer for record        */
)
```

Set a database record buffer to blank values. Page 127. Listing 6.15, cdata.c.

```
void clrrcd(
    void *bf,              /* buffer                  */
    const ELEMENT *els     /* data element list        */
)
```

Set a buffer of data elements to blank values. Page 128. Listing 6.15, cdata.c.

DATAFILE FUNCTIONS

These are the lower-level file-management functions that the Cdata database manager uses to create, read, and update data files.

```
void file_create(
    char *name,          /* file name           */
    int len               /* record length       */
)
```

Create a data file. Page 145. Listing 6.17, datafile.c.

```
int file_open(
    char *name           /* file name           */
)
```

Open a data file. Page 145. Listing 6.17, datafile.c.

```
void file_close(
    int fp                /* file handle         */
)
```

Close a data file. Page 145. Listing 6.17, datafile.c.

```
RPTR new_record(
    int fp,              /* file handle         */
    void *bf)            /* record buffer       */
)
```

Add a new record to a data file. Page 146. Listing 6.17, datafile.c.

```
int get_record(
    int fp,              /* file handle         */
    RPTR rcdno,           /* logical record number */
    void *bf)            /* record buffer       */
)
```

Get a record from a data file. Page 146. Listing 6.17, datafile.c.

```
int put_record(
    int fp,              /* file handle         */
    RPTR rcdno,           /* logical record number */
    void *bf)            /* record buffer       */
)
```

Write a record to a data file. Page 146. Listing 6.17, datafile.c.

```
int delete_record(
    int fp,           /* file handle      */
    RPTR rcdno        /* logical record number */
```

)
Delete a record from a data file. Page 147. Listing 6.17, datafile.c.

B-TREE FUNCTIONS

These are the lower-level B-tree functions that the database manager calls to maintain the data file indexes.

```
void build_b(
    char *name,        /* B-tree file name   */
    int len             /* key length         */
```

)
Build an index file. Page 173. Listing 6.19, btree.c.

```
int btree_init(
    char *name        /* B-tree file name   */
```

)
Initialize processing an index file. Page 173. Listing 6.19, btree.c.

```
int btree_close(
    int tree          /* tree number        */
```

)
Close an index file. Page 173. Listing 6.19, btree.c.

```
int insertkey(
    int tree,          /* tree number        */
    char *key,          /* key value          */
    RPTR ad,            /* key's leaf          */
    int unique          /* TRUE = unique key */
```

)
Insert a key into an index. Page 174. Listing 6.19, btree.c.

```
RPTR locate(
    int tree,           /* tree number          */
    char *key,          /* key value            */
)
```

Locate the R PTR leaf value associated with a key. Page 174. Listing 6.19, btree.c.

```
int deletekey(
    int tree,           /* tree number          */
    char *key,          /* key value            */
    R PTR ad            /* key's leaf           */
)
```

Delete a key from an index. Page 175. Listing 6.19, btree.c.

```
R PTR firstkey(
    int tree           /* tree number          */
)
```

Retrieve the leaf value associated with the first index entry in the sequence of the index. Page 175. Listing 6.19, btree.c.

```
R PTR lastkey(
    int tree           /* tree number          */
)
```

Retrieve the leaf value associated with the last index entry in the sequence of the index. Page 175. Listing 6.19, btree.c.

```
R PTR nextkey(
    int tree           /* tree number          */
)
```

Retrieve the leaf value associated with the next index entry in the sequence of the index. Page 176. Listing 6.19, btree.c.

```
R PTR prevkey(
    int tree           /* tree number          */
)
```

Retrieve the leaf value associated with the previous index entry in the sequence of the index. Page 176. Listing 6.19, btree.c.

```
void keyval(
    int tree,           /* tree number      */
    char *key           /* key value       */
)
```

Retrieve the key value of the current entry in an index. Page 176. Listing 6.19, btree.c.

```
RPTR currkey(
    int tree           /* tree number      */
)
```

Retrieve the leaf value associated with the current index entry in the sequence of the index. Page 177. Listing 6.19, btree.c.

CONSOLE FUNCTIONS

```
int get_char()
```

Read a character from the keyboard. Performs function key translation. Page 184. Listing 7.2, sys.c.

```
void put_char(
    int c             /* the character to write */
)
```

Write a character to the screen. Translates space forward and backspace characters into ANSI.SYS values. Page 184. Listing 7.2, sys.c.

```
void clear_screen()
```

Clear the screen. Page 185. Listing 7.2, sys.c.

```
void cursor(
    int x,            /* cursor column coordinate */
    int y             /* cursor row coordinate */
)
```

Position the screen cursor. Page 185 Listing 7.2, sys.c

SCREEN MANAGER FUNCTIONS

```
init_screen(
    char *name,          /* template name      */
    const ELEMENT *elist /* data element list  */
    char *bf             /* data entry buffer   */
)
```

Initialize the screen to perform template data entry. Page 221. Listing 7.11, screen.c.

```
protect(
    ELEMENT el,          /* field data element */
    int tf               /* true to protect    */
)
```

Protect and unprotect a data entry template field from data entry. Page 221. Listing 7.11, screen.c.

```
edit(
    ELEMENT el,          /* field data element */
    int (*func)()         /* edit function to call */
)
```

Provide a custom validation function for a field on a template. Page 222. Listing 7.11, screen.c.

```
display_template()
```

Display a template on the screen. Page 222. Listing 7.11, screen.c.

```
int data_entry()
```

Perform data entry into a template. Page 222. Listing 7.11, screen.c.

```
tally()
```

Display the data values for all the fields on a screen template. Page 222. Listing 7.11, screen.c.

```
put_field(
    ELEMENT el           /* field data element */
)
```

Display the data value for a selected field on a screen template. Page 223. Listing 7.11, screen.c.

DATA RECORD DISPLAY FUNCTIONS

```
int ellist(
    int count,           /* number of names in the list */
    char *names[],      /* the names */
    int *list            /* the resulting list */
)
```

Build an array of ELEMENT values from an array of data element name strings.
Page 205. Listing 7.9, ellist.c.

```
void dblist(
    FILE *fd,           /* output file */
    DBFILE f,            /* data base file number */
    int k,               /* key number */
    const ELEMENT *list /* element list */
)
```

Display selected data elements from the records in a file in file order or key sequence.
Page 223. Listing 7.12, dblist.c.

```
void clist(
    FILE *fd,           /* output file */
    const ELEMENT *fl,  /* list of elements in buffer */
    const ELEMENT *pl,  /* list of elements to print */
    char *bf,            /* input buffer address */
    const char *fn       /* file name for report header */
)
```

Display selected data elements from a buffer of data elements.
Page 224. Listing 7.13, clist.c.

UTILITY FUNCTIONS

```
DBFILE filename(
    char *fn             /* file name to convert */
)
```

Convert a file name into its DBFILE value.
Page 229. Listing 7.14, filename.c.

```

int name_cvt(
    char *c2,           /* destination for converted string */
    char *c1           /* string to convert */
)

```

Convert a string to upper case. Page 230. Listing 7.14, filename.c.

SORT FUNCTIONS

```

int init_sort(
    ELEMENT *list,      /* element list of record to be sorted */
    ELEMENT *seq,       /* element list of sort sequence */
    int direction       /* TRUE = ascending sort */
)

```

Initialize the sort process by describing the buffer to sort and the data elements to sort. Page 232. Listing 7.16, sort.c.

```

void sort(
    void *rcd
)

```

Pass a record to the sort process to be sorted. Page 232. Listing 7.16, sort.c.

```
char *sort_op()
```

Retrieve sorted records from the sort process. Page 232. Listing 7.16, sort.c.

DATA STRUCTURES

Data File Header

```

typedef struct fhdr { /* header on each file
    RPTR first_record; /* first available deleted record */
    RPTR next_record;  /* next available record position */
    int record_length; /* length of record */
}FHEADER;

```

A data file header structure appears at the beginning of every data file. Page 113. Listing 6.16, datafile.h.

B-Tree Header

```
typedef struct treehdr {
    RPTR rootnode;      /* root node number */
    int keylength;     /* length of a key */
    int m;             /* max keys/node */
    RPTR rlsed_node;   /* next released node */
    RPTR endnode;      /* next unassigned node */
    int locked;        /* if btree is locked */
    RPTR leftmost;    /* left-most node */
    RPTR rightmost;   /* right-most node */
} HEADER;
```

A B-tree header structure appears at the beginning of every B-tree file. Page 116. Listing 6.18, btree.h.

B-Tree Node

```
typedef struct treenode {
    int nonleaf;        /* 0 if leaf, 1 if non-leaf */
    RPTR prtnode;      /* parent node */
    RPTR lfsib;         /* left sibling node */
    RPTR rtsib;         /* right sibling node */
    int keyct;          /* number of keys */
    RPTR key0;          /* node # of keys < 1st key this node */
    char keyspace [NODE - ((sizeof(int) * 2) + (ADR * 4))];
    char spil [MXKEYLEN]; /* for insertion excess */
} BTREE;
```

This is the format of a B-tree node record. Page 116. Listing 6.18, btree.h.

Screen Template Elements

```
static struct {
    int prot;           /* element protected = TRUE */
    int (*edits)();    /* custom edit function */
} sb [];
```

This structure specifies whether a data entry field is protected and contains a pointer to its custom validation function. Page 208. Listing 7.11, screen.c.

GLOBAL SYMBOLS

Symbol	Description	Page	Listing	File
BTREE	Typedef for a B-tree node	150	6.18	btree.h
DBFILE	Typedef for database files	72	6.10	cdata.h
ELEMENT	Typedef for data elements	70	6.10	cdata.h
FHEADER	Typedef for a data file header	144	6.16	datafile.h
HEADER	Typedef for a B-tree header	150	6.18	btree.h
LEASTMEM	Least memory for sort buffer	231	7.15	sort.h
MAXSORTS	Maximum fields in a sort sequence	233	7.15	sort.h
MOSTMEM	Most memory for sort buffer	231	7.15	sort.h
MXCAT	Maximum elements per index	85	6.10	cdata.h
MXELE	Maximum data elements in a file	85	6.10	cdata.h
MXFILS	Maximum files in a data base	85	6.10	cdata.h
MXINDEX	Maximum indexes per file	85	6.10	cdata.h
MXKEYLEN	Maximum key length for indexes	85	6.10	cdata.h
MXTREES	Maximum concurrently open B-trees	147	6.18	btree.h
NAMLEN	Data element name length	85	6.10	cdata.h
NODE	Length of a B-tree node	147	6.18	btree.h
RPTR	Data type of record and node pointer	85	6.10	cdata.h

CDATA UTILITY PROGRAMS

Program	Description	Page	Listing	File
DBINIT.EXE	Data base initializer	191	7.5	dbinit.c
DBSIZE.EXE	Data base size calculator	188	7.4	dbsize.c
DS.EXE	Data base file lister	203	7.7	ds.c
INDEX.EXE	Rebuild indexes	206	7.8	index.c
QD.EXE	Data base file query	193	7.6	qd.c
SCHEMA.EXEC	data schema compiler	89	6.11	schema.c

CONSULTANT'S BILLING SYSTEM EXAMPLE PROGRAMS

Program	Description	Page	Listing	File
INVOICE.EXE	Prepares client invoices	113	6.14	invoice.c
PAYMENTS.EXE	Posts payments to clients file	260	8.6	payments.c
POSTTIME.EXE	Posts time and expenses to projects	257	8.5	posttime.c
ROSTER.EXE	List clients by client name	246	7.17	roster.c

CDATA SCHEMA DATA DEFINITION LANGUAGE

```
; ----- Comment
#schema <database name>
#dictionary
    <data element name>,<element type>,<element length>[,<element mask>]
    ; <element type>:
    ;   Z = zero-filled integer
    ;   N = space filled integer
    ;   A = string
    ;   C = currency
    ;   D = date (mm/dd/yy)
    ; <element mask>
    ;   "(____)___-__"
    ;   "$_____.__"
#end dictionary
; ---- file specifications
#file <file name>
    <data element name>
    <data element name>
#end file
; ---- index specifications
#key <file name> <data element name>[, <data element name>]
#end schema <database name>
```

Index

#	cbs.h, 101	lengths, 64
#dictionary, 80	cbs.sch, 77	names, 13, 65
#end dictionary, 80	Cdata, 61	types, 12, 66, 80
#end file, 80, 81	cdata.bld, 265	data entry, 247
#end schema, 80	cdata.c, 129	data items, 31
#file, 80, 81, 113, 116	cdata.h, 82	data manipulation
#key, 80, 81, 116	cdata.prj, 267	language, 52, 118
#schema, 80	clear_screen, 185	data models, 48
A	clist, 224	data posting, 251
add_rcd, 119	clist.c, 225	data value, 11
ANSI, 1	clrrcd, 128	database, 8
ANSI.SYS, 185, 260	comments, DDL, 79	database management
applications programs, 55	compiler comparison, 279	system, 43
argc, 108	concatenated keys, 21, 74	datafile.c, 142
argv, 108	CONFIG.SYS, 260	datafile.h, 141
B	connector file, 21, 51	data_entry, 222
B-trees, 114	Consultant's Billing	dberror, 125
benchmarks, 279	System, 61, 245	dberrors, 85
BIOS, 185	conversion programs, 54	DBFILE, 72, 85, 107
Boolean query, 37	CP/M, 4	dbinit.c, 189
btree.c, 149	curkey, 177	dbinit.prj, 267
btree.h, 147	curr_rcd, 124	dblist.c, 224
btree_close, 173	cursor, 185	DBMS, 43, 55
btree_init, 173	D	dbsize.c, 186
build_b, 173	data definition language,	dbsize.prj, 268
C	15, 47, 55, 69, 79	db_cls, 112, 125
CBS, 61, 245	data element, 10	db_list, 223
cbs.c, 103	data element dictionary, 11,	db_open, 112, 119
	33, 63, 80	DDL, 15, 47, 55, 69, 79
	data value, 11	DDL directives:
	keys, 15	#dictionary, 80
		#end dictionary, 80
		#end file, 80, 81

- #end schema, 80
- #file, 80, 81, 113, 116
- #key, 80, 81, 116
- #schema, 80
- deletekey, 175
- delete_record, 147
- del_rcd, 123
- directives, DDL, 80
- display masks, 67, 81
- display_template, 222
- DML, 52, 118
- DOS, 4
- ds.c, 201
- ds.prj, 269
- D_BOF, 122, 123
- D_DUPL, 120, 123
- D_EOF, 121, 122, 124
- D_NF, 120, 121
- D_PRIOR, 123, 124

- E**
- edit, 222
- ELEMENT, 70, 85, 107
- ellist, 205
- ellist.c, 206
- enumerated data type, 57
- environment variables, 261
- epos, 126
- errno, 120, 125
- error codes:
 - D_BOF, 122, 123
 - D_DUPL, 120, 123
 - D_EOF, 121, 122, 124
 - D_NF, 120, 121
 - D_PRIOR, 123, 124
- error messages in schema.c, 108

- F**
- fhdr, 113
- FieldChar, 185

- file, 13
- file handles, 260
- file integrity, 39, 51
- file names, 70
- file relationships, 17, 39
- file scan query, 37
- file structure, specifying, 14
- filename, 229
- filename.c, 229
- files, specifying, 15, 68
- FILES=, 260
- file_close, 145
- file_create, 145
- file_ele, 72, 126
- file_open, 145
- find_rcd, 120
- firstkey, 175
- first_rcd, 121
- functional, 30
- functional requirements, 30
- functions:
 - add_rcd, 119
 - btree_close, 173
 - btree_init, 173
 - build_b, 173
 - clear_screen, 185
 - clist, 224
 - clrrcd, 128
 - currkey, 177
 - curr_rcd, 124
 - cursor, 185
 - data_entry, 222
 - dberror, 125
 - db_cls, 112, 125
 - db_list, 223
 - db_open, 112, 119
 - deletekey, 175
 - delete_record, 147
 - del_rcd, 123
 - display_template, 222
 - edit, 222
- ellist, 205
- epos, 126
- filename, 229
- file_close, 145
- file_create, 145
- file_open, 145
- find_rcd, 120
- firstkey, 175
- first_rcd, 121
- get_char, 184
- get_record, 146
- init_rcd, 127
- init_screen, 221
- init_sort, 232
- insertkey, 174
- keyval, 176
- lastkey, 175
- last_rcd, 121
- locate, 174
- name_cvt, 230
- new_record, 146
- nextkey, 176
- next_rcd, 112, 122
- prevkey, 176
- prev_rcd, 122
- protect, 221
- put_char, 184
- put_field, 223
- put_record, 146
- rcd_fill, 125
- rlen, 127
- rtn_rcd, 123
- seqrcd, 124
- sort, 232
- sort_op, 232
- tally, 222
- verify_rcd, 120
- FWD, 184

G

getchar, 184
get_char, 184
get_record, 146

H

hierarchical data model, 48

I

INCLUDE environment variable, 261
index builder programs, 54
index.c, 204
index.prj, 268
indexes, 74
index_ele, 74
initialization programs, 54
initializing a database, 188, 246
init_rcd, 127
init_screen, 221
init_sort, 232
input/output redirection, 202
insertkey, 174
integrity between files, 39, 51
invoice.c, 110
invoice.prj, 269

K

key data elements, 15
key range query, 37
key retrievals, 35
keyboard functions for qd, 200
keys, 15, 74
keys.h, 181
keys: concatenated, 21
keys: primary and secondary, 16, 82
keyval, 176

L

lastkey, 175
last_rcd, 121
LEASTMEM, 231
LIB environment variable, 261
limits of a DBMS, 45
listings:
 btree.c, 149
 btree.h, 147
 cbs.c, 103
 cbs.h, 101
 cbs.sch, 77
 edata.bld, 265
 edata.c, 129
 edata.h, 82
 edata.prj, 267
 clist.c, 225
 datafile.c, 142
 datafile.h, 141
 dbinit.c, 189
 dbinit.prj, 267
 dblist.c, 224
 dbsize.c, 186
 dbsize.prj, 268
 ds.c, 201
 ds.prj, 269
 ellist.c, 206
 filename.c, 229
 index.c, 204
 index.prj, 268
 invoice.c, 110
 invoice.prj, 269
 keys.h, 181
 makefile.msc, 262
 makefile.tc, 271
 makefile.wat, 275
 maketc.bat, 266
 payments.c, 256
 payments.prj, 270
 posttime.c, 253

P

posttime.prj, 270
qd.c, 191
qd.prj, 268
roster.c, 244
roster.prj, 269
schema.c, 86
schema.prj, 267
screen.c, 208
screen.h, 207
sort.c, 233
sort.h, 231
sys.c, 183
sys.h, 182
watcdata.bld, 278
locate, 174

M

makefile.msc, 262
makefile.tc, 271
makefile.wat, 275
maketc.bat, 266
malloc, 127
many-to-many relationships, 17, 51
Microsoft C, 5, 262
models of data, 48
MOSTMEM, 231
multiple-file queries, 38, 41
MXCAT, 85
MXELE, 85, 109
MXFILS, 85, 110
MXINDEX, 85, 109
MXKEYLEN, 85
MXTREES, 147

N

names, 13
name_cvt, 230
network data model, 49
new_record, 146
nextkey, 176
next_rcd, 112, 122
NODE, 147
normalization, 52

O

one-to-many relationships, 17
one-to-one relationships, 17

P

payments.c, 256
payments.prj, 270
performance, 30
performance requirements, 30
posttime.c, 253
posttime.prj, 270
prevkey, 176
prev_rcd, 122
primary and secondary, 16, 82
primary key, 16, 82
primary keys, 74
protect, 221
putchar, 184
put_char, 184
put_field, 223
put_record, 146

Q

qd.c, 191
qd.prj, 268
qsort, 243
query expressions, 36

R

rcd_fill, 125
relational data model, 50
relationships between files, 17, 39

report writer programs, 54
reports, 250
requirements basis, 27
requirements: functional, 30
requirements: performance, 30

retrieval characteristics, 35
retrieval paths, 41
rlen, 127
roster.c, 244
roster.prj, 269
RPTR, 85, 113, 146
rtn_rcd, 123

S

Schema, 14, 42, 55
schema.c, 86
schema.prj, 267
screen.c, 208
screen.h, 207
secondary key, 16, 82
seqrcd, 124
significance of identifiers, 58
sort, 232
sort programs, 54
sort.c, 233
sort.h, 231
sort_op, 232

strcmp, 243

strongly typed language, 58
structures:

 fhdr, 113
 treenode, 116
 tree_hdr, 116
sys.c, 183
sys.h, 182

T

tally, 222
TermElement, 107
TermFile, 107
TopSpeed C, 5, 266
treenode, 116
tree_hdr, 116
Turbo C, 5, 270
Turbo C++, 270
TurboDos, 4
types, 12

U

UP, 184
utilities:
 conversion programs, 54
 index builder programs, 54
 initialization programs, 54
 report writer programs, 54
 sort programs, 54
utility programs, 53, 179

V

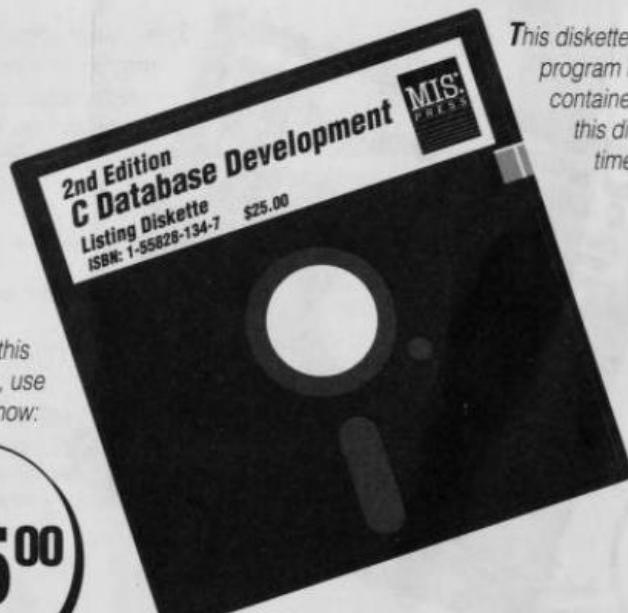
verify_rcd, 120

W

watcdata.bld, 278
WATCOM C, 5, 107, 275

ORDER FORM

PROGRAM LISTINGS ON DISKETTE



If you did not buy this book with diskette, use this form to order now:

Only:
\$25⁰⁰

MIS:PRESS

P.O. Box 5277 • Portland, OR 97208-5277
(503) 282-5215 FAX (503) 222-7064

NAME (Please print or type): _____

Address: _____

CITY: _____ STATE: _____ ZIP: _____

Call free

1-800-626-8257

C Database Development 2nd Edition
ISBN: 1-55828-134-7 Diskette only **\$25.00**
Please add \$3.00 for shipping and handling. (Foreign \$10.00)

Check one:

VISA MasterCard American Express

Check enclosed \$ _____

ACCT. _____

EXP. DATE _____

SIGNATURE _____



MANAGEMENT INFORMATION SOURCE, INC.

CUT TO OPEN

Stevens C DATABASE DEVELOPMENT

2nd Edition ISBN disk: 1-55828-134-7
Copyright © 1991 Management Information Source
Subsidiary of Henry Holt and Company, Inc.



MANAGEMENT INFORMATION SOURCE, INC.

F
D
E

2ND
EDITION

C DATABASE DEVELOPMENT

Learn how to use all the tools needed for writing C Database programs, including a library of database management functions.

Master C Database Development by using helpful programs and utility functions.

Discover how to use the ANSI Standard C language to design and develop programs that manage information in a database.

Obtain the knowledge to apply the standard C language in defining a relational database and implementing the programs needed to process that data.

Acquire the understanding of data structures and how to use them.

TOPICS COVERED INCLUDE:

Database Fundamentals

Designing a Database

The Database Management System

C as a Data Definition Language

Cdata: The cheap Database Management System

Cdata Utility Programs

An Application: The Consultant's Billing System

Building the Software

PUT THE POWER OF C DATABASE DEVELOPMENT AT YOUR FINGERTIPS!

ISBN 1-55828-135-5



9 781558 281356

AL STEVENS' monthly column in Doctor Dobb's Journal has earned him a loyal following among professional programmers who appreciate his insight and clear writing style. He is also a nationally known and respected consultant who assists a variety of business, industry, and government agency clients. The author of C Database Development, Quick C, Turbo C, (MIS:Press, 1987), Extending Turbo C Professional, Teach Yourself DOS Yourself (MIS:Press, 1989), and Teach Yourself C++, Teach Yourself Windows 3 (MIS:Press, 1990), Stevens began programming in 1958 and has been designing systems and programs in C for many years.

Book Only: \$24.95

Book/Disk: \$49.95