DISK AVAILABLE

C++ DATABASE DEVELOPMENT

- FEATURING PARODY—THE PERSISTENT ALMOST-RELATIONAL OBJECT DATABASE MANAGER
- PERSISTENT OBJECTS
- OFFERS FULL C++ SOURCE CODE

MIS: PRESS

AL STEVENS

C++ Database Development

featuring

PARODY

The Persistent Almost-Relational Object Database Management System

A Subsidiary of
Henry Holt and Co., Inc.

Copyright © 1992 by Al Stevens

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 damages resulting from the use of the information contained herein.

ISBN 1-55828-216-5

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

MIS:Press books are available at special discounts for bulk purchases for sales, promotions, premiums, fundraising, or educational use. Special editions or book excerpts can also be created to specification.

For details contact: Special Sales Director
MIS:Press
a subsidiary of Henry Holt and Company, Inc.
115 West 18th Street
New York, New York 10011

TRADEMARKS

Throughout this book, trademarked names are used. Rather than put a trademark symbol after every occurence of a trademarked name, we used the names in an editorial fashion only. Where such designations appear in this book, they have been printed with initial caps.

Development Editor: Cary Weinberger

Copy Editor: Suzanne Ingrao

▼ DEDICATION

To the memory of Admiral Grace Hopper, USN

▼ Table of Contents

Preface XIII

Chapter 1: Introduction 1

C++ and Databases 3
The Organization of C++ Database Development 4
What You Should Already Know and Have 6
The Software Diskette 6
How to Get Help 7
Your Rights to the Software 8

Chapter 2: An Introduction to Object-Oriented Programming 9

The Basics 11
Procedural Programming 11
Object-Oriented Programming 12
- The Object-Oriented Program 12
- The Ojbect 13
- Abstraction 14
- Encapsulation 15
- Methods and Messages 17
- Inheritance 20
- Polymorphism 22

Summary 24
References 25

Chapter 3: A Brief History of Database Management 27

The Database Management System 29
- The Data Definition Language 30
- The Data Manipulation Language 30
- The Query Language 30
- The Report Writer Language 30

Database Models 30
- Proprietary Data Models 31

The Hierarchical Data Model 31
The Network Data Model 32
The Relational Data Model 33
The Object Data Model 36
Summary 39
References 39

Chapter 4: C++ and Persistence 41

File Input/Output in C 42
File Input/Output in C++ 42
The Problem with C++ Persistent Objects 43
The Size of the Object 44
Nonpersistent Members 47
Pointers in the Object 47
References in the Object 48
Format-Defining Related Objects 48
Embedded Objects 48
Objects from Class Libraries 49
Virtual Functions 49
Copies of Objects 50
Summary 51
References 51

Chapter 5: Solving the Persistent Object 53

Alternative Solutions 54
Custom File Input/Output Methods 54
Extend the C++ Language 55
Write a Preprocessor 55
Limit the Scope of a Persistent Class 56
Use a Relational DBMS 56
The Persistent Class Participates in its Own Persistence 56
Objectives of a Persistent Object Database 57
Associative Access 57
Maintenance of Objects 58
Object Integrity 58
Class Relationships 59

Copies of Objects 60
Summary 60
References 61

Chapter 6: The Object Database and the Relational Database .. 63

The Object Database Model 64
The Relational Database Model 66
Choosing a Database Model 68
Adapting the Relational Model to Objects 69
Summary 75
References 75

Chapter 7: Designing an Object Database 77

General Design Guidelines 78
The Ten Steps of Object Database Design 79
Identify the Basis 79
Starting from an Existing Solution 81
Starting from Scratch 81
Write Down the Basis 81
Define the Requirements 82
Functional Requirements 82
Performance Requirements 83
Identify the Data Items 84
Separate the Data Members from the Classes 85
Build the Data Member Dictionary 86
Gather the Data Members Into Classes 87
Identify the Key Memebers 93
Primary Keys 93
Secondary Keys 95
Identify the Class Relationships 95
Identify The Methods 97
Make the Class Persistent 98
Summary 98
References 99

Chapter 8: PARODY 101

Defining a PARODY Database 102
Assigning Class Identifiers 102
Declaring the PARODY Database 103
Defining the Persistent Object Class 104
Defining Keys 108
Secondary Keys 111
Concatenated Keys 112
Relating Classes 113
Multiple-Copy Objects 115
Persistent Objects within Persistent Objects 116
How PARODY Brings it Together 119
Opening the PARODY Database 119
Constructing a Persistent Object 119
Destroying a Persistent Object 122
Object Input/Output: an Overview 125
Integrity Checks 125
PARODY File Architecture 126
Nodes 127
Object File 128
Index File 129
B-Tress 130
Summary 132
References 132

Chapter 9: Using a PARODY Database 133

Building a Database 134
Managing Persistent Objects 134
Retrieving Persistent Objects 135
Creating Persistent Objects 135
Changing Persistent Objects 136
Deleting Persistent Objects 136
The Effect of Adds, Changes, and Deletes 137
Integrity 137
Navigating the Database 138
Navigating by Key: Content-Addressable Objects 138
Navigating by Address: Position-Addressable Objects 141
Converting PARODY Databases 146

Adding Clasees 147
Removing Classes 147
Adding Data Items to an Existing Class 147
Removing Data Items from an Existing Class 148
Adding Keys to an Existing Class 148
Converting the Database 148
Rebuilding the Index File 150
Database Integrity Errors 150
The Index Rebuilding Program 150
Summary 152

Chapter 10: Case Studies 153

Utility Classes 154
String 155
Linked List 155
GUI: The Generic User Interface 158
The Display 158
The Keyboard 159
The GUI Object 159
The Game of Life Application 163
The Concept of Life 163
An Evolution of Generations 164
The Life Program 164
Rules for Survival 165
Running the Program 165
Life as a Persistent Object 166
The Personnel Application 166
The Department Class 167
The Employee Class 168
The Project Class 168
The Assignment Class 168
The Personnel Program 169
Summary 171

Appendix A: Building the Software 173

General Instructions 174
Install Your Compiler 174
Install the PARODY Source Code 174

Borland C++ 175
Microsoft C++ 175
Zortech C++ 176
Testing the Software 176
Files Built by the Make 176
Run the Example Application Programs 177
Rebuild the Indexes 178
The Files 178
Listing A.1 MAKEFILE.BCC 178
Listing A.2 MAKEFILE.MSC 179
Listing A.3 MAKEFILE.ZTC 180
Listing A.4 PARODY.BLD 182

Appendix B: PARODY Reference Guide 183

Global Constants 183
Data Types 184
Bool (parody.h) 184
ObjAddr (parody.h) 184
NodeNbr (parody.h) 184
Class Library Reference 184
GUI (gui.h) 185
String (strings.h) 186
LinkedListHead (link.ist.h) 188
LinkedListEntry (linklist.h) 188
Parody (parody.h) 189
Key (key.h) 189
Persistent (parody.h) 190
Handle (parody.h) 192
Disk Data Structures 192
Generic Disk Files 192
Persistent Objects 193
B-Trees 194

Appendix C: Software Listings 197

C.1 GUI.H 200
C.2 GUI.CPP 201
C.3 STRINGS.H 206

C.4 STRINGS.CPP 208
C.5 LINKLIST.H 210
C.6 LINKLIST.CPP 211
C.7 LIFE.CPP 213
C.8 EMPLOYEE.H 221
C.9 EMPLOYEE.CPP 223
C.10 DEPT.H 227
C.11 DEPT.CPP 229
C.12 PROJECT.H 232
C.13 PROJECT.CPP 235
C.14 ASSIGN.H 239
C.15 ASSIGN.CPP 241
C.16 PERSONEL.H 245
C.16 PERSONEL.CPP 246
C.18 PARODY.H 257
C.19 PARODY.CPP 261
C.20 KEY.H 276
C.21 KEY.CPP 277
C.22 BTREE.H 278
C.23 BTREE.CPP 280
C.24 TNODE.CPP 291
C.25 NODE.H 300
C.26 NODE.CPP 301
C.27 INDEX.CPP 304

Glossary 307

Index 315

▼ Preface

In 1987 I wrote the first edition of *C Database Development*, a book that describes how to build database software applications with a subset of the relational data model implemented in the C language. That book included the C source code for a database management system and implemented a design approach where the programmer uses the features of C to define an application's database. In the approach C serves the traditional functions of the data definition and data manipulation languages, which are usually implemented as separate languages in formal DBMSs.

In 1991, after the ANSI X3J11 committee completed the standard definition of C, I wrote a second edition of *C Database Development*, which used the same methods, but with ANSI-standard C. Both editions of *C Database Development* continue to be widely read, and programmers throughout the world use the software in them to develop many diverse applications. The principle is tried and proven.

This book continues that tradition with the C++ programming language and the newer object-oriented data model—not with a C "wrapper" around the old C function library, but with a completely new C++ class library that uses the object-oriented features of C++ to implement what has come to be known as *persistent objects*.

By using the techniques presented in this book, you can define and build an object database in C++. The programs that you write will use abstraction to define the database; they will use inheritance to become persistent objects; they will use polymorphism to assist in their own persistence. In short, they will be object-oriented database programs. This book contains the C++ source code to achieve all this, a valuable tool for the development of applications. As you read the book and use the software, you will learn about persistent objects, object-oriented databases, and object-oriented programming itself.

Object-oriented programming languages have been in use since the 1970s. C++ itself saw its first light as the object-oriented extension to C over ten years ago within AT&T. It is only recently, however, that C++ compilers for personal computers have been available. The proliferation of these compilers and the warm reception given them by programmers has prepared the way for C++ to replace C as the language of choice for software development in the 1990s.

Database management technology dates back to the 1960s when the first general purpose file management utility packages were developed. Through-

out its history the data processing industry has refined and improved the formal database models and the techniques and procedures that implement them, culminating in the object-oriented data model.

This book brings C++ and the persistent object data model together in a form that gives programmers all the tools with source code to build their own persistent object databases.

Al Stevens
1992

PROGRAM LISTINGS ON DISKETTE

MIS:PRESS

This diskette contains the complete listings for all programs and applications contained in this book—it eliminates the need to type in pages of program code.

If you did not buy this book with a diskette, use this form to order now. *Only*

$25.00

MIS:PRESS
4375 West 1980 South
Salt Lake City, Utah 84104
(810) 972-2221

NAME (PLEASE PRINT OR TYPE)

ADDRESS

CITY STATE ZIP

Call toll-free
1-800-488-5233
1-800-628-9658

C++ Database Development
ISBN: 1-55828-217-3 Only $25.00
Please add $2.00 for shipping and handling
(Foreign $6.00)

Check one:
❑ VISA ❑ Mastercard
❑ American Express
❑ Check enclosed $ _____

ACCOUNT NO.

EXP. DATE

SIGNATURE

Chapter 1

Introduction

"She did not cease to wonder at the persistence of the unforeseen..."
Thomas Hardy

This book and the accompanying software address the subject of C++ persistent objects. The pervasive growth of object-oriented programming technology has made it the chosen technique for software development in the 1990s, and C++ has experienced similar growth. It has become the language of choice for writing object-oriented programs. Persistent objects are one facet of the multi-faceted, object-oriented programming paradigm, but one that is not and has never been supported by the C++ programming language. To have persistent objects in a C++ program, programmers must implement them, either by using a database management system or by writing the persistence code themselves.

This book adds the persistence ingredient to the C++ object-oriented culture by providing support for persistent object database management. This is not to imply that object-oriented programs have had to get by without database management for all this time. To the contrary, traditional database management techniques still work as well as they always have, and you can integrate them effectively into object-oriented programs. However, the traditional methods do not fully exploit the wealth of expression that object-oriented programming brings to a software development project. There are several commercial object-oriented database management systems available now as well. This book is one of the first efforts to bring the technology directly to the programmer in a well-documented class library with source code and discussions on the implementation of the persistent object database management system and its underlying principles.

C++ Database Development is, therefore, about building database software applications by using the features of the C++ programming language. By carefully reading the text, studying the source code, and running the software, you will learn how to design and implement a persistent object database application. You will use a persistent object system called PARODY (Persistent, Almost Relational Object Database), which later chapters fully explain, including complete source code.

Before proceeding into the depths of object-oriented design and database design, we should dispense with one anomaly in our industry's use of the English language. The word *hierarchy* refers to an organization of ranking where each level is subordinate to the one above. As you will learn in Chapter 3, the traditional language of database management defines a *hierarchy* as an organization where a member may have only one superior—or parent—record but may have multiple subordinate—or child—records, reflecting the ancient clerical use of the word. An ordering of rank where a child may have multiple parents is a "network" in database parlance. That distinction necessarily defines boundaries between those two very different database models. The language of object-oriented design makes no such distinction, however, because none has been necessary until now. Class systems with single inheritance and class systems with multiple inheritance are both called hierarchies. When we mix the technologies, though, the varying usages confuse us. This book will try not to add to the confusion but to keep the differences clear.

▼ C++ and Databases

C++ is an object-oriented superset of the C language. By using C++'s object-oriented extensions, a programmer can define new data types and build a class architecture that orders the abstract data types into a well-formed hierarchy.

The concept of the database record format, which dates to the earliest days of data processing, is enhanced by the C++ facility that allows a programmer to build a new data type by defining a class. The class describes the database record. If your database will record essential data about a company's departments, for example, then there will probably be a Department class that describes what the application needs to support department records.

That approach is similar to database definition from the past but with this significant exception: design approaches prior to object-oriented design define only the data representation of a database record and none of its behavior. All processing of the data, which—in the object-oriented view—determines a data record's behavior, is in the domain of the application programs themselves. Different applications programs are free to define different and contradictory behavior for the same entity of data, which is a weakness of that classic approach. Separation of data representation from data behavior produces a potential for data disintegrity and, what is more, does nothing to prevent it.

Object-oriented design defines the data record's functional behavior along with its data representation in one encapsulated unit called a *class*. Application programs that use the class but that are themselves outside the class's definition use but do not specify or otherwise change the class's behavior. You will learn what that concept means, how it works, and its consequences beginning in Chapter 2.

What C++ does not include as an inherent part of the language is the ability to automatically store and retrieve instances of classes (objects) onto and from a persistent storage device. In other words, C++ does not have a built-in database management system for storing and retrieving records in files. That's not suprising, because neither does C. Applications that are written in C and that require such record storage and retrieval either use a general-purpose database management function library or a custom approach to reading and writing disk files. C++ applications could use similar techniques and many of them do. They simply read and write the data values defined in the class definitions of the objects. However, there are extensions to the way C++ classes are built that go beyond the traditional C structure definition, and these

extensions create their own unique problems with respect to data storage and retrieval. Furthermore, the concepts of an object-oriented data store offer the potential to support requirements that are not particularly well supported by traditional database management techniques. So, the evolution of object-oriented technology has spawned a new data model—the object database—with the C++ language as a sound vehicle for building and using the model; that is what this book is about.

▼ The Organization of C++ Database Development

The chapters that follow teach you the concepts of persistent object database design and how to build your own object database by using the PARODY software.

Chapter 2 provides a brief introduction to object-oriented programming. This is not a comprehensive treatment of the subject, which would take a volume of its own. Rather, the chapter defines the terms common to object-oriented programming and relates them to the architecture of an object-oriented computer program. The discussion addresses object-oriented programming as it applies to C++. The C++ object-oriented extensions to C do not, strictly speaking, support pure object-oriented programming as defined in other programming languages. Yet C++ is the leader among object-oriented programming languages, and its approach to the technology is the one most programmers are using and the one you will use with the concepts taught by this book.

Chapter 3 is a brief history of database technology. It discusses the three traditional database models, the *hierarchical*, *network*, and *relational models*; and it introduces the newest model, the object database model. As you will learn, students of object-oriented technology often confuse the hierarchical parent—child relationships of data files and those of base and derived C++ classes, perhaps because they can look alike in design diagrams. You must keep these distinctions clear, however, and Chapter 3 will help you do that.

Chapter 4 is about C++ and persistence. The chapter discusses how C++ programs store data structures on disk and how those traditional techniques do not work with persistent objects in a class inheritance object-oriented design.

Chapter 5 defines the objectives for persistent objects. The chapter offers alternative solutions to the problems posed by Chapter 4; and then it identifies the objectives of the persistent object database management system that later chapters present.

Chapter 6 describes what the object-oriented and relational database models have in common, how they differ, and how many of the traditional relational design methods are still valid for an object-oriented design. The chapter addresses when to use a relational database management system and when to use the object database model.

Chapter 7 describes a step-by-step approach for designing an object database to support the functions of an application. These steps involve more than just designing and writing code. You must learn the functional application, derive the requirements for automating parts of it, and build each class that goes into the design.

Chapter 8 introduces PARODY. You will learn how PARODY works and how to use the features of C++ class design to define the schema for an application's database—the *data description language (DDL)* of traditional database management systems. You will learn to integrate the classes that describe your database with the PARODY base classes to add persistent behavior to your objects.

Chapter 9 describes how a C++ application interacts with the objects in a PARODY database—how to create the database, add new objects, retrieve existing objects, modify them, and delete them from the database. This chapter describes what is sometimes called the *applications program interface (API)* to the database management or the *data management language (DML)*.

Chapter 10 presents examples of programs that use PARODY. The chapter includes some utility classes and three case studies of applications that use the PARODY database management system. The case studies include the complete source code to define the respective databases and to implement the example applications. You will use this chapter for examples in the design and implementation of your own applications.

Appendix A contains the instructions for building the PARODY class library and the case study applications. The appendix includes makefiles and instructions for using several contemporary C++ compiler products that run under MS-DOS.

Appendix B is the PARODY reference guide with descriptions of each of the PARODY classes and their application program interfaces. The reference guide includes examples of each of the class methods.

Appendix C has the source listings for the PARODY database management system. These are provided for reference during the technical discussions that appear elsewhere in the book. Most programmers will not build PARODY by typing in the code but will get it from one of the sources described below.

▼ What You Should Already Know and Have

This book assumes a rudimentary understanding of the C++ programming language. I recommend my book, *Teach Yourself C++*, 3rd Edition, MIS:Press, as a starting tutorial for non-C++ programmers before they attempt an understanding of the techniques presented here. A reference list follows each of the chapters of this book and lists advanced texts that the C++ programmer will find useful.

To use the software, you will need one of the compiler products discussed in Appendix A. These compilers build the software with the MS-DOS operating system. As an alternative, you might use a different compiler and a different operating system. The software is independent of any particular platform, but it might not work everywhere because I do not have all possible platforms with which to test. If you find exceptions in language usage that prevent the code from working somewhere else, I want to know about them.

The programs in this book have a user interface provided by the command line interface of the standard C++ iostream class. Chapter 10 describes the generic user interface, which the GUI class defines.

Readers of C Database Development often asked when I would integrate its database management system with the screen management software that I published in other books. The answer is, I have no such intentions for that software or for this. PARODY is independent of other systems software beyond the requirement for a C++ compiler. You may integrate it with the user interface of your choice. It is bound to no particular architecture, and there are no portability layers for you to rewrite. It manages the database and leaves the user interface to the applications framework package of your choice.

▼ The Software Diskette

If you did not purchase the book/disk edition of this work, you can order the diskette separately from MIS:Press by using the procedures described at the

back of the book. I encourage you to work from the diskette for several reasons. First, the diskette will have the most recent version of the programs, which might be more recent than the program listings in the book. Next, the diskette will have makefiles and build procedures that reflect the most recent compiler versions, which, due to the dynamic nature of that market, will probably be more recent than the printed versions. Finally, in seven years of publishing C and C++ code in books and magazines, I observe that virtually all bugs reported by readers who type in the code are the result of typing errors. The readers often reason that the typing exercise gives them a better understanding of the code. There could be validity in that argument, but get the diskette for the code anyway. That way when we correspond about problems with the programs, we will be talking about the same version of source code.

▼ How to Get Help

The best—and often only—way to get in touch with me is with the CompuServe Information Service, a national on-line system with electronic mail and many topical forums where members share their knowledge and experience. I log onto CompuServe every day and will answer your questions about this book and its software. I am sorry, but I cannot promise to answer letters or return phone calls, and I do not have a FAX.

Besides giving you access to me, CompuServe is the official on-line support facility used by most software vendors including the vendors of the compiler products discussed later in this book. All you need is a personal computer with a modem and communications program and a CompuServe account. Most book stores can sell you a kit that has what you need to open a CompuServe account. Once you have an account you can log on, go into the MAIL section, and post messages directly. My account number is 71101,1262.

You can find help on CompuServe from the experts on Borland C++, Microsoft C++, Windows, DOS, UNIX, GNU, OS/2, and practically any other development or operating environment that you might use. The technical support available on CompuServe far outstrips any such support you get from a vendor's toll-free support number. CompuServe is as important a tool to a programmer today as are a good editor and debugger.

Other electronic mail delivery systems such as INTERNET have gateways to CompuServe; you can write to me through those gateways if you initiate the dialog and send me your E-mail address.

▼ Your Rights to the Software

The source code in this book is for use by anyone for the development of computer software applications. You can make as many copies of the source code as you need for yourself and the other programmers on the team. Those who use the code will, I hope, have their own copy of the book to help them along.

You may build and distribute applications by compiling and linking with the code that you get from this book. There are no royalties for its use, and I do not require you to acknowledge my contribution to your system. If the conditions under which you distribute your programs require you to include source code, then buy a copy of this book for each of your customers.

You may not publish the source code in a book, article, or other technical journal. If you are publishing an application that uses PARODY; please tell your readers where they can get the book and the code.

Chapter 2

An Introduction to Object-Oriented Programming

"Knowledge is the conformity of the object and the intellect."
Averroës

In the 1990s object-oriented programming and C++ in particular have begun to dominate the software development industry. C++ has been available in UNIX installations for years, and now most of the C compilers sold for MS-DOS systems include a C++ compiler. Many other contemporary software development systems use the object-oriented paradigm as their basis. As pervasive as

object-oriented programming has become, many programmers still do not understand it. This chapter presents to those programmers a brief introduction to object-oriented programming.

This book assumes that you have a working knowledge of C++. That experience alone has exposed you to the object-oriented paradigm. If you have built and used C++ classes, even in a tutorial or classroom exercise, then you have already written some object-oriented programs. There are, however, many terms that you will hear frequently. You should understand them so that you can relate your code to the discussions around you. Simply writing a C++ program does not qualify a programmer as an expert on object-oriented technology. There are certain rules and procedures to follow, and there are certain benefits to be gained by understanding and following the disciplines of object-oriented programming.

The traditional procedure-oriented programmer does not intuitively understand object-oriented programming. The notations and approaches to design are different than what you have learned and used in the past. Furthermore, explanations such as the one in this chapter do not usually complete the understanding of object-oriented programming. You need reinforcement—both from experience and by a feeling that there is something to be gained from a new and different approach. Programmers might understand the concepts at the intellectual level and yet not accept them as a pragmatic approach to programming simply because they have been writing good programs for a while and see no compelling reason to tamper with that success. For this reason, it is difficult to teach object-oriented programming, although it is not difficult to learn. Learning object-oriented programming is a process of discovery. Teaching it, therefore, becomes the management of that process.

Many of the current generation of programmers have learned and accepted object-oriented programming as a better way to express software algorithms. Most of them started from the procedural paradigm, having programmed in traditional languages such as C. Those who make the switch become object-oriented advocates. Enough of them have done so that the rest of us cannot deny that there is something to it. You can take a lesson from the observation that once a programmer takes the plunge, he or she almost always becomes a convert. Therefore, the best way to learn object-oriented programming is to try it. Until recently, however, the tools were not widely available. Now, virtually every PC can have a C++ compiler, and virtually every programmer has access to the tools.

▼ The Basics

The first understanding of object-oriented programming is to be found in this design guideline:

The expression of an algorithm should model the application domain that it supports.

This concept reflects an older one that states that the solution to a problem should resemble the problem that it solves, a guideline that allows observers of a solution to recognize its purpose without necessarily knowing in advance about the problem. When you see a properly designed word processor, you will intuitively understand that the problem being solved is one of capturing and manipulating text. When you see a properly designed inventory management system, you will recognize that its purpose is to maintain stock quantities and locations. You recognize those things because the designs resemble and therefore remind you of the problems that they solve.

Carrying this concept to a higher level of abstraction, you will recognize that the purpose of a programming language is to express with algorithms the solution to a data processing problem. The techniques used in that expression will determine how successfully the solution models its problem domain. Object-oriented programming is one of several different approaches to the expression of algorithms, and it is often misunderstood—primarily by those who do not use it.

▼ Procedural Programming

The classic approach to programming designs a set of data structures first followed by some functions and procedures to process the data. This approach is called *procedural programming* because it starts with the procedures. We are used to thinking of programming in this way because that is how programming has been done for the past forty years.

Procedural programming does not always deliver a solution that resembles the problem, however, because it emphasizes the functions rather than the data—the procedures rather than the objects. Furthermore, procedural programming does not encourage the programmer to separate and hide the procedures related to different data objects from one another. Programmers

have long known that those are worthwhile practices, but procedural programming languages do not encourage them.

▼ Object-Oriented Programming

The world and its applications are not organized into values and procedures separate from one another. Problem solvers do not perceive the world that way. They deal with their problem domains by concentrating on the objects and letting the characteristics of those objects determine the procedures to apply to them. To build a house, grow a tomato, or repair a carburetor, first you think about the object and its purpose and behavior. Then you select your tools and procedures.

The world is, therefore, object-oriented, and the object-oriented programming paradigm expresses computer programs in ways that model how people perceive the world.

The Object-Oriented Program

Object-oriented programming is described with a vocabulary of terms that are used in unfamiliar ways. You will hear these terms used frequently in object-oriented programming discussions:

- abstract base class
- abstract data type
- abstraction
- base class
- class
- derived class
- encapsulation
- implementation and interface
- inheritance
- instantiate

- message
- method
- multiple inheritance
- object
- polymorphism
- subclass
- superclass

An object-oriented program has four fundamental characteristics:

- *Abstraction* defines new data types.
- *Encapsulation* defines a data type's representation and its behavior in one encapsulated entity.
- *Inheritance* derives a new data type from an existing one.
- *Polymorphism* customizes the behavior of a derived data type.

The program defines *abstract data types* in *classes* each of which has its own *implementation* and *interface*. Inherited abstract data types are *derived subclasses* of *base* classes. The program *instantiates* an *object* and sends *messages* to the object by using the object's *methods*.

The object-oriented programming community uses these terms universally. Therefore, when object-oriented programmers say "encapsulate," for example, you know that their meaning is consistent with the way others use it. When programmers acknowledge that a program is "object-oriented," you know that the program contains abstraction, encapsulation, inheritance, and polymorphism, and that it defines abstract data types, instantiates objects, and sends messages to the object's methods. The discussion that follows draws on your experience with C++ to explain these terms.

The Object

The first question that programmers often ask is, "What are the objects in object-oriented programming?" What about object-oriented programming sets it apart from traditional programming? Early writings on the subject effectively explained object-oriented programming to those who already understood it, but their explanations were sometimes too abstruse for the newcomers.

Simply stated, an object is an instance of a data type. The program in Figure 2.1. declares two objects. The first object is a simple integer; the second object is an *abstract data type*.

```
voif f()
{
    int ndays; // an instance of an int
    Date cdt;  // an instance of an ADT
    // ...
}
```

Figure 2.1 *A Program with Two Objects*

Abstraction

Abstraction is the definition of an abstract data type. The definition includes the data representation and the data type's behavior. An abstract data type is new. It is not one of the primitive data types which are built into the programming language. The **int**, **long**, and **float** data types are primitive C++ data types. Their data representations and behavior are known to the compiler. A primitive data type's format and response to arithmetic, assignment, and relational operators are defined as a part of the C++ language.

An *abstract data type*, however, is not known to the language; the programmer defines its format and behavior by defining a C++ *class*. For example, the calendar date could be an abstract data type. The compiler and the computer do not know about calendar dates. Programmers have always had to define the behavior of dates by designing structures and functions. The standard C library has several such date definitions. When you define a C++ calendar date class like the one in Figure 2.2, you express its format and behavior very much like you did in C. It will have month, day, and year data members. It might even support some arithmetic and relational operations.

```
class Date  {
    int month, day, year;
public:
    Date(int mo, int da, int yr);
    int operator+(int n);
    // ...
};
```

Figure 2.2 *An Abstract Data Type (ADT)*

You can then declare an object that has the type of the date class and add to the object, subtract from it, compare it with other objects, and assign values to it. A C++ program declares instances of abstract data types the same way it declares instances of primitive data types. Refer again to Figure 2.1. The declaration of the **cdt** object is an instance of an abstract data type.

C++ programmers often use string classes to implement a string abstract data type similar to the strings of the Basic programming language. Other abstract data types might be container classes for stacks, lists, trees, and queues, perhaps with ordered and random access variants of such containers.

Another term you will hear used is *instantiate*, a word coined by object-oriented programmers for convenience. When an object-oriented program declares an instance of a data type, the program has *instantiated* an *object*, regardless of whether the data type is primitive or abstract. The program in Figure 2.1 has instantiated the **ndays** and **cdt** objects.

Encapsulation

Encapsulation is the design of a C++ class. A programmer encapsulates the data representation and behavior of an abstract data type into a class definition, thereby giving it its own *implementation* and *interface*. Figure 2.3 is an example of an encapsulated class.

```
class Date {
// --- the class implementation
private:
    int month, day, year;
// --- the class interface
public:
    Date(int mo, int da, int yr);
    Date operator+(int n);
    // ...
};
```

Figure 2.3 *Encapsulation*

The implementation of a class, which consists of the data members and private member functions, is hidden from the using program. The **month**, **day**, and

year data members in Figure 2.3 are the class's implementation. A user of the class does not care about the details of implementation—only that it works. The class designer could totally change the implementation—perhaps changing the date representation to a long integer count of days—and the using program would not be affected.

The interface, which is visible to the user of the class, consists of the public member functions. The class user reads and modifies values in the data representation by calling public member functions. The class interface in Figure 2.3 consists of the constructor, which contains initialization parameters; and an overloaded plus operator, which presumably allows the user of the class to add an integral number of days to the date.

To use the abstract data type defined in Figure 2.3, a programmer will make assumptions about the interface based on its appearance and the programmer's understanding of C++ syntax. If the programmer is not the class's author, those assumptions could be invalid. The author might not have designed an intuitive interface. In this case, most programmers would assume that the overloaded plus operator adds an integer to a **Date** object and returns another **Date** object with the result. Figure 2.4 illustrates how the class interface will work if it is intuitive.

```
voif f()
{
   Date dt(6,29,92);
    dt = dt + 30; // should now be 7/29/92
}
```

Figure 2.4 *An Intuitive Class Interface*

Designing an intuitive class interface is not automatic. The C++ language provides the tools for you to do a proper job, but nothing in the language enforces good design. The class author can use unnecessarily clever techniques that obscure the meaning of the interface. An object-oriented design is not a good design by default. Experienced C++ programmers generally agree on certain design standards for C++ classes. The data members are usually private and constitute most of the implementation. The interface consists only of member functions that restrict access to the data members. The interface is generic in that it is not bound to any particular implementation. The class author should be able to change the implementation without affecting the using programs. These apparent rules are only guidelines, however. Some-

times you need to step around them to achieve some purpose. The fewer times you do that, the stronger your design.

Methods and Messages

Method is another name for C++ public member functions. Methods may be constructors, destructors, procedures, and overloaded operators. They define the class interface. The constructor and overloaded plus operator in Figure 2.3 are methods.

A *message* is the invocation of a method, which, in C++, is the same thing as calling the public member function. The program sends a message to an object telling it to invoke the method and sometimes provides parameters for the method to use.

There are different kinds of methods characterized by how they support the class definition. There are *functional* methods, *data type* methods, and *implicit conversion* methods. Note that this delineation and these terms are coined here for convenience and are not part of the official object-oriented lexicon. They define different levels of support that C++ provides for class design.

Functional Methods

Figure 2.5 shows three methods added to the **Date** class.

```
class Date  {
    // ...
public:
    // ...
    void Display();              // display the date
    void AdjustMonth(int m);     // +/- m months
    int DayOfWeek();             // return 0-6 = Sun-Sat
};
```

Figure 2.5 *Functional Methods*

The three methods in Figure 2.5 illustrate three typical method variants. The first method tells the object to do something, in this case to display itself. The class's implementation will know how to display the object's data representation. The programmer who uses the class is unconcerned about the details of the implementation.

The second method tells the object to change itself, in this case to adjust its month data value up or down some number as specified in the method's parameter. This method is an example of one that includes a parameter. Once again, the using programmer does not care how the object stores the month value or what algorithm adjusts the month, only that it works.

The third method is one that returns a value, in this case the day of the week represented by the object's current value. The methods in Figure 2.5 define behavior related to the functional properties of the class. The class is a calendar date, and you want it to behave like a date.

Data Type Methods

Data type methods make a class act like a primitive data type by giving it the properties of a primitive data type. These properties are usually implemented as overloaded operators in the class interface. Figure 2.5 shows the methods to compare dates, assign them to one another and do some arithmetic on them.

```
class Date  {
    // ...
public:
    // ...
    // ---- arithmetic operators
    Date operator+(int n);
    Date operator-(int n);
    int operator-(Date &dt);
    // ---- assignment operators
    Date& operator=(Date &dt);
    // ---- relational operators
    int operator== (Date &dt);
    int operator!=(Date &dt);
    int operator<(Date &dt);
    int operator>(Date &dt);
};
```

Figure 2.6 *Data Type Methods*

Implicit Conversion Methods

The C++ language handles implicit conversion of primitive data types. If you write an expression with an **int** where the compiler expects a **long**, the compiler knows how to make the conversion and does so quietly. If, however,

you write an expression where the compiler expects an abstract data type, and you provide a different data type, primitive or abstract, the compiler does not know how to deal with that. Similarly, if the expression expects a primitive data type, and you use an abstract data type, the compiler does not know what to do. In either case, unless you have provided implicit conversion methods, the compiler issues an error message and refuses to compile the program.

You can add implicit conversion methods to a class so that the compiler will know how to convert from one data type to another. The conversion constructor function converts a data type to an abstract data type, and the member conversion function converts an abstract data type to a primitive C++ data type. When you code an expression where the compiler expects to see something other than what you provide as just described, the compiler will execute one of your implicit conversion methods to convert the data type. This is an example of a method that you do not explicitly call (or send a message to) from within your program. It executes as the result of the implicit call inferred by the compiler by its interpretation of your expression. Your program implies a call to the method by the context in which it uses data types in an expression.

Figure 2.7 shows two implicit conversion methods added to the **Date** class. The first method converts an **int** data type to a **Date** object. The second method converts a **Date** data type to an **int** object. The conversions are not restricted to converting between abstract data types and primitive data types. You can convert between abstract data types in the same manner. For example, you might have a **Date** class and a **JulianDate** class, and the same principles apply.

```
class Date {
    // ...
public:
    // ...
    Date(int n);          // conversion constructor
    operator int();       // member conversion function
};
```

Figure 2.7 *Implicit Conversion Methods*

Member Functions

Having just learned about methods and messages in this much detail, you will soon find that the terms themselves are not used much in C++ circles. The terms come from the object-oriented lexicon and reflect the syntax of pure

object-oriented programming languages such as SmallTalk. Most C++ programmers prefer to say that they call member functions rather than saying that they send messages through methods. Nonetheless, you should understand the analogy, because you will read and hear the object-oriented terms frequently. Beyond this chapter, however, this book uses the C++ conventions and refers to methods and their messages as class member functions.

Inheritance

A class can inherit the characteristics of another class. The original class is called the *base* class, and the new class is called the *derived* class. You will also hear these classes called the "superclass" and the "subclass." The word "subclass" is sometimes used as a verb to mean the act of inheriting as well. This book will refer to base classes and derived classes exclusively.

The derived class inherits the data representation and behavior of the base class except where the derived class modifies the behavior by overloading member functions. The derived class adds behavior that is unique to its own purpose.

A program can instantiate objects of a base class as well as those of a derived class. If the base class is an *abstract base class*—one that exists only to be derived from—the program may not instantiate objects of the base class.

Inheritance is the foundation of most object-oriented designs, and it is often over-applied. Some programmers get carried away with the power of inheritance, and C++ can offer some surprises to the unwary designer. Many designs use inheritance to solve problems that would be better supported by a different approach, usually an object member of the base class in what is otherwise the derived class. Despite this warning, inheritance is a powerful feature, which when properly used, offers a rich design capability to the object-oriented programmer.

Single Inheritance

The C++ inheritance mechanism lets you build an orderly hierarchy of classes. When several of your abstract data types have characteristics in common, you can design their commonalities into a single base class and separate their unique characteristics into unique derived classes. That is the purpose of inheritance.

For example, a personnel system maintains information about employees. Employees are people, and people have common characteristics—name, ad-

dress, date of birth, and so on—yet the system might record different kinds of employees. Managers, project workers, and support personnel might be recorded differently. Therefore, you could design a base class of employees that stores name, address, social security number, and date of birth and then derive separate classes for managers, project workers, and support personnel. Each of the derived classes would inherit the characteristics of the employee class and would have additional characteristics common to themselves. For example, the manager class might include an annual salary-augmenting bonus data member that the other employee classes do not have. The project worker class could have a list of project assignments. The support personnel class could record overtime hours worked. Figure 2.8 illustrates such a class hierarchy.

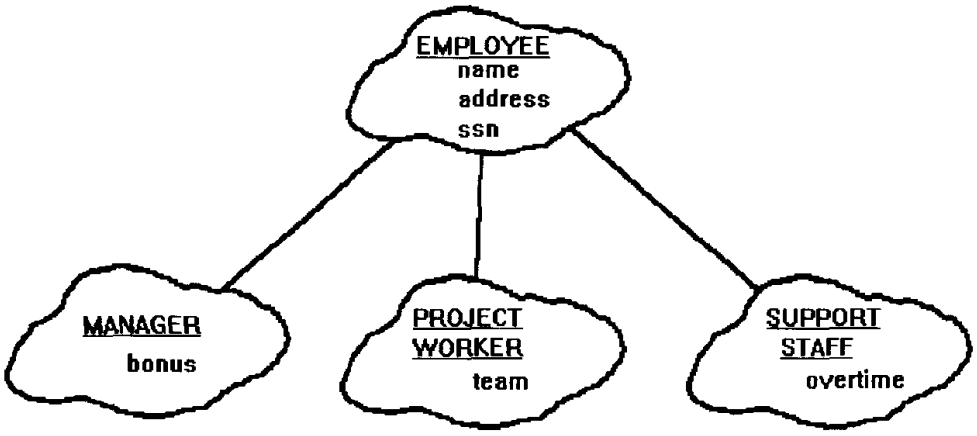

Figure 2.8 *An Inherited Class Design*

Object-oriented designers strive to use inheritance to model relationships where the derived class "is a kind of" the base class. A car "is a kind of" vehicle. An engineer "is a kind of" employee, which "is a kind of" person. This relationship is called the ISA relationship. Inheritance is not appropriate for relationships where one class has an instance of another. An employee "has a" date of birth. A department "has a" manager. This relationship is called the *HASA relationship*. Instead of using inheritance, the class that "has" an object of another class embeds the object as a data member.

Multiple Inheritance

A derived class can inherit the characteristics of more than one base class. This technique is called *multiple inheritance*. Not all object-oriented programming languages support multiple inheritance, and many experts assert that it is not

a necessary design tool. Nonetheless, C++ does support multiple inheritance, and there are times when you will find it a good way to express class relationships.

You will recall that the effectiveness of a programming language can be measured in its ability to model the problem domains that it will support. The objects in the world reflect membership in multiple-inheritance hierarchies, and programmers are called upon to write programs to model those objects. A sofabed is a sofa and it is a bed, both of which are items of furniture. An amphibious airplane is a boat and an airplane, both of which are vehicles. A carphone is not a car and a telephone, however, and sometimes programmers get their designs confused by thinking that they need to inherit when they do not.

Suppose that the system supported by the class design in Figure 2.8 needed to store the support personnel class as an entry in a linked list. The support person class "is a kind of" employee, and now it "is a kind of" linked-list entry, too. The other employees are not listed in that manner, so their classes do not need to inherit the characteristics of the linked list. You might argue that in life a person is not a linked-list entry and that this solution does not clearly model the real-world problem. That argument would depend on what the linked list does and who is reading the design. If the linked list is a convenient way to represent the employee's seniority ranking in a union membership, for example, then the relationship is valid because the employee "is a kind of" union member. That might not be clear to everyone who reads the design documentation, so use notation that clearly describes the purpose for the class relationship. If, on the other hand, you are using the linked list to maintain some arbitrary sort sequence for the support class, then perhaps a different design approach would be more appropriate. Sometimes you have no choice. This book uses a linked-list class that must be inherited because it uses its own object address as a pointer to the entry.

Figure 2.9 shows how you would derive the support personnel class from the linked-list class as well as the employee class by using multiple inheritance.

Polymorphism

Polymorphism is when a derived class customizes the behavior of the base class to meet the requirements of the derived class. A C++ base class uses the

virtual member function to specify that overriding member functions in derived classes will have polymorphic behavior with respect to that method.

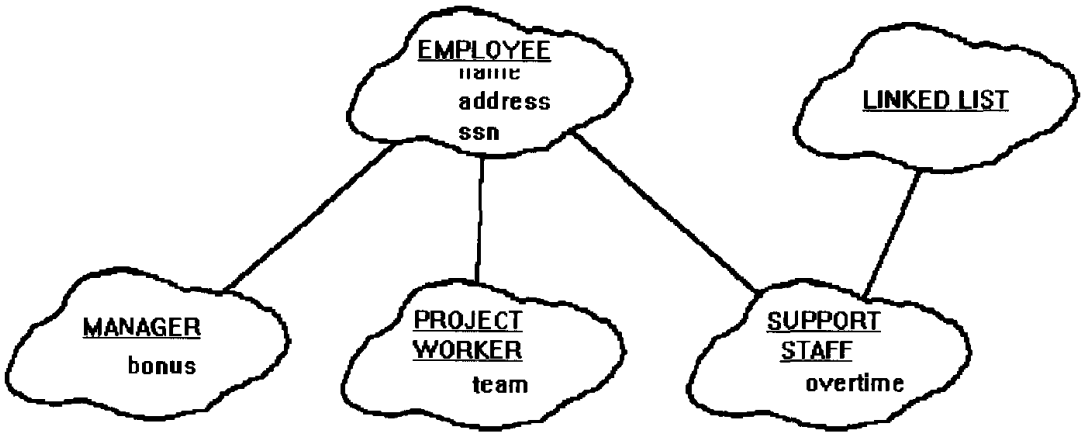

Figure 2.9 *Multiple Inheritance*

If a derived class overrides a base class method, and the base class method is not a virtual function, then the overriding behavior is effective only when the compiler is de-referencing a pointer, reference, or object of the derived class itself. If the object's pointer or reference refers to the base class, then the base class method has precedence. Such behavior is not polymorphic. However, if the base class method is virtual, then the compiler will select the overriding derived class method regardless of the type being de-referenced by the compiler.

Suppose, for example, that the support staff class from Figure 2.8 was further decomposed into derived classes that represented the various kinds of support personnel. You could have typists, corporate pilots, chauffers, maintenance personnel, instructors, and so on. The system measures skill levels differently for each of these disciplines, yet there is a requirement for a general-purpose skill index for the base support class. Each derived class exhibits different behavior for data entry and retrieval of the skill index, but some parts of the system will invoke the skill method without knowing which kind of support personnel object is involved. The polymorphic skill method would modify the class's behavior at run-time based on the type of the derived class. Figure 2.10 shows how you might design such a class.

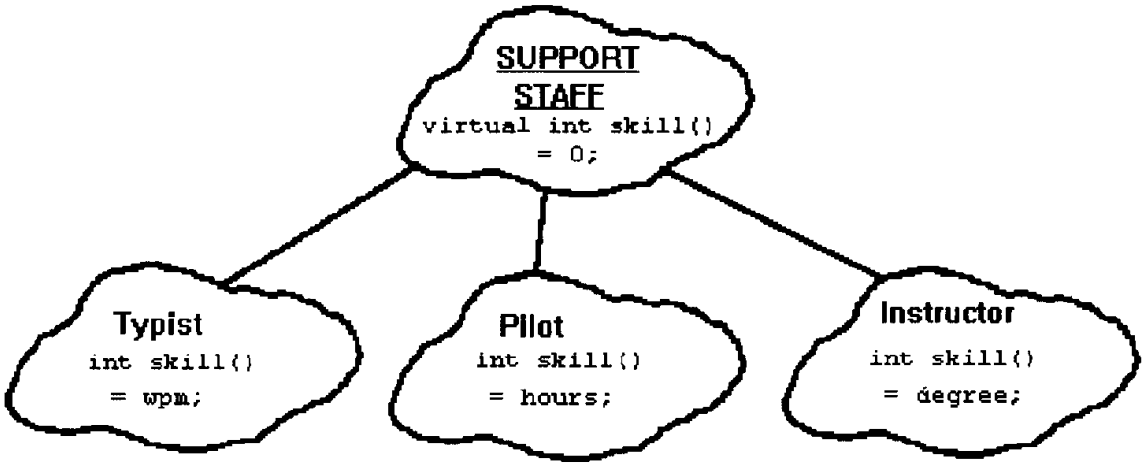

Figure 2.10 *Polymorphism*

▼ Summary

The object-oriented programming paradigm is a rich environment for the expression of the data formats and functions for an application. It is not necessary, however, for C++ programmers to emerse themselves totally in the object-oriented passion. The availability of improved design and programming methods does not automatically outdate all the traditional approaches. C++ has the facility to support the basics of object-oriented programming while permitting the programmer to use traditional procedural programming where it seems appropriate. C++, in fact, encourages that approach. By supporting traditional C flow control of nested functions, C++ allows you to leverage your existing investment in mature and useful C function libraries. Furthermore, C++ does not force a pure object-oriented hierarchical data structure where every data type descends from one generic root base object. Instead, C++ allows you to build a number of class hierarchies representing the different problem domains that your application might deal with. There can be classes that define the data structures of the application's functional purpose; there can be classes that supply general-purpose data structures such as strings, lists, queues, and so on; there can be classes that integrate your application with a particular user interface. All these classes can co-exist independently of one another in a system of hierarchies integrated by you into an application.

▼ References

C++ Programming Style, Tom Cargill, 1992, Reading, MA: Addison-Wesley.

Object-Oriented Design with Applications, Grady Booch, 1991, The Benjamin/Cummings Publishing Company, Inc.

The C++ Programming Language, Second Edition, Bjarne Stroustrup, 1991, Addison-Wesley.

Chapter 3

A Brief History of Database Management

"Why is not all nature in confusion, instead of the species being, as we see them, well defined?"

Charles Darwin

This chapter is about the progress of database management technology from the earliest ad hoc, hierarchical, and network models, through the enduring relational model, and into the newest model, the object database, which is the subject of this book. The discussions of the earlier models are here for purposes of historical interest and to prepare you for what you are about to learn. We

must understand our past in order to manage our present and face our future. You are encouraged to learn more about the earlier database models by reading the references cited at the end of the chapter.

The history of database management traces the history of data processing itself. Ever since there have been information processing machines, programmers have had to design ways to organize data records into formats that the machines could process. Although we are primarily concerned with computers as data processing machines, the science of database management traces itself back to machines that processed information recorded in other forms, such as punched cards and paper tape. The common characteristic shared by all database management approaches from the first is reflected in this self-evident concept:

An entity of data must take a form that the computer can recognize.

An entity of data is called a *data element*. Aggregate entities of data are called *records*. Collections of like records are called *files*. To be able to manage files of records, a database management system assumes that this rule applies:

All like records assume a common form.

Given a file of records, all of which adhere to a common form, a database management system can add records to the file, retrieve records from the file, change records in the file, and delete records from the file.

A record consists of data elements. A data element is a singular entity of data that can take a value that has functional meaning to the particular record. Data elements can be names, addresses, dollar amounts, hat sizes, batting averages, dates, and so on. A data element may consist of one or more *data types* in a structure. For example, a calendar date consists of the integers day, month, and year.

A collection of files that supports a particular application is called a *database*. The information resource of an organization might consist of a number of independent databases. Some databases have points of interfaces with others. For example, the payroll database might contribute transactions to the general ledger database.

The software that manages a database is called, appropriately, a *database management system*.

▼ The Database Management System

The purpose of a database management system is to store, maintain, and retrieve database records in files. A file collects records of the same format that serve a common purpose. A payroll system might have a file of employee records, for example. There would be one record for each employee. The database management system lets your program add new employees when the company hires them, delete them when they resign, change an employee's record when the employee gets a raise, and retrieve the employee's record once every pay period to calculate the employee's paycheck.

A database management system can be the custom code that an application uses to access the database, or it can be a general-purpose package of systems software. Programmers discovered long ago that there are database design and programming considerations common to most applications, and that those commonalities can be supported by a general-purpose system of database management software. The advantage of a database management system is that system developers do not repeat the development of database management software for each new application. The development team designs a system that the database management system can support and then writes programs that interface with the database management system to create, process, and maintain the data.

The introduction of general-purpose database management systems mandated the definition of certain data models that system designers would use. These models evolved over time, and their evolution reflects the industry's experience with the applicability of each of them to a wide range of applications and the inherent stability of databases that used them.

General-purpose database management systems use four language interfaces between the application programming language and the database manager: the data definition language, the data manipulation language, the query language, and the report writer language. Depending on the database management system, these interfaces are: (1) built into the programming language; (2) a separate language processed by a separate language processor; or (3) some combination of the two.

The Data Definition Language

The Data Definition Language (DDL) defines the format—or *schema*—of the database. It identifies files, record formats, and the relationships between files. The DDL is usually an external file of statements that a database builder utility program uses to establish the first, empty copy of a new database. Subsequently, the database manager uses the DDL to specify the format of the files and records to support calls to the Data Manipulation and Query Languages.

The Data Manipulation Language

The Data Manipulation Language (DML) is the applications program interface to the database management system. It provides the program interface to open and close a database, find records in files, navigate among the various file records, add new records, and change or delete existing records.

The Query Language

The Query Language is a general-purpose technique for describing a search of the database for records that match the search criteria. Some query languages are processed by utility programs, and others are implemented as embedded extensions to the application's programming language. A query generally returns a temporary subset of records from the database. The query subset can often have its own format independent of record formats in the formal database definition, and sometimes the subset can consist of records formed from the logical combination of related records in the database.

The Report Writer Language

Some database management systems offer a report writer utility program that uses the output from a query to generate a report. The report writer language specifies the format of the presentation of the data values on the report, usually in row/column tabular form with report headers and footers and the control breaks for totals and subtotals taken as a function of the sequence of the data.

▼ Database Models

Early database models were ad hoc models. An application used whatever set of data files the designers came up with. If there was a database management

system, it was usually a set of custom subroutines developed for the application itself. The disadvantages to this practice are apparent, and they were observed and recorded early on. In fact, they led to the development of the first general-purpose database management systems. Nonetheless, many systems were—and still are—developed this way for a lot of reasons.

Proprietary Data Models

Most spreadsheets, word processors, and other horizontal applications use proprietary database formats. There are three reasons for this: (1) their vendors prefer a closed architecture that other software vendors cannot exploit; (2) the use of a general-purpose database management system requires the cost and performance overhead of the database management system itself over and above that of the application; and (3) most implementations of the three traditional database models—discussed next—do not support variable length data elements, lists, queues, and so on, that the applications need to store.

The Hierarchical Data Model

The first acknowledged formal data model was the hierarchy. Programmers observed that many databases consisted of files in a hierarchical relationship. An organizational database would have files of companies, divisions within each company, and departments within each division. Figure 3.1 illustrates the hierarchical relationships of such a database.

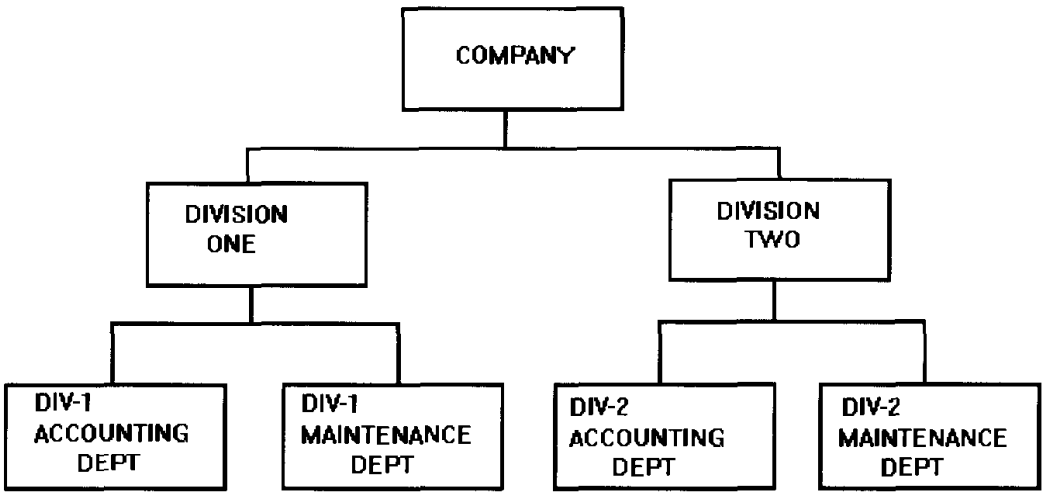

Figure 3.1 *A Hierarchical Database*

A file higher in the hierarchy is a *parent* file that contains parent records. Subordinate files contain *child* records. A child record may have children of its own; it can be a child to one level and a parent to another. In the hierarchical data model, a child record type may have only one parent record type, and each child record instance may have only one parent record instance.

Each department in Figure 3.1 might have its own independent hierarchical database as well. The maintenance department would have a file of equipment maintenance schedules. The accounting department would have files for the general ledger, accounts receivables, accounts payable, and payroll.

The hierarchical nature of these files indicated that a common system of software could support all of them if it knew their formats and interrelationships. Systems of file descriptions called the *schema* evolved, and programmers could define the files and attach their applications programs to the general-purpose database management system.

Hierarchical database management systems supported the interfile relationships by embedding physical pointers in the data records. The company record would have the disk address of the first division record assigned to the company. Each division record would have the disk address of its parent company as well as the disk address of the next division that belonged to that company. The disk record addresses were invisible to the applications programs, which would ask the database manager to retrieve the first and subsequent divisions for a particular company. From a particular division record, the program could ask for the first and subsequent department records, and so on.

The Network Data Model

The weakness of the hierarchical data model is that it would not support interfile relationships where a child record could have more than one parent. If the organization decided to reorganize and allow one accounting department to support more than one division, for example, the hierarchical data model could not support that relationship. Although the hierarchical model supports many applications effectively, there were others that required multiple parent records for a child record. The network data model evolved to support those requirements. Figure 3.2 shows the organization's database reorganized into a network.

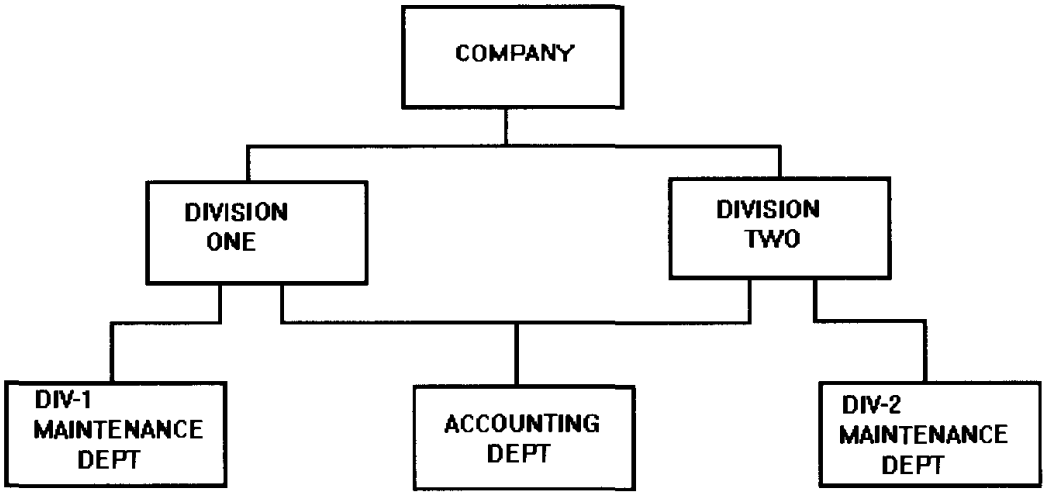

Figure 3.2 *A Network Database*

Network databases were usually implemented with record address pointers like those of the hierarchical data model except that more than one parent record could point to a particular child record, and the child record pointed to a list of parents.

In 1969 the Conference on Data Description Languages (CODASYL) formalized the definition of network databases with the development of a standard for a data description language. A database management system that uses the CODASYL standard is said to implement a CODASYL database. The language itself resembles the data division statements of the COBOL programming language, and it organizes records into named sets that consist of owner records and member records. A set follows the form of a hierarchy—a parent can have many children, and each child can have only one parent. Sets can overlap with respect to the records included in them, however, and a child record can have multiple parents if the parents are in different sets.

The Relational Data Model

The relational model evolved out of concern that the hierarchical and network models were, due to the their use of hidden physical pointers in the data records, vulnerable to the compromise of data integrity, and therefore, inherently unstable. System errors would cause broken pointer chains, and the relationships between records were compromised.

The solution to this concern was the relational data model, where the relationships between files are represented by data values in the file records themselves rather than by physical pointers. This does not fully solve the problems of data disintegrity. It is possible for a system error to cause an employee record to point to a nonexisting department, for example. However, the process of error identification and correction is simpler when data values rather than hidden pointers are involved.

Files in a relational database consist of fixed-length, fixed-format records, which form tables of rows and columns. Each file record has a *primary key* data element to uniquely identify the record. This requirement models the way systems traditionally identify things. People have Social Security numbers or employee numbers. Spare parts have part numbers and stock numbers. Anything we can identify has at least one piece of identifying information that sets it apart from other objects in the same category.

A record in a relational database file is *flat* in that it has no arrays—lists of common data elements—especially none with a variable number of entries.

File relationships in the relational model are maintained by data item values. A department record contains the division identification that it belongs to. The division identification data element in the department record is a *secondary key* data element, which means that the application can retrieve a set of department records that has a specified division identification. Figure 3.3 shows a typical relational database.

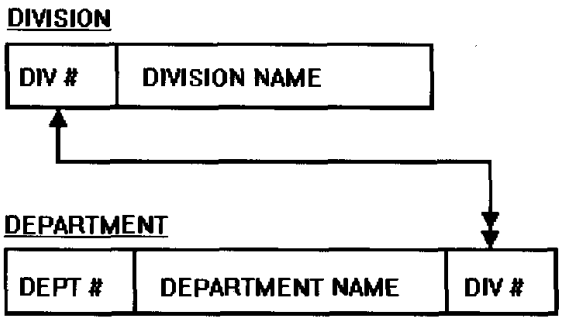

Figure 3.3 *A Relational Database*

The arrow icon that links the two record definitions in Figure 3.3 is not a physical pointer. Instead, it represents the relationship implied by the division number in the department record. Inasmuch as the division number data element is also the primary key data element of the division record, several department records can be related to a single division record, and only one

division record can be related to any department record. This is called a *one-to-many relationship* and is indicated by the single and double arrowheads at the two ends of the arrow icon in Figure 3.3.

The one-to-many relationship between divisions and departments does not support the problem where one department can be assigned to multiple divisions, however. The relational data model uses a connector file such as the one shown in Figure 3.4 to support the *many-to-many relationship* between data files.

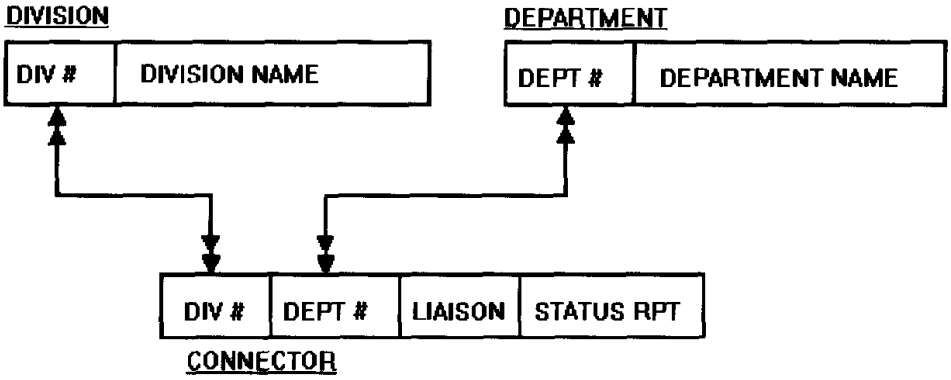

Figure 3.4 *The Many-to-Many Relationship*

The concatenated division and department number data elements form the primary key to the connector file in Figure 3.4. Either number by itself also forms a secondary key. The application can thus find all the division records related to a given department record and all the department records related to a given division record. It might even identify data items that are unique to the connection itself and put them in the connector record as well. For example, the connector record might store the employee number of the division's liaison in the department and the date of the department's next scheduled status report to the division.

It has been shown that any interfile data relationships that you can represent with the hierarchical or network models can be represented by the relational model just as well through a process called *normalization*. This process examines a database's file relationships and reorganizes the data elements in several steps designed to eliminate redundancy, internal file pointers, and repeating groups (lists) of data elements in the records. Proponents of the relational model assert that the resulting gain in data stability far outweighs any overhead that the approach might add to the database.

The relational model has become the standard method for representing data in a database. The reasons are because of the inherent strength of the model, the fact that the entire data file format is visible to the user, and the relative ease with which a general-purpose relational database management system can be designed and developed when compared to the effort required to build, for example, a CODASYL database manager.

The Object Data Model

Object-oriented programming does not, by itself, change the traditional requirements for database management. The developer of an application still has to identify data elements and organize them into files for retrieval and maintenance. What object-oriented programming adds to the culture are improved ways to represent data inside the program, including the ability to bind the behavior of an object to the data values that represent it, the ability for new data types to inherit behavior from old ones, and the ability to use canned abstract data types from class libraries.

Certainly, an object-oriented program can use a traditional database management system. To build or modify a database record, the program can collect the data values from the objects into a record and write the record to the database. To rebuild an object, the application can retrieve its record from the database management system and distribute its data values into the data members of an object that has been instantiated to receive the data.

Before committing to such a strategy, however, it is proper to consider what there is about an object-oriented design that lures the designer away from the traditional relational database.

Data Types

The relational database assumes a flat file of fixed length, fixed format records. There are no variable-length data items, no repeating groups, and no abstract data types within the record's definition.

These restrictions move us to look for an alternative to the relational database, but the last one—the absense of abstract data types—is particularly compelling. One of the strengths of the object-oriented paradigm is the ability to design abstract data types. An object-oriented application will have them because programmers find in them the solution to many classic problems. If

the database management system does not support the abstract data type, the programmer will have to intercede between the parts of the design that define classes and the parts that manage the database.

Consider those other restrictions. If you cannot store a variable-length object in your database, what applications are left out? Many applications use variable-length database objects. CAD/CAM programs store digital renderings of engineering drawings. Cartographic systems store variable-length lists of geographic coordinates, feature renditions, and free-form map legends. Imagery systems store the compressed raster scans of digitized images. Multimedia audio and video data streams have widely varying lengths. Word processors use data streams to record documents, macros, style sheets, and graphics. Electronic mail systems store variable-length files of text and attachments. Simulations can consist of anything at all. The relational database is not much use to these applications, and it was never intended for them; but until the object model was defined, those data types were not provided the kind of support that they require.

Class and Database Design Notation

A natural tendency exists to confuse the hierarchical relationships among database files with those among classes in an object-oriented design. This tendency is reinforced by the close organizational similarities between the two. In the database, the employee is subordinate to the department. In the class design, the engineer is derived from the employee. The difference is subtle and can be obscured by the way that we represent the relationships in our design notation.

Think of the differences this way: an engineer *is* an employee. The class that defines the engineer inherits all of the characteristics of the employee class and adds some unique ones of its own. The employee, however, *is not* a department. The department *employs* the employee instead, and the employee does not inherit any of the department's behavior.

When you design a class to be a data record, you design the data file's record format, and the format includes all of the data elements that the class has, *including the data elements that it inherits from its base class.*

When you design a child record, you design the relationship between the record and its parent. The two records might have only one data element in common, the primary key of the parent record.

To distinguish these two kinds of relationships, this book will use two distinct design notations as shown in Figure 3.5.

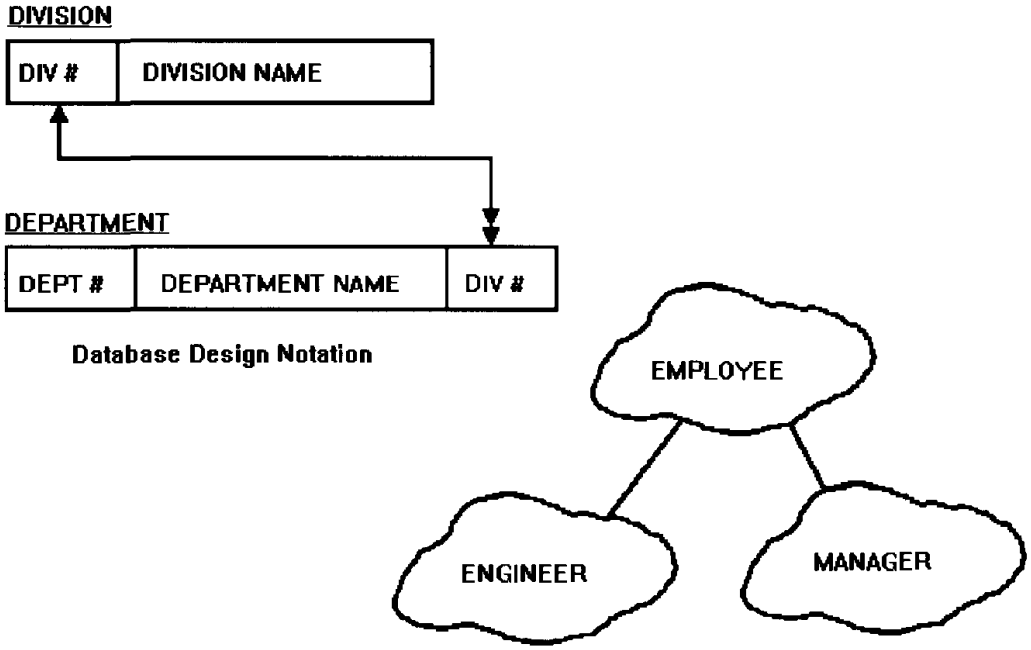

Figure 3.5 *Class and Database Design Notations*

Many object-oriented design texts employ notation that uses a common symbol to represent classes and uses different pointer notation to distinguish between what Chapter 2 called *ISA* (as in "an engineer *is an* employee") from *HASA* ("an employee *has a* department"). These texts usually address the design of object-oriented applications and do not encounter the collisions that occur between the notation of class design and that of a database design.

There are interclass relationships beyond the simple ISA and HASA types, and the succinct object-oriented design notations of other texts allow for them. The examples in this book do not use complex class design, however, being primarily aimed at presenting and explaining the PARODY database manager and its use. For purposes of clarity, therefore, this book adopts the more visually distinct notation shown in Figure 3.5.

▼ Summary

This chapter discusses the evolution of database management technology and presented an overview of what an object database is. Implementing such a database, particularly as a general-purpose class library to support the persistence of as-yet undefined class designs, is no trivial task. Chapter 4 compares the traditional techniques for storing C structures with those for storing C++ classes and describes some of the problems you encounter when you try to support that from a common, general-purpose database manager.

▼ References

C Database Development, Second Edition, Al Stevens, 1991, New York: MIS:Press.

A Guide to the SQL Standard, C. J. Date, 1987, Reading, MA: Addison-Wesley.

An Introduction to Database Systems, Second Edition, C. J. Date, 1977, Reading, MA: Addison-Wesley.

Computer Data-Base Organization, Second Edition, James Martin, 1977, Englewood Cliffs, NJ: Prentice-Hall.

Chapter 4

C++ and Persistence

"...persistence covers every thing from writing an object out to disk and getting it back to having a full-blown distributed object-oriented database system with concurrency control, transaction logging, etc., for multiple users."

Bjarne Stroustrup

This chapter discusses how C++ programs store data structures on disk and how those traditional techniques do not always work with C++ objects in a class inheritance, object-oriented design. The problems presented here cause programmers to reconsider how they think about file and database management. Ultimately you must address those problems in any design where object persistence is a concern. Later chapters help you do that. First, however, you must identify and understand the problems.

▼ File Input/Output in C

Reading and writing data records in C is a relatively simple matter. You define a structure, put some data in it, and write it to a disk file. A generic file input/output function can read and write the structure by using the **sizeof** operator. This method is self-adjusting when you modify the program to change the contents of the structure. If you need more features, you use a DBMS, but the solution still centers around the basic flat record structure, and so the C solution looks like Figure 4.1.

```
struct Employee   {
    /* ... */
} Employee empl;

fread(&empl, sizeof(struct Employee), 1, dp);
```

Figure 4.1 *Persistence in C*

The C solution is not perfect, however. If the structure contains a pointer, the generic file input/output function becomes somewhat less generic. It has to know what the pointer points to, the length of that object, and, if the pointer points to the first member of an array, the size of each individual member and how many there are. If the pointer points to another pointer, the problem is compounded. But, except for these restrictions, file input/output in C is a straightforward procedure.

▼ File Input/Output in C++

Does the absense of consequential problems mean that C is a better language for database management than C++? No, it does not, because C's ability to support straightforward record input/output reflects its limited vocabulary for the expression of data records themselves. You can read and write in C what you can express in C.

C++ offers a much richer environment in which to express abstract data types. The breadth of that capability has consequences that influence how you store and retrieve the data values found in objects of the abstract data types.

C++ introduces a new set of problems primarily because of all the additional things that you can put into a class.

The additional features of C++ suggest a solution as well. Inheritance would imply that you can raise the problem of persistence to a higher level of abstraction. Consider the design of a Persistent base class that would automatically manage all aspects of file input and output for any derived abstract data class. If such a base class was possible and one existed, you could derive a class from it in the manner shown in Figure 4.2.

```
class Employee : public Persistent {
    EmployeeNo empNo;
    String *name;
    Department& dept;
    int promotion_ctr;
    Date *promotions;
public:
    virtual int SalaryReviewPeriod();
};
```

Figure 4.2 *A Derived Persistent Class*

The would-be persistent **Employee** class in Figure 4.2 has several data members:

- an instance of another class, known to the application;
- a pointer to an object of a class taken from a class library;
- a reference to an object of one of the application's classes;
- a count of the number of members in an array; and
- a pointer to the first member of an array.

To complicate matters, the class has a virtual function, which means that there is a **vptr** (virtual function table pointer) somewhere among the other data members. Furthermore, you do not know if the embedded objects have **vptrs** themselves.

▼ The Problem with C++ Persistent Objects

How would you design a **Persistent** base class that would know how to find all the pertinent data members and write them to disk? How would the class

know how to construct those data members properly to read them back in? How would it know the size and location of the data members? How could it possibly understand the application-dependent relationship between the **promotion_ctr** member and the **promotions** member? How would it know to get around the one or more **vptrs** in the class?

The answer is, it would not, and you could not design such a **Persistent** base class. The following paragraphs examine each of these problems in detail.

The Size of the Object

Assuming that the object's format is a flat structure of data members with none of the attendant problem members such as pointers, references, and so on, you might consider designing a **Persistent** base class that begins like the one in Figure 4.3.

```
class Persistent  {
    fstream objfile;
public:
    Persistent()
        { objfile.read(this, sizeof(*this)); }
    virtual ~Persistent()
        { objfile.write(this, sizeof(*this)); }
};

class Employee : public Persistent {
    // ...
};
```

Figure 4.3 *A Flawed Persistent Base Class*

In theory, a class that wants to be persistent would derive itself from the **Persistent** base class as shown in Figure 4.3. The base constructor and destructor functions would read and write the object.

Can you see the obvious flaws in this scheme?

The constructor for a base class executes before the constructor for the derived class. Therefore, when the base constructor executes, no **Employee** object exists yet, other than the part that the **Employee** class inherits from the **Persistent** class. None of the derived class's members have been initialized. None of what its constructor contributes to its construction has been done. Assuming that the object would eventually somehow identify itself well enough for the **Persistent** class to find its disk record address, none of that work is done yet.

Conversely, a base destructor does not execute until the derived destructor has completed. The **Persistent** destructor executes after the **Employee** object's destructor has destroyed the **Employee** data members. There might be nothing left to save to disk. The data values may or may not have survived in the object's allocated memory. Everything would depend on what the derived destructor does.

The solution in Figure 4.3 will rarely work if at all. The program must wait until the object is fully constructed before it can read the data members, and it has to write them before they are destroyed. Figure 4.3 is a failed design. The base class in Figure 4.4 is an attempt to correct these flaws.

```
class Persistent  {
    fstream objfile;
public:
    void Read()
        { objfile.read(this, sizeof(*this)); }
    void Write()
        { objfile.write(this, sizeof(*this)); }
};
```

Figure 4.4 *A Better but Still Flawed Persistent Base Class*

The **Persistent** base class in Figure 4.4 adds **Read** and **Write** member functions that can be called after the object is constructed and before it is destroyed. Apparently the derived class and/or its user must participate in the object's persistence. Merely inheriting the behavior of a **Persistent** base class is not enough. This is an important discovery. It will figure prominently in later chapters.

Whether or not the derived class calls **Read** and **Write** from its own constructor and destructor or the using program calls them while the object is in scope is irrelevant. The base class still does not work. Can you see why?

The base class is using the **sizeof** operator to specify the size of the object. The **sizeof** operator is not polymorphic. When you apply **sizeof** to a base class, it returns the size of an object of the base class, regardless of whether or not the object is a derived class. To solve that problem the **Persistent** class levies another responsibility on the derived class—the derived class must supply its own size. Figure 4.5 shows that modification in place.

```
class Persistent  {
    fstream objfile;
    int size;
protected:
    Persistent(int sz)
    {
        size = sz;
        objfile.open("DB",
                ios::in|ios::out|ios::binary);
    }
    void Read()
        { objfile.read((char *)this, size); }
    void Write()
        { objfile.write((char *)this, size); }
    virtual ~Persistent() { objfile.close(); }
};

class Employee : public Persistent  {
    // ...
public:
    Employee() : Persistent(sizeof(*this))
        { Read(); }
    ~Employee()
        { Write(); }
};
```

Figure 4.5 *A Persistent Base Class That Almost Works*

The **Persistent** base class in Figure 4.5 is almost acceptable. The constructor for the **Employee** class provides the size of its own object in the call to the constructor to the **Persistent** class. From that value, the **Persistent** class now knows the size of the object, and it can read and write the correct size. If you can assume that the problems yet to be discussed cannot occur or that they will not be allowed in a derived class, then the base class is capable of reading and writing the data members of objects of the derived class. There is one small bug, however. Can you find it?

This discussion has avoided, for now, how to position the file's pointers to the object's location. It also defers any discussion about how to handle objects whose sizes change while they are in scope and how to delete objects. These are genuine considerations, but their solutions involve no major obstacles but simply the selection of an appropriate approach. The book will address these matters after the more serious problems are identified and resolved.

Nonpersistent Members

The issue of object size is resolved, but the other problems are still ahead. Figure 4.5 gives a hint about the next problem. By blindly reading and writing the entire size of the object beginning at its object address, the program reads over the **objfile** data member that defines the **fstream** object. This bug will surely cause the input/output operations to fail. You could define the **objfile** object as **static** and get it out of the object's data space. That would solve the problem if all **Persistent** objects used the same file, but the bug is an indication of ones that your derived classes could have as well. It is possible and probable that a derived class will have objects that should not be read and written. Some data variables deal with the run-time context. There could be state variables such as input/output handles or user interface window handles that depend on run-time conditions and that must not or should not persist beyond a specific instance of the data type. The **Persistent** base class cannot know about these data members in the derived class and, therefore, cannot, without cooperation from the derived class, do anything to get around them.

Pointers in the Object

Refer back to Figure 4.2. The hypothetical derived class has pointers. A **Persistent** base class cannot know anything about those pointers. It has no way of knowing the types of whatever the pointers point to. Even though you, the programmer, can look straight at that class definition and see clearly what the pointers are, the base class cannot, particularly at run time. C++ is simply not that kind of object-oriented language. Such languages might exist, but C++ is not one of them.

Clearly the base class cannot simply write and read the pointers to and from the database. The value from one instance of the object is not likely to persist for a subsequent instance. The pointer would point into unknown territory. The object that it points to might not even have been constructed for the subsequent instance. Besides, the base class cannot tell that the data member is a pointer because the base class cannot know anything at all about the contents of the derived class. Even if the base class had some kind of list of data members, C++ has no type identification. There is no expression such as "ifPointer(item)" in C++.

Clearly, something other than the **Persistent** class has to decide which data members to read and write and which ones to skip.

References in the Object

The class in Figure 4.2 has a reference to another object. C++ compilers implement references as pointers that act like aliases and then manage the pointer de-referencing for the programmer. When you see a reference to an object within another object, you can assume that the data members for the referenced object are somewhere else. Chances are the referenced object has not been constructed at the time the referencing object gets read back in. The problems are similar to those of the pointer but with one more wrinkle—a reference in an object *must* be initialized by the object's constructor declarator block. To instantiate an object that has a reference in it, you must already have the referenced object in scope. A **Persistent** base class cannot know how to do that. Only the derived class or the application that uses it can know where the referenced object is.

Format-Defining Related Objects

You can probably assume that the **promotions** pointer in the class definition in Figure 4.2 points to an array of objects and that the **promotions_ctr** integer is a count of the number of objects in that array. You can make that assumption because the members' names and their proximity to one another suggest that to be the case. A **Persistent** class could not infer such relationships among the members in the derived class and could not, therefore, do anything appropriate with them.

Embedded Objects

An object that is embedded in the would-be persistent object repeats all of the same problems that the persistent object has except that they aren't as obvious at first glance. You have to go to the class design for the embedded object. If you designed the embedded object for your application, then you have some control over how much and how well it will participate in its own persistence. Nonetheless, you still have to address all the same problems of persistence that the original class has.

Don't forget that embedded objects can themselves contain embedded objects.

Objects from Class Libraries

If the persistent class embeds, references, or points to objects from class libraries, the format and design of those classes are probably beyond your control. You can do little to recruit those classes for participation in their own persistence, particularly if you have no source code for the class members. Even if you have the code, the classes might be under rigid project management, which forbids their modification.

It would be a shame to have to adopt a persistent object strategy that does not permit you to use other class libraries in your design. A reason for choosing C++ in the first place is the vast resource of class libraries that support specific problem domains. You should not have to forsake such a resource just to include persistence.

Virtual Functions

The class in Figure 4.2 includes a **virtual** function. When a class has virtual functions, the compiler builds a table of pointers to the functions to accompany the class. Usually only one table exists in memory for the class, and each object of the class has a pointer to the table. These conventions might vary between compilers, but the mechanics, and therefore the attendant problems, will be similar to those described here. The pointer in the object is called the **vptr** and the table is called the **vtbl**. When the class has derived classes, and when those derived classes override the virtual functions, the compiler puts the overriding function addresses in the **vtbl**. The C++ language manages polymorphism by calling virtual functions this way.

The presence of a hidden **vptr** presents another problem for the **Persistent** base class and is another reason why the base class shown in Figure 4.5 cannot work some of the time. If the program writes the full content of the object to a disk file, the image will include the **vptr**. If another program instantiates the object and the base class reads its image in from the database, the image will write over the new program's **vptr** for the object with an old, incorrect value. When the program tries to execute one of the virtual functions, the system will probably crash.

▼ Copies of Objects

A persistent object can exist in two places—in the database and in the memory of the computer when a program instantiates it. While the object is in memory, the memory copy represents the object. Any changes that the program makes to the object reflect the current state of that object, and the disk copy is—for the moment at least—out of date and unstable. When the object goes out of scope, the persistent database manager returns the memory copy to the disk, and everything returns to a stable condition.

What happens when the program makes two copies of the object? There are several ways that this can happen: the program can instantiate more than one copy and have both of them in scope at the same time; the program can pass an object by value to another function; the program can assign the object to another object of the same class.

To C++, these multiple copies are different objects; but to the application, they are the same. Employee 123 is Terry Myers, regardless of how many copies of Terry's employee record are in the computer. Now suppose that the program changes the two copies in different ways. Whichever copy of the object goes out of scope last is the one that will prevail, and that is not necessarily the desired effect.

The situation just described represents a less-than-perfect program design, but sometimes it can happen to the unwary, particularly given the way that C++ can construct anonymous, hidden, temporary objects. Should the persistent database manager worry about this problem or should there simply be a rule stating that you will not write code that duplicates objects?

▼ Summary

This chapter discussed the problems associated with storing and retrieving objects in a persistent object database. The process is not as easy as it might appear at first glance, and this chapter explained why. Chapter 5 discusses some alternative solutions to those problems and identifies the objectives of a persistent object database.

▼ References

The Annotated C++ Reference Manual, Ellis and Stroustrup, 1990, Reading, MA: Addison-Wesley.

Chapter 5

Solving the Persistent Object

"Persistence is the property of an object through which its existence transcends time and/or space."

Grady Booch

Chapter 4 posed several problems related to supporting peristent objects in a higher-level C++ base class that assumes most of the burden for persistence. This chapter looks at alternative solutions to those problems and identifies the objectives of a proposed solution.

▼ Alternative Solutions

How can you solve the problems posed in Chapter 4? There are a number of ways to approach the solution. Some of them solve parts of the problem, and some solve other parts. Some solutions are less than no solution at all. Among the alternatives are:

- Use custom file input/output methods;
- Extend the C++ language;
- Write a C++ preprocessor;
- Limit the scope of a persistent class;
- Use a relational database management system; and
- Let the persistent class participate in its own persistence.

The first and most frequently used solution, described in the next section, is one of the least desirable.

Custom File Input/Output Methods

This first alternative is nothing more than what many C++ programmers have been using all along. They forego the idea of a **Persistent** base class because of the difficulties. Instead, they write custom file storage and retrieval methods for the persistent classes in their design. This approach, although widely used, betrays everything programmers have learned about database management in the last thirty years.

A long time ago, programmers discovered the inherent similarities in the requirements for database management across applications. From that discovery grew disciplined data models and database management software systems that implemented them. These were important discoveries and developments. Many large-scale software systems exist today only because database management tools exist to make them possible. It is unthinkable that the software industry would not have hit upon what is now obvious—that the processes of organizing, storing, and retrieving data records should be managed by reusable, general-purpose systems software.

The apparent persistence-contrary nature of the C++ class definition described in Chapter 4 is the result of its power. You can design a class with

all the things that are difficult for a database manager to pin down. You want those things in your classes even though they get in the way of an easy persistent strategy. But that doesn't mean you should abandon the effort—only that it is going to take some thought and work.

Extend the C++ Language

The second alternative is to change the syntax of C++ to include persistence. You can't do that on your own, though. C++ is a formal language undergoing formal definition by the ANSI X3J16 C++ standardization committee. You might propose to X3J16 that they add the persistent attribute to the language and that they figure out how to make it work. Perhaps you could develop the syntax and implementation details. Assuming that you propose a workable design, it is doubtful that you could persuade them to adopt it. The committee has charted a course that standardizes the language in ways that conform with the *Annotated Reference Manual* (*see* References), and too many influential experts rightly assert that persistence is not a language issue. The nature of persistence tends to draw the solution closer to the target operating environment. C++ is a portable language, as portable as C if not more so. One of the features of both languages is that their compilers are written in the languages themselves. If you add input/output primitives to the language, the portability of the compiler and the language is compromised. That is one reason why C and C++ have no input/output operators.

There are language features that the X3J16 committee can add to C++, however, that will enhance an object database management system. One such feature under consideration now is type identification. Eventually that feature will be in the language, and software such as the PARODY system in this book will be able to use it. Beyond minor extensions such as type identification, though, do not expect much in the way of language change to support persistent objects.

Write a Preprocessor

You can extend C++ without getting involved with X3J16 by writing a preprocessor program that translates an extended C++ language into correct C++ code. This approach is consistent with the tradition of C++; the original C++ compilers translated C++ code into C. Depending on the extent of your

translator's support for persistence, the preprocessor might need to search all the header files included in your program down to the deepest nested level. This search would ferret out the formats of the embedded classes to see how to save and retrieve their data members. The preprocessor approach does not solve problems such as the relationship between the array element counter and the array pointer discussed in Chapter 4; but it is a step closer, and some object-oriented database management systems use preprocessors.

Limit the Scope of a Persistent Class

You can probably get around all the problems mentioned in Chapter 4 by laying down some ground rules for users of your persistent object database system. You can specify that to derive from the **Persistent** class, an object may not use any of the C++ features that cause those problems. If your application can get by without the features, then such an approach might work. However, if limiting a persistent object class to what is easy to implement suits you, then you might as well use one of the relational DBMSs that are already available.

Use a Relational DBMS

You can build an object database manager by putting a C++ wrapper around an existing C library that implements a relational DBMS. That is a viable approach, and there are times when it is the appropriate one. Chapter 6 discusses some of the considerations for making this decision.

The Persistent Class Participates in its Own Persistence

This alternative is the one used by the software in this book. Every problem identified in Chapter 4 points to one undeniable conclusion: the intelligence that identifies what can and should be persistent in a class is contained in the class itself. The class must cooperate with the persistent object database management system. That conclusion is the basis for the PARODY database management software described later in this book. The rest of this chapter is about the objectives for PARODY, the Persistent, Almost Relational Object Database manager where a class cooperates in its own persistence.

▼ Objectives of a Persistent Object Database

The first objective of a persistent object database manager is the identification of a method to describe the using class's cooperation in a way that is easy for programmers to use. The persistent object database must consist of a simple, intuitive, and consistent class library interface. There should be a minimum number of class member functions in the interface, and the interface should use the features of C++ in an intuitive manner. This objective permeates the entire PARODY project. Following are the specific objectives for how the persistent object database system must work.

Associative Access

When a program creates an object, its identity is (if nothing else) its location in the program's memory, its address. Certainly the program knows where it is. When the program is done with the object, the database manager must store the object in a database. At that time, the creating program has no need to identify the object further. Subsequently, however, when the program or another program wants that same object again, the program needs to provide some identifying information for the database manager to use to retrieve the object. The address is without meaning then. There must be something about the object that is known to the program and that the database manager can use to find it in the database. In the traditional database, one of the data members is used to associate the object with its position in the database. Objects must therefore be identified by key data member values. Each class must therefore have a primary key data member.

When the persistent object database saves an object, it maintains an index that associates key values with their object records in the database. When a subsequent program declares an object of the class and specifies the same key value, the persistent object database can retrieve the correct associated object. The database must also maintain secondary indexes so that the program can retrieve specific objects on the basis of data values other than the primary key.

Associative access is preferred over navigational access, which assigns object identifications—usually related to the object's logical address in the database. An application that uses a navigational database must remember these addresses in order to retrieve the objects at a later time. Objects that are

related to other objects of other classes would similarly use addresses to point to their relatives. Some existing systems work this way. Navigational access is one of the characteristics that the relational database model strives to eliminate because of the inherent instability in its use. When the architecture of a database uses interfile pointers to maintain file relationships, the integrity of the entire database is dependent on the correctness of those pointers. The relational model uses data element values to form the relationships. Pointer chains cannot become broken because no such chains exist. Relating object locations and relationships by associating data values is what associative access achieves and is the foundation of the relational data model. The approach described by this book uses what it can from the relational data model in the implementation of the object database.

Maintenance of Objects

The persistent object database will provide methods for the application to change and delete objects that exist in the database. The database is not merely a repository or cache of stray objects. The application needs to create objects and specify that they are to be persistent. The application needs to retrieve specific persistent objects that were created earlier, change their data values, and return the updated objects to the database. The application needs to delete specific objects from the database. These capabilities are the usual add-change-delete maintenance operations that you would expect in any database management system.

Object Integrity

Object integrity means that the database does not store ambiguous objects and that it does not seem to retrieve objects that have not previously been stored. The persistent object database will indicate when a declared object does not already exist in the database, and it will refuse to add an object to the database if another object exists with the same key value.

If the application declares an object and specifies a primary key value that the database does not represent in an object record, the database manager will indicate as much. The object exists as far as the compiler's run-time process is concerned. There is a declared instantiated object of the class. It is a null object, however, containing no data values. The application can ask the object if it exists as a retrieved object, and the object will return a negative response.

If the application attempts to add an object to the database when one already exists with the same primary key value, the database manager will refuse to add the object and will return a negative response to the add request.

Maintaining object integrity keeps the database in a stable condition with respect to the requirements of the application. There should not be two department records with the same department identification. A program should not think it has retrieved an empty employee record when no employee record exists with the specified employee identification.

Class Relationships

Classes are related to one another in ways that the application manages. The relationship is important to the application. Objects of the employee class are related to the objects of the department class where the employees work. In a strictly object-oriented program, classes can be related by references or pointers in one class to other classes. This method works well as long as the objects are memory-bound. The references and pointers refer to memory addresses. When the objects are in the database, such addresses are not useable. When the objects are retrieved by other programs, the addresses are meaningless. Some object-oriented database managers assign logical addresses to the persistent objects and an index of those addresses to the physical disk addresses. For those addresses to have meaning to an application, the application must relate them to some data values that identify the object.

The persistent object database proposed in this book will maintain relationships between classes on the basis of the key data values themselves rather than on the basis of pointers, and it will maintain the integrity of those relationships. If a class includes another class's primary key as a secondary key, the classes have an implied relationship. The persistent object database will not delete an object if other objects that are related to it remain, and it will not attempt to relate an object to one that does not exist.

If the application retrieves an employee object and changes its department identification—perhaps because of a user's input—to an invalid department, the database manager must not allow the object to be returned to the database. What is more, the database manager must notify the application that the replacement was not allowed. Without these controls, a database would have the potential to become chaotic with records implying relationships that could not exist. The database would have lost track of the employee's department

assignment. A subsequent application program might depend on the presence of department information for every employee, and that program would fail when it processed the errant object.

If the application deleted a department object while there are employee objects related to it, the same kind of integrity violation would occur. Therefore, the object database manager should not allow an application to delete an object if other objects are related to it. You can't shut down (delete) the department while employees are still working there. You must transfer (change) or terminate (delete) the employees so that none of them are assigned to the department that you want to delete.

Copies of Objects

If a program declares multiple memory copies of an object, whether through assignment, copy construction, or multiple instances, the database manager should know about it. There are several ways to deal with that situation. One solution marks all subsequent copies as being read-only. The application cannot change any but the original copy. Another solution simply ignores the problem and treats it as a policy matter. Programs should not make copies of objects in this approach. A third solution, which the software in this book uses, is to apply a C++ idiom called "reference counting" where all such copies are actually references to the original. Chapter 8 describes how that works.

▼ Summary

This chapter discusses several alternative solutions to the problems presented in Chapter 4. Then it chooses one of them as the basis for the PARODY system and develops the objectives for that system. Chapter 6 compares the object database with the relational data model, shows how the object model benefits from the lessons of relational technology, and discusses when you should choose each one.

▼ References

Object Oriented Design with Applications, Grady Booch, 1991, The Benjamin/Cummings Publishing Company, Inc.

The Annotated C++ Reference Manual, Ellis and Stroustrup, 1990, Reading, MA: Addison-Wesley.

Chapter 6

The Object Database and the Relational Database

"The compass and square produce perfect circles and squares. By the sages, the human relations are perfectly exhibited."

Mencius

The object database is a relatively new concept. There are not many implementations now, although you can expect to see more and more as the popularity of C++ and object-oriented programming continues to grow in the community of personal computers.

Some existing implementations have tried to emulate with disk images the in-memory nature of object relationships in a program. Others have simply put C++ wrapper classes around existing relational database management systems. Still others use the **Extern "C"** linkage specification of C++ to link programs with C database management function libraries. There is no consensus about what constitutes a genuine "object-oriented database."

The relational database is, on the other hand, a time-worn and proven concept. Relational technology evolved in the 1970s to solve the problems associated with traditional hierarchical and network databases, which were fragile due to the architecture of hidden embedded file pointers and which were nonintuitive in the view of users. The relational data model eliminated the hidden pointers by forming file relationships with data values, and it improved the user's perception of the data representations by making the database schema visible to the user and functionally relevant to the application.

There are restrictions in the relational data model that would discourage its use in an object-oriented design. Conversely, it has benefits that would encourage its use. This chapter compares the requirements for an object database to the advantages of the relational data model with two purposes: (1) to identify the items to consider when deciding whether to use the object or relational models; and (2) to identify and justify those features of the relational model to adopt for the PARODY object model.

▼ The Object Database Model

If the object model—the basis for object-oriented design—applies the principles of abstraction, encapsulation, inheritance, and polymorphism, then the object database model must extend the object model into the realm of database management. Objects, therefore, may be stored in and retrieved from an object storage medium that survives after the object's creator expires. An object stored in such an object database is a *persistent object.*

Each persistent object possesses a unique identity by which to index it in the object database. Other than for the *key* identifying data member, the object database model imposes no requirements or restrictions on the content or format of the persistent object. Therefore, if you can define and instantiate an object of an abstract data type, and if that object can be uniquely identified with a data member, you can store that object in an object database and retrieve it at a later time.

The key data member does not need to be associated with the object's data at all. An object database can use arbitrarily assigned object addresses as object identification. That is the easy way out for the object database management system because it offloads onto the application the responsibility for indexing the object's identification with its functional purpose.

There is no formal definition as to how the object database model is to be implemented. Its internal format and architecture and its interface with a using program are undefined. The PARODY object database implementation in this book does not comply with any standard, *de facto* or otherwise, because no such standard exists.

To understand why this is so, you must consider the relationships between traditional programming paradigms and database models. There is no *relational* programming paradigm. There is no *structured* database model. Programmers adopted structured programming because it was superior to the ways that they had been expressing procedural algorithms. Database designers adopted the relational database model because it was superior to the ways that they had been designing databases. The two came together only coincidentally. A structured program could always use a nonrelational data model. A nonstructured program was in no way prevented from using the relational data model. But because procedural programming does not encapsulate the design of data representation and its behavior, the two then-superior approaches to software design were never necessarily bound.

The object model is the first attempt to marry a programming model with a database model. Why? Object-oriented design expresses abstract data types in ways that no traditional data model supports well. Object-oriented design needs its own database model.

Inasmuch as there are no restrictions on the format and content of a persistent object, the object data model is able to represent any kind of data. There are applications that use streams of data that a traditional database management system is incapable of storing. Usually, these data streams represent long, variable-length, formless bit packages sometimes in arrays of variable dimensions.

CAD/CAM systems deal with such objects. They store digital representations of engineering designs that consist of wire-frame and solid models. The models record the properties of their composition to facilitate finite-element analysis of their reactions to stress, temperature, vibration, and motion.

Imagery systems store raster scanned images. Cartography and meteorological systems store points, vectors, and references to cataloged features. Multimedia systems store bit streams of audio and video recordings. Text processing systems store streams of text with embedded formatting control fields.

Objects such as the ones just described appear formless to the casual observer, and so they are not particularly suitable for a database model that imposes rigid form. You would have to reform the objects to make them fit the database. That is like modifying the house because of the limitations of your hammer and saw—or of your carpenter.

▼ The Relational Database Model

The relational model is the result of the work of E. F. Codd in the 1960s. He described a data model where entities of data are represented by *relations*, which resemble tables of *rows* and *columns*. Most programmers view these relations as files in a database. The rows are the individual file records, and the columns are the fields within the records. Codd does not define the model to that level, however, leaving the details of how to form the relations to the implementor. Codd calls the rows by another name—*tuples*.

Figure 6.1 shows a typical, although simple, relational database with two relations.

DEPARTMENTS

DEPT_NO	DEPT_NAME
00020	SALES
00030	ACCOUNTING
00040	ENGINEERING
00050	MANUFACTURING

EMPLOYEES

EMPL_NO	EMPL_NAME	DATE_HIRED	SALARY	DEPT_NO
00001	JONES	05/06/84	22000	00030
00002	SMITH	02/04/83	22000	00020
00003	BROWN	09/21/82	21000	00030
00004	GREEN	07/12/80	18000	00040
00005	WHITE	11/11/85	25000	00020

Figure 6.1 *A Relational Database*

Any discussion of the relational model is incomplete without consideration of Codd's twelve rules. Without enumerating them here or presenting them in their original sequence, this discussion addresses the essence of the rules to the extent necessary to compare the features with the object model.

As shown in Figure 6.1, all the rows in a relation have the same columns. The data value for a particular row/column may be *null*, which means that there has been no value assigned.

There are no repeating groups—arrays—in a relation. There are no variable-length columns.

Each relation has a column or aggregate of columns that is its primary key data element. Each row must have a unique value in that key to distinguish the row from the other rows in the relation. The EMPL_NO column is the key data element for the EMPLOYEES relation in Figure 6.1. The DEPT_NO column is the key data element for the DEPARTMENTS relation. A program may not add an EMPLOYEE relation with an EMPL_NO column value of 00002, for example, because one already exists.

The application program may retrieve any element of data in the database by specifying the relation, column name, and the value of the relation's key column, which might not be the same as the column being retrieved. Many implementations simply allow the application to retrieve the full record by providing a buffer that will hold it and the key value.

The relational database describes its own schema in a file called the *catalog*. This catalog allows general-purpose query and report writer languages to interact with the user and the database.

The user can create *views* of the database from queries. The views look like relations but are the result of specifying a different configuration of columns with the PROJECT operator, combining the contents of multiple relations with the JOIN operator and using a subset of the possible rows with the SELECT operator. These views are not, theoretically, new relations. If a data value in the database changes, any views that the data element participates in will reflect the change. The user can change a view only when the change would not damage the integrity of other views or of the database itself.

The database manager enforces the integrity of the relationships that exists between relations as specified by rules in the catalog. With the database shown in Figure 6.1, a program cannot delete the DEPARTMENT row that has the 00020 DEPT_NO column value because employees are still assigned.

Similarly, a program cannot assign an employee to a department that does not exist.

Database administration will not affect the users' operations. If the administrator reorganizes the physical or logical organization of the database, the users are unaffected.

▼ Choosing a Database Model

When should you use a relational database, and when should you use an object database? The answer will depend on several considerations. Is your problem domain already supported by a mature database and database management system? If so, you might not want to risk the change. Pioneers, while heroes in history, did not reap the rewards of their explorations. Those who followed, once the way was made safe, found profit and success. The same is true in software design. If your design can leverage earlier successful efforts, it stands a better chance for success of its own.

If you are not building on earlier work, then the answer lies in an analysis of the mutually exclusive strengths and weaknesses of the two data models and the requirements of your application. The relational data model has advantages that the object data model cannot support. Among these are:

- The relational schema is stored in the database catalog.
- General-purpose query programs can use the catalog.
- The SELECT, PROJECT, and JOIN operators can build new database views at run time.
- The database is cross-compatible with other applications that use the same DBMS.

Conversely, the object data model supports data representations that the rules for the relational data model prohibit.

- Variable length data members can support such applications as imagery, multimedia, geographic data, weather.
- Abstract data types support user-defined data abstractions.
- Arrays permit repeating groups.
- Encapsulation of data formats with methods bind data representation and behavior.
- Polymorphism customizes the behavior of derived data types.

Clearly the strength of the object data model is that it supports an object-oriented design. A designer must decide which way to go by weighing the benefits of both approaches. If you decide that you need those relational features not available in the object model, then you will use a relational database manager. You will find, however, that the object model can emulate some of the behavior of the relational model, and if the parts it leaves out are expendable, then the object model is a better choice for the C++ program.

▼ Adapting the Relational Model to Objects

Having chosen the object model, our next step is to decide how to implement it. Chapter 5 identified the objectives for the PARODY system of object database management. There are advantages to the relational model that the object model can adopt in support of those objectives, the most prominent one being the ability to support associative access, and a secondary one being the maintenance of relational integrity among objects. Later chapters will describe the PARODY interface in detail. The discussion that follows will define how PARODY will emulate some of the features of a relational database.

The Database Schema

The relational model includes a data definition language (DDL) in its catalog that defines the format of records in the database files and the relationships of files to one another. The DDL is said to define the database schema. The PARADY DDL is C++ itself. You design a database by designing classes that define the file formats and their key data members. Figure 6.2 is an illustration of such a design.

The **Employee** class in Figure 6.2 is derived from the **Persistent** class, which is how it acquires the persistent attribute. You will learn how the **Persistent** class works in later chapters. You will learn, too, that classes derived from the **Persistent** class will do specific tasks in their constructors and destructors and will provide a few additional members to support their persistence.

Two of the classes in Figure 6.2 are derived from the **Key** class. This is how you specify a key data member. Later chapters will explain how derived key classes provide overloaded operator functions to support the PARODY indexing mechanism.

```
// ---- key department number
class DeptNo : public Key    {
      int deptno;
      // ...
};

// ---- department class
class Department : public Persistent    {
      DeptNo deptno;          // primary key
      String *name;
      // ...
};

// ---- key employee number
class EmployeeNo : public Key{
      int empno;
      // ...
};

// ---- Employee class
class Employee : public Persistent {
      EmployeeNo empno;       // primary key
      DeptNo deptno;          // secondary key
      String *name;
      Date date_hired;
      Currency salary;
public:
      // ...
};
```

Figure 6.2 *A Database Schema*

Observe that the **Employee** class has two key members: the **EmployeeNo** object and the **DeptNo** object. The first derived **Key** object is the class's primary key. All subsequent **Key** objects are secondary keys. There can be only one object of a particular class in the object database with a given primary key value because the primary key value is what identifies the object. Multiple objects of the same class can share secondary key values, however. The **Department** class in Figure 6.2 has a **DeptNo** member as its primary key, so there may be only one **Department** object with a department number of 123, for example. The **Employee** class has a **DeptNo** member as a secondary key. There can be several employees assigned to department number 123. This relationship is

implied based on the presence of those **Key** data members. The design implies the relationship, so the persistent object database maintains it. You will not be allowed to write an **Employee** object with a non-null **DeptNo** key value unless there is a corresponding **Department** object with the same key value. You will not be allowed to delete a **Department** object if any **Employee** objects include the matching **DeptNo** key value.

Retrieving Objects

A program retrieves a persistent object by declaring an instance of it with its primary key member value as an initializer. The program then asks the instantiated object if it does, in fact, exist in the PARODY database. Figure 6.3 is an example of that.

```
void f()
{
    Employee empl(123);
    if (empl.ObjectExists())   {
        // --- the object exists
        // ...
    }
}
```

Figure 6.3 *Retrieving a Persistent Object*

Adding Objects

A program adds an object to the PARODY database by declaring one that does not exist, providing whatever data member values it needs, and telling the object to write itself to the database. Figure 6.4 is an example of how that works.

```
void f()
{
    Employee empl(123);
    if (!empl.ObjectExists())   {
        // --- the object does not exist
        // ...
        empl.AddObject();
    }
}
```

Figure 6.4 *Adding a Persistent Object*

Changing an Object

A program changes an object on the PARODY database by declaring one that exists, changing the object's data members, and telling the object to rewrite itself to the database. Figure 6.5 is an example of how that works.

```
void f()
{
    Employee empl(123);
    if (empl.ObjectExists())   {
        // --- the object exists
        // ...
        empl.ChangeObject();
    }
}
```

Figure 6.5 *Changing a Persistent Object*

Deleting Objects

A program deletes an object from the PARODY database by declaring one that exists and telling the object to delete itself from the database. Figure 6.6 is an example of how that works.

```
void f()
{
    Employee empl(123);
    if (empl.ObjectExists())   {
        // --- the object exists
        // ...
        empl.DeleteObject();
    }
}
```

Figure 6.6 *Deleting a Persistent Object*

The objects in Figures 6.4 through 6.6 may continue to process after receiving the **AddObject**, **ChangeObject**, or **DeleteObject** messages. The messages tell them to prepare to be added, changed, or deleted. The addition, change, and deletion occurs from the object's destructor when the object goes out of scope.

Maintaining Object Integrity

A program may not always add, change, or delete an object. It may not add an object that already exists. It may not add an object or change one such that the new or changed object is related to another object that does not exist. It may not delete an object if other objects are related to it. The program needs to test for these conditions. The errors might result from user input errors, for example, and the program would need to issue an error message to the user. The **AddObject**, **ChangeObject**, and **DeleteObject** functions return a true value if they will permit the action and a false value if they will not. Figure 6.7 shows how those functions work.

```
if (!empl.AddObject())
    // --- add was rejected

if (!empl.ChangeObject())
    // --- change was rejected

if (!empl.DeleteObject())
    // --- delete was rejected
```

Figure 6.7 *Maintaining Object Integrity*

Navigating a Class

As you saw in Figure 6.3, the program locates an object by constructing it from the value of its primary key. A program can also find an object in the persistent database by specifying a value for one of its secondary keys. First it instantiates an empty object, which the database will not search for. Then it initializes the secondary key value and uses the **FindObject** function to find the first object of that class in the database that has the secondary key value. Figure 6.8, which assumes that the **Employee** class has **SetDeptNo** and **DeptKey** member functions, shows how that works.

```
Employee empl;
empl.SetDeptNo(deptno); // initialize the dept#
empl.FindObject((Key *)&empl.DeptKey());
if (empl.ObjectExists())
    // --- at least one object exists
```

Figure 6.8 *Finding an Object with a Secondary Key*

The program can retrieve the first or last persistent object of a class in the sequence of the key value by using the **FirstObject** or **LastObject** functions as shown in Figure 6.9.

```
Employee empl;
empl.FirstObject((Key *)&empl.DeptKey());
if (empl.ObjectExists())    {
    // --- the object exists
}
// ....
empl.LastObject((Key *)&empl.DeptKey());
if (empl.ObjectExists())    {
    // --- the object exists
}
```

Figure 6.9 *Finding the First and Last Object of a Class*

Following a **FindObject**, **FirstObject**, or **LastObject**, a program can navigate the persistent objects in a class in the sequence of the keys by using the **NextObject** or **PreviousObject** functions as shown in Figure 6.10.

```
// ----- navigate forward
Employee empl;
empl.FirstObject((Key *)&empl.DeptKey());
while (empl.ObjectExists())    {
    // --- process the object
    empl.NextObject((Key *)&empl.DeptKey());
}
//----- navigate backward
empl.LastObject((Key *)&empl.DeptKey());
while (empl.ObjectExists())    {
    // --- process the object
    empl.PreviousObject((Key *)&empl.DeptKey());
}
```

Figure 6.10 *Navigating a Class*

Navigating Among Classes

To navigate class relationships, the program declares an object of a class that has as a secondary key the primary key class of another persistent class. It uses the secondary key value to declare an object of the other class. Then it processes that object and following objects as shown in Figure 6.11, which assumes that the **Department** class has **DeptNo** and **DeptKey** member functions.

```
Department dept(321);
if (dept.ObjectExists())  {
    Employee empl;
    empl.SetDeptNo(dept.DeptNo());
    empl.FindObject((Key *)&empl.DeptKey());
    while (empl.ObjectExists() &&
            empl.DeptNo() == dept.DeptNo())   {
        // ...
        empl.NextObject((Key *)&empl.DeptKey());
    }
}
```

Figure 6.11 *Navigating Among Classes*

▼ Summary

This chapter compares the relational and object data models and offers ways to decide which you should use. Then it identifies the features of the relational model that the PARODY system will incorporate into the object model. Chapter 7 digresses to discuss the design process for an application's object database.

▼ References

Database Modeling in the PC Environment, Bradley D. Kliewer, 1992, New York: Bantam Books.

An Introduction to Database Systems, Second Edition, C. J. Date, 1977, Reading, MA: Addison-Wesley.

Computer Data-Base Organization Second Edition, James Martin, 1977, Englewood Cliffs, NJ: Prentice-Hall.

Chapter 7

Designing an Object Database

"Nothing is particularly hard if you divide it into small jobs."
Henry Ford

Object-oriented design is a big topic, one that a single chapter in a book cannot do justice. Designing a full object-oriented software system—particularly a large, complex one—is a major undertaking that should include the counsel and participation of experienced designers. One facet of an object-oriented design is its database. Most texts about object-oriented design concentrate on in-memory object hierarchies rather than databases, teaching you to visualize classes and build accurate and meaningful class relationships. If they get into database design at all, they usually retreat to traditional relational methods, which is understandable; few object-oriented programming languages support a persistent object database. There are no systems to provide examples for a book.

This chapter explores the basics of object database design. You will learn that any programmer can design a workable database by using intuition and common sense and without applying or even understanding the mysteries of database administration. This book contends that if you are smart enough to write a computer program, you are smart enough to design a database. I took the same position in C Database Development, asserting that database design is inherently easy requiring only a few intuitive procedures to ensure a sound design. Thousands of programmers used the book and proved it. But that position could have been a potential target for harsh criticism by the practiced experts of relational technology. It flies in the face of tradition and threatens the sanctity of their priesthood. Taking the same position with an object database is safer. Few experts exist who can criticize it; the technology is too new.

Readers who used the earlier book will find few differences in this approach because the procedures for identifying and decomposing traditional and object database requirements are similar.

This chapter avoids the arcane disciplines usually advanced as proper database design techniques. You won't need special forms, templates, rigid rules, or disciplined procedures for design review and refinement. This chapter does not promote a formal "methodology." Those tools and procedures have their place; they are for the big development project with a big system, a big staff, a big problem to solve, and structured management watching every move.

▼ General Design Guidelines

Develop 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. Be willing to modify the solution as it evolves.

Design an object database in small increments. Learn to isolate the functionally independent parts. Build individual components that have single points of interface. Deal with each component as if it was its own database. Then integrate them as loosely as you can so that modifications to one component will not interfere with the operations of the other. If you build an **Employee** class and a **Department** class, build them separately. Other than for the appearance of a department number in the employee's class and the maintenance of that relationship, the two classes should be independent.

When an object of one class needs data from an object of the other, or when one needs to update data in the other, add public member functions to the class interfaces.

▼ The Ten Steps of Object Database Design

The design of an object database involves ten steps taken, for the most part, in succession:

1. Identify the basis for the database requirements;
2. Define the database's functional and performance requirements;
3. Identify the data items;
4. Separate the data members from the persistent classes;
5. Build a data member dictionary;
6. Gather data members into persistent classes;
7. Identify the key data members of each class;
8. Identify the relationships between classes;
9. Identify the class methods; and
10. Add the inheritance and member functions to make the classes persistent.

There is always an eleventh step, which is to reiterate the first ten. 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, stay loose, and change your results. Don't be afraid to throw out old work and start over. This eleventh step, called *refinement* is often left out.

▼ Identify the Basis

If you are about to design a database, you must have a mission and a purpose. Someone has asked you to automate something. Who made that request and

how? How much did they know about the problem when they asked for a solution? Is the problem 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 an understanding of 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, what resources 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 multivolume report. Consider a personnel database. 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 dates they were hired. Record the status of projects assigned to departments. Record the employees who are working on projects. Distinguish managers from project workers. Distinguish department managers from project managers.

The list of available resources includes such information as existing source documents (employee time cards, for example), interfaces to other databases, people who understand the function, 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, you need to analyze 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.

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 database. 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. An existing system is a good basis for a database design.

If the user works with a manual system, start there. Collect the functions that the system supports and turn them into functional and performance requirements. A manual system probably contains enough information to guide you in automating it. As such, it is a good basis for your design. Usually, the purpose of automation is to improve the system's performance, functionality, or both, in which case you will extend the basis to include the improvements.

Starting from Scratch

It is hard to imagine designing a database from scratch with no prior procedure to use for a model. Almost everything has either been automated or set up as a manual procedure. If a user asks you to initiate a complete new procedure and solve a brand new problem, you should look for an existing solution to a similar problem. Chances are someone has already developed one. If you do not find one, you must start from scratch. The best approach is to develop 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 your design and implementation will no doubt change several times 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. Make it clear enough that anyone who understands the application can understand the basis. Keep the written basis handy while you design and develop the system, and make updates to it as design changes occur. The basis is your foundation for the requirements analysis.

▼ Define the Requirements

This step enumerates the requirements for the database, addressing functional and performance requirements separately. 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. Word it so that users and programmers alike can understand it. The requirements list is the planning document for the database. Anyone who has it and understands it should know what to expect from the system. Without it, you will never know when you are done.

Functional Requirements

Functional requirements specify the kind of data the database will contain as well as the system's output. Document everything you know about the functions that are supported. Write about the problem not the solution; say what, not how. Be specific in identifying pieces of information the database must know about. Here is an example of a list of requirements for the personnel database.

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 orgranization's projects by project number. Each project's record will include its name and schedule, represented by the start and completion dates and the number of labor hours budgeted to and expended against the project.
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.
8. The system will record the assignment of an employee as department manager.
9. The system will record the assignment of an employee as manager of a specific project.

This list is a beginning. You would also develop requirements for specific queries and reports, perhaps with accompanying screen or report layouts. Identify cyclic processes such as transaction posting cycles. Identify archiving requirements. Such requirements can influence a database design.

Performance Requirements

Performance requirements specify frequencies, speeds, quantities, and sizes—how often, how fast, how many, and how big—the database must support. Some of the personnel system's performance requirements might look like this list.

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 unlimited employee-to-project assignments—all the employees can be assigned to all the projects.
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.

This list is also only a beginning. The completed list would include performance criteria about backups, restart/recovery, scheduled maintenance down

time, and database administration. In a multiuser environment, it would specify access privileges and record and file locking requirements.

▼ Identify the Data Items

Having identified the requirements for the database, you can translate those requirements into identifiable elements of data. First you will simply identify data items. Later those items will become classes and data members. Delay that definition; it begins to define the solution, and, for now, you are working on a clear and unambiguous expression of the problem.

Organize the data items that you identify. Use 3 x 5 cards, an on-line notepad on your computer, or some other technique that allows you to itemize, sort, and shuffle.

Rummage around in the work you have already done, looking for potential data items. From your basis and your requirements, extract references to things that look like data items. Start by pulling the nouns out of your work. Each noun is a potential data item. Write the name of each onto a 3 x 5 card. Add everything else you know about the item to the card, highlighting the verbs. You'll use the verbs later when you identify the class methods.

The personnel system basis and requirements list delivers a collection of nouns like this:

- employees
- departments
- salary
- date hired
- projects
- schedules
- hours budgeted
- hours worked
- department number
- department
- name

- employee name
- employee number
- project number
- project name
- start date
- completion date
- assignments
- hours expended
- manager

You can use this list as an initial collection of the items that will be recorded in and reported from the database. As your work proceeds, you will change the list.

▼ Separate the Data Members from the Classes

You now have a collection of data items. Some of them will take the form of data aggregates that represent persistent classes, or *files* in the database. Others will be data members within those classes. This step distinguishes the two.

Unlike traditional database design, this step does not clearly separate the aggragates and the data elements. In an object-oriented design some of the so-called data elements will themselves be abstract data types, or classes. Your task is to recognize which of the data items are to be persistent classes and which are to be data members of persistent classes.

This is where you apply judgement, intuition, and guesswork. Look at the data items you have collected. Which of these items seem to be individual data members, and which seem more like logically organized aggregates of data members? Shuffle them up, sort them out, and move them around. The 3 x 5 card method works well here. It should become obvious which items are members and which are not. A date, for example, is clearly a data member even though it is likely to be an instance of an abstract data type, perhaps taken from a class library. It is not as obvious that an "assignment" is not a data member. Can you see how a department number is a data member, while a department

is an aggregate represented by a number of data members, including the department number?

The objective of this exercise is to separate the data members from the persistent classes so that you can take the next two steps. The data member nouns from the list above are in this subset of the list:

- salary
- date hired
- hours budgeted
- hours worked
- department number
- department name
- employee name
- employee number
- project number
- project name
- start date
- completion date
- hours expended
- manager

▼ Build the Data Member Dictionary

Now you begin to formulate the solution. Once you have identified what the data members are and have a list like the one above, build the data member dictionary, which enumerates the various data members in the database. It is important to itemize these members separately so that later you can identify the redundancies and interclass relationships.

The nouns in the list of data items are the ones that are not going to be persistent classes; they are going to be members of persistent classes. Some of them will be abstract data types. The salary will probably be an instance of a currency class. The dates will, no doubt be instances of a date class. The names will be instances of a string class.

The data types for several data items are not apparent just from this list. The employee number, department number, and project number could be

something other than simple integer types, depending on how the application defines them. The hours data items might be primitive integer types, or they might be abstract integer types with defined ranges.The format of the manager data item is unclear at this time. The requirement is to record department and project managers. You will address that one later.

You must collect everything that is known or that can be determined about each data member. At the very least, you will need to know the data type as was just shown. 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 member 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 manual system uses the data members. Sometimes these forms have accompanying procedures, and you can use these procedures to find the descriptions of the manual entries and extrapolate the requirements for the data members.

Describing dates, social security numbers, telephone numbers, zip codes, or shoe sizes is easy: these data members have known ranges, formats, and/or enumerated values, and there might be library classes that you can use. But other data members are not so well defined by common usage. Quantities and amounts might have limits—how high can they get? can they be negative? How many decimal places? The list goes on. Account numbers, vendor numbers, client numbers, employee numbers, part numbers, stock numbers, customer numbers, and untold numbers of other numbers are not universally defined. 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 members that will be in the database. You must complete as much of this definition as possible before you proceed with the design. When a data member has a clearly defined format and behavior, you should build a class to implement the member as an abstract data type.

▼ Gather the Data Members into Classes

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

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

- employees
- departments
- projects
- schedules
- assignments

Look at your requirements now to see which of those items should be persistent classes. None of the requirements call 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 potentially (but not absolutely) the persistent classes.

Now you can define the data representation of the persistent classes. To define a class, you define its data members. Use your trusty 3 x 5 cards. 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 related to the employee. But is the data member 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 member into the employee's class because there could be multiple values for that member for the same employee—he/she can work on more than one project. 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 member and viewing it in the light of its relevance to an employee, you can build a stack of 3 x 5 cards that are the data members for the **Employee** class. This stack will contain the cards for the employee number, employee name, date hired, salary, and department number. Assuming that the abstract data types for dates, currency, and strings already exist, and that the employee, project, and department numbers can be integers, you can make a first cut at the data representation of the **Employee** class design now. Figure 7.1 shows how that might look.

```
class Employee {
    int emplno;
    String name;
    Date date_hired;
    Currency salary;
    int deptno;
};
```

Figure 7.1 The **Employee** Class

Next you design the other persistent classes. Remember that you are never done. Always be willing to change what you have done before. Do not allow your current design to be constrained by the ones that precede it. If you cannot get something designed correctly because of an earlier design, then retreat and review that earlier design. Figure 7.2 shows a first cut at the **Department** and **Project** classes, which assume that data members such as **hours_budgeted** are unbounded integer types.

```
class Department {
    int deptno;
    String name;
};

class Project {
    int projno;
    String name;
    int hours_budgeted;
    Date start_date;
    Date completion_date;
};
```

Figure 7.2 The **Department** and **Project** Classes

The design has not considered as yet the **Assignment** class, the requirement to record employee hours expended on a project, or the requirement to record department and project managers. You have some design decisions to make about these items. It's been easy until now because the design is producing nice flat files. You'd almost think you were designing a relational database.

There are several things to consider about the **Assignment** class. The requirements state that an employee can work on several projects and that a project can have several employees working on it. The system has to record the hours expended by each employee on each project. You have three alternatives for this design. The relational database designer has only one. The alternatives are:

- Build an **Assignment** class that records the employee-project assignment and records the hours expended by the employee against the project.
- Put an array in the **Employee** class that contains an entry for each project that the employee works on along with the hours expended.
- Put an array in the **Project** class that contains an entry for each employee working on the project and the hours expended by that employee.

The relational database designer has the first alternative only. Which alternative do you choose for the object database? Your decision should consider the implications of each alternative. Consider the second alternative. There can be 100 employees and only 20 projects. The project array could have, at most, 20 elements for any employee object. That seems more efficient than using the third alternative where a project record might need an array of 100 employees.

Suppose you choose the second alternative and one of the requirements is to retrieve all the employees that work on a project. The retrieval would involve reading the entire employee file and scanning the project array of each to pick out the employees assigned to the project.

Given that requirement, you choose the third alternative instead of the second. But suppose another requirement is to retrieve all the projects that a selected employee works on. You have the same problem in reverse.

Not so bad, you say, because you do not have those requirements. Well, what if the functional requirements change after the system is running? Suppose those requirements are added to the basis. Besides, it looks like you do have one of those requirements. The requirement to report hours expended against a project implies that the system needs to deliver the sum of the hours expended as recorded in whichever array you chose. (More about that later.)

Not so bad, you say again, because the small number of employees and projects do not represent a serious performance penalty when you scan those files. Suppose, then, that the performance requirements change after the system is running. Suppose the company grows to 1000 employees and 200

projects. Suppose that the requirements expand to include more data about each assignment, such as expense items, duty location, task identification, and deadlines. What happens to performance then?

Why did the founders of relational technology eliminate repeating groups such as these two arrays from the data model in the first place? The object model supports arrays, and there are valid uses for them: an array of points can define a graphic rendering; an array of coordinates can define a map; an array of dots can define a screen; an array of notes can define a concerto. The example of employee-project assignments is not, however, a good application for an array. By looking ahead, you can forestall problems by learning from the relational tradition and normalizing the design. Remember that the design of a database is influenced more by what you must get out of it and how than by what you must store in it.

Figure 7.3 is the first cut at the **Assignment** class;

```
class Assignment {
    int empno;
    int projno;
    int hours_expended;
};
```

Figure 7.3 *The* ***Assignment*** *Class*

The design is beginning to take form, but there is a problem. The system maintains the hours expended against a project as the sum of those hours for each employee. Suppose the employee leaves the project. What do you do with the hours expended when you delete the **Assignment** object that associates the employee with the project?

Suppose the employee leaves the company. According to the rules of interclass integrity, you could not delete the employee object as long as another object was related to it, in this case the object that associates employees and projects. Once again, what do you do with those hours? Clearly, the **Projects** class needs to maintain an independent sum of hours expended, and so you must add that data member to its design. Furthermore, that sum cannot be expected to reconcile with the totals for the project from the **Assignment** objects for the project since some employees may have left.

The requirement to record department and project managers is still unresolved. A review of the requirements reveals that they are unclear as to whether a department or employee can have more than one manager. Furthermore, they do not specify if the manager of a department must be assigned as an employee of that department or if the manager of a project must be assigned to work on that project. Here is a situation where you iterate the design process, revisit the requirements, and make some modifications.

Assume that a department and a project can have only one manager each, that the department manager must be an employee of the department and the project manager cannot charge labor hours to the project, and therefore, may not be assigned as a worker on the project. How do you represent all that in the database?

There are some data interdependencies that a general-purpose database management system cannot enforce, and these appear to fall into that category. By adding a department manager employee number data element to the department class and a project manager employee number to the project class, the database can reflect the required management positions. Figure 7.4 shows the two classes with their manager data items in place.

```
class Department {
    int deptno;
    String name;
    int manager;  // an employee number
};

class Project {
    int projno;
    String name;
    int hours_budgeted;
    Date start_date;
    Date completion_date;
    int manager;  // an employee number
};
```

Figure 7.4 *Managers Added to Departments and Projects*

The manager employee numbers in both classes satisfy the requirement to record managers, but they introduce the potential for data disintegrity.

You will now have an **Employee** object that points to a **Department** object, which points to an **Employee** object. According to the requirements, the

department manager must be an employee of his/her own department. If the two objects do not point to each other, the database is out of synchronization.

Conversely, you will have a **Projects** object that points to its manager's **Employee** object, which might point to an **Assignment** object—assuming the manager can work on projects other than his/her own—which points to a **Projects** object. According to the requirements, the project manager must not be a worker on his/her own project. If the chain of objects complete so that the project manager is assigned to his/her own project, the database is out of sync again.

Maintenance of complex relationships such as these involves the application of a rule-based inference engine that is way beyond the scope of most general-purpose database management systems. The relational data model does not solve the problem any better than the object model can. The responsibility for such database integrity remains with the application. Failure to do so does not create the same kind of chaos caused by objects being related to nonexisting objects, a forbidden condition that the data model can enforce. At worst, the database will reflect relationships that the application prefers not to permit, and so the responsibility is placed at the correct doorstep.

▼ Identify the Key Members

A class must have a primary key data member and can have several secondary key data members. The primary key indentifies objects of the class. The secondary keys identify alternate ways that you can find an object. The employee number is the primary key for the **Employee** class. Every **Employee** object has a unique employee number. The department number is a secondary key for the **Employee** class. Every **Employee** object has a department number, and multiple employees can have the same department number.

The next step in the object database design is the identification of the primary and secondary keys.

Primary Keys

The four persistent classes are defined. Now you can identify the data member that, as the primary key, uniquely defines objects in each one. The **Employee**,

Department, and **Project** classes are easy. The employee number, department number, and project number serve that purpose.

What about the **Assignment** class? What is its primary key? It has employee number and a project number data members. Can either of them be the key? Since an employee can work on several projects, there will be multiple **Assignment** objects with the same employee number. The employee number cannot be the primary key. Since a project can have many employees working on it, there will be multiple **Assignment** objects with the same project number. The project number cannot be the key. What's left?

Remember from Chapter 2 that a many-to-many relationship such as the one that exists between employees and projects is represented in a relational database by a connector file that has the concatenated primary keys of the related files as its primary key. The **Assignments** class is the object database equivalent of a relational connector file. Therefore, its primary key will be the concatenation of the employee number and the department number. That combination of data members uniquely identifies each **Assignments** object.

If the work you have done up to now includes a would-be persistent class that has no uniquely identifying data member or combination of data members, then either the design is incomplete or the class cannot be persistent. If you cannot identify an object, you cannot retrieve it, even if you should allow it to become persistent.

This rule is consistent with relational theory, but some object-oriented database systems do not enforce it. In those systems, an object acquires an identity when it becomes persistent. The database manager assigns the identity and returns it to the program that created the object. The application must associate the system-defined identity with the object at a later time.

To be useful in a data processing context, an object needs to be unique, so it follows that there is something about every useful object that sets it apart from others of its class. Otherwise, there is no point to its persistence. Whatever sets the object apart is its real identity, and an arbitrary one assigned by a database manager is just a handle. Associating true object identity with an internal handle should be in the province of the database manager and not something the application needs to be concerned with.

If your persistent class lacks a primary key, redesign your persistent class.

Secondary Keys

Secondary keys are the ones that provide alternate access to the objects of a persistent class. In theory, you can retrieve any subset of objects by serially retrieving all the objects in a class and testing them against your search criteria. Many times this is appropriate. A monthly or yearly report does not need its own index to support its unique sort requirements. Other requirements are for on-line access, and they can use secondary keys. If you frequently retrieve all the employee objects who work for a selected department, then the department number in the **Employee** class should be a secondary key. If the receptionist looks up the telephone extensions of employees based on their names, then the employee name should be a secondary key in the **Employee** object.

Secondary keys support class relationships, which are discussed in the next section. When a secondary key is the same data item as the primary key of another class, an implied relationship exists between the classes.

In a connector class such as the **Assignments** class, the primary key is the concatenation of two key members, which are primary keys of the two classes being associated in a many-to-many relationship. The two key members in the **Assignments** class must not only be half each of the primary key, but must also be secondary keys. This permits the system to locate the first instance of an object with a specified value for either half and to navigate the connector objects retrieving the connections for the specified connected object.

▼ Identify the Class Relationships

In an object database, persistent classes are related when they contain a common data member and that data member is the primary key in one of the classes. The **Employee** class is related to the **Department** class because the employee record includes the department number data member, and the department number is the primary key to the **Department** class.

Because of this relationship, you can view the **Department** class 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 **Department** database rather than a class in a larger database. Other than to identify its manager, it needs nothing from the **Employee** class to fulfill its purpose. Second, the **Department** class is the table of information about the department in which an employee works. It validates the department number being stored in an employee's record and provides the department name employee.

For the implied relationship to work, it must have integrity. If an **Employee** object contains a department number, there should be a matching **Department** object.

In your design, you evaluate each potential relationship to see if it is real. Sometimes a control number is included in a class as information only. Users do not care if the corresponding object still exists; it may have been retired. For this reason, database management systems do not usually automatically infer a relationship simply on the basis of shared key data members. The database designer must explicitly declare the relationship so that the database manager knows whether to enforce or ignore it.

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, the connector class supports the relationship, and it must always be synchronized with the two classes it connects.

The relationships between classes can describe potential retrieval paths for multiple-class retrievals. Sometimes, the apparent paths are incorrect. Observe the relationships between the **Employee**, **Department**, and **Project** classes. If you designed a retrieval that began by retrieving a department object, 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 assigned to the department, 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 interclass 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.

▼ Identify the Methods

An object database consists of objects of persistent classes. A C++ class has not only its data representation, but its behavior as well. There is a misunderstanding about what this means. Some object-oriented programmers believe that an object-oriented database will store the methods along with the data. This is not practical for a C++ program. The methods are the public member functions, compiled by the compiler, cataloged in relocatable libaries, linked by the linker to your application programs, and stored with them in executable binaries. Those executable files are where the methods are kept. The compiler associates them with the persistent objects by virtue of the class definitions. There is no magic involved.

Nonetheless, you must identify the methods for your persistent classes. This part of the design is exactly the same as for the design of any other C++ program. Recall that when you built those 3 x 5 cards, you made notes on them about the data items. Return to the ones that refer to data items that became persistent classes. Look at the verbs in your notes. Those verbs are the first step toward defining the application-dependent behavior of the persistent classes.

Identify the ways that the application must massage the data members. You will immediately recognize the need for member functions in the class interface that let the application read and change the data members. Beyond that, look for more distinguishing behavior. Does the system need a value that it computes from among the data members of a class? Do you need the hourly rate for an employee to compute charges against a project? Add a member function to the **Employee** class to do the computation. Figure 7.5, which assumes a work year of 2080 hours, shows how that would work.

The example in Figure 7.5 is a small one, but it makes this point: add object-oriented methods to your persistent classes the same way you would in any other C++ program design. The so-called persistent method is bound to the persistent object, not because some heretofore unknown operating-system/database-system magic cookie stores the method in the persistent object database, but because the program that retrieves the object uses the same class definition and class library used by the program that created the object.

```
const int hours_year = 2080;
class Employee {
    // ...
    Currency salary;
public:
    virtual Currency Rate() { return salary / hours_year; }
};
```

Figure 7.5 *A Persistent Class Method*

▼ Make the Class Persistent

The last step—other than to continue to iterate—is to integrate the persistent object class design into the persistent object database management system. You will need to add attributes to the key data members and the persistent classes so that they can participate in their own persistence.

The nature of those attributes will depend on the database management system that you use. With the PARODY software contained in this book, those attributes consist of inheritance, calling some base class functions at the correct times, and some special member functions added to the persistent class that the base will call.

▼ Summary

Now you know what constitutes an object database and how to design one. A database design needs a software system to process it; you've done some of that already. The member functions of the classes you designed are a large part of your database application. Database software consists of two parts: those applications programs that process the objects—from within and outside of the object—and the general-purpose software that manages database organization, storage, and retrieval. This software is called the database management system. Chapter 8 introduces PARODY, the database management system that you will use.

▼ References

C Database Development, Second Edition, Al Stevens, *1991,* New York: MIS:Press.

Chapter 8

PARODY

"Parodies and caricatures are the most penetrating of criticisms."
Aldous Huxley

PARODY is the Persistent, Almost Relational Object Database management system implemented in this book as a C++ class library. PARODY achieves the objectives identified in Chapter 5. It implements a persistent object database with a relatively simple interface to its classes. Its data model resembles the relational model in some respects and assumes the properties of C++ objects in others—a parody of sorts. This chapter and the next introduce PARODY and show you how it works. This chapter describes how to define the classes in a PARODY database and how PARODY and the classes coordinate the persistence of objects of their own type. Chapter 9 discusses how an application program declares and uses the persistent objects. You will also learn in Chapter 9 an alternative to PARODY's almost-relational data model where you can

retrieve objects by address rather than by key values—in effect, using navigational rather than associative access. Appendix B is a programmer's reference guide for using PARODY, and Appendix C contains the source code.

Defining a set of related persistent classes is equivalent to building a database schema that describes the files' contents. Most traditional database management systems use a data definition language to describe the database schema. PARODY uses the C++ class definition into which you encapsulate not only a description of the data members in each class, but the integration of the class with the PARODY database manager as well.

▼ Defining a PARODY Database

You define a database by designing classes that represent the persistent objects and by designating the key data members of those classes. The persistent object and key member classes derive from abstract base classes defined in the PARODY class library. Objects of the persistent class types cooperate in their own persistence by calling PARODY member functions at specific times and by providing member functions that PARODY can call to perform tasks that require knowledge of the content and format of the persistent object.

Assigning Class Identifiers

C++ does not have type identification. PARODY needs to distinguish types so that it can locate objects of each type in the database. You must assign unique type identification numbers to each of your persistent classes. These numbers should start at zero and increment by one. Each distinct database has its own set of type identifiers starting at zero. You assign these values in constants that are within scope of the constructor functions of your classes. If you put them in one enumerated type in one header file, they will be easier to maintain. Figure 8.1 is an example of the assignment of class identifiers.

```
enum ClassIdent {
    EMPLOYEE,
    DEPARTMENT,
    PROJECT,
    ASSIGNMENT
};
```

Figure 8.1 *Class Identifiers*

Someday, when the C++ language formally supports type identification, such class identification values will be unnecessary.

Declaring the PARODY Database

To use persistent objects, a program must declare the persistent object database first. PARODY includes a class named **Parody**. You must declare an object of this class for each database. The object must be in scope for as long as your program creates and retrieves persistent objects. The address of the object must be available to the constructors of your persistent classes. Figure 8.2 is an example of how a program declares a **Parody** database object.

```
Parody *personnel;

void main()
{
    personnel = new Parody("PERSONNEL");
    Employee empl(123);
    Department dept(5);
    // ...
    delete personnel;
}
```

Figure 8.2 *Declaring a PARODY Database*

The "PERSONEL" initializer in the **Parody** declaration is the name of the database. PARODY will use it to name two files, one for the objects and one for the index. These files are discussed later in this chapter. The initializer is in the format of a file name without a file extension. In MS-DOS systems the initializer must be a valid one-to-eight-character DOS file name, for example. PARODY adds the file extensions ".DAT" and ".NDX" to the two file names.

If you declare a **Parody** database and none exists with the name of the initializer, PARODY builds a new database with that name, and you can begin using it by creating persistent objects for it.

Defining the Persistent Object Class

A PARODY object is an instance of a class. You define the class, giving it attributes that make it persistent in the **Parody** object. PARODY provides the base classes and the interface that make it possible. Your class manages its own persistence along with the help of the behavior it inherits from PARODY.

Class Definition

A persistent class derives from the **Persistent** abstract base class, which is defined by PARODY. At a minimum, the persistent class includes a primary key data member, a constructor, a destructor, and two private member functions named **Read** and **Write**. Figure 8.3 is a **Department** class, which represents the minimum persistent class along with a **name** data member and member functions in the class interface to set and read the department name. This class and the other example classes in this chapter figure prominently in the case studies of Chapter 10.

The **Department** class in Figure 8.3 acquires the persistent attribute by deriving from the **Persistent** class. Other than for the base class specification, the only visible things that distinguish the **Department** class from a non-persistent class are the **Read** and **Write** member functions. The constructor and destructor have a role to play in the class's persistence, as well. The following sections address the four basic cooperating member functions.

```
class Department : public Persistent {
    DepartmentNo deptno;      // primary key
    String name;
    void Read();
    void Write();
public:
    Department(int dn = 0);
    ~Department();
    String &Name()            { return name; }
    void SetName(String& nm) { name = nm; }
};
```

Figure 8.3 *A Persistent Object*

Persistent Object Constructor

The constructor has three responsibilities in a persistent class. First, it initializes the base **Persistent** constructor with a reference to the **Parody** database object and the class identification value. Second, it initializes the persistent object's primary key data member to identify the object. Then, after any custom construction of its own, the constructor calls the **LoadObject** member function in the **Persistent** base class. Figure 8.4 is an example of a persistent class constructor function.

```
Department::Department(int dn) :
    Persistent(*personnel, DEPARTMENT), deptno(dn)
{
    // ...
    LoadObject();
}
```

Figure 8.4 *The Persistent Object Constructor*

Usually, the constructor's custom construction consists of setting the data members to null values in case the specified object is not in the database. If other data members need special construction, such as allocating memory from the free store, the persistent object's **Read** member function can do it.

The constructor should call the **LoadObject** function even if the key data member value is null. PARODY uses the first call to **LoadObject** from a class in a program's execution to capture the class's key data member configuration.

Persistent Object Destructor

The persistent class destructor has one responsibility. It calls the **SaveObject** member function in the **Persistent** base class before completing its own destruction. Figure 8.5 is an example of a persistent class destructor function.

```
Department::~Department()
{
    SaveObject();
    // ...
}
```

Figure 8.5 *The Persistent Object Destructor*

Usually, the **SaveObject** function call is all the destruction that the destructor needs. If any data members need further destruction, such as deleting allocated free store memory, the persistent class's **Write** member function can do it.

Read Member Function

The persistent object's class constructor calls the **Persistent** base class's **LoadObject** function to locate and read an object from the PARODY database. That function in turn calls the derived class's **Read** function. **Read** is defined in the **Persistent** base class as a pure virtual function, so all derived classes must include one.

The **Read** function reads the data members into the persistent object by calling the base class's **ReadObject** function for each data member. Figure 8.6 is an example of a persistent class's **Read** function.

```
void Department::Read()
{
    int dpno;
    ReadObject(dpno);
    deptno = DepartmentNo(dpno);
    ReadObject(name);
}
```

Figure 8.6 *The Read Member Function*

The **Read** function calls the base class's **ReadObject** function once for each data member in the object. The **ReadObject** function accepts a reference to the data member and does the physical input operation into the data member. The **ReadObject** understands certain generic abstract data types, such as **String** and **Currency**, but it does not understand specific ones such as the **DepartmentNo** object. For that reason, the program uses a local variable and then assigns the read value to the object's data member.

It might seem odd that the persistent derived and base classes bounce back and forth in this series of function calls. The derived class's constructor calls the base **LoadObject** function, which calls the derived **Read** function, which calls the base **ReadObject** function. Later in this chapter you'll learn why. The same pattern applies to the **Write** function discussed next.

If the object contains variable length fields such as arrays, lists, or data items in dynamic free store allocations, the **Read** function constructs them. To know how many array or list elements to construct or how much free store to allocate and, therefore, how many data items to read, the **Read** function first calls **ReadObject** to get an integer value that represents the dimension of the array or the buffer size. The **Write** function will have written that value when the object became persistent, as you'll see next.

Write Member Function

The persistent object's class destructor calls the **Persistent** base class's **SaveObject** function to save new or changed objects in the PARODY database. That function in turn calls the derived class's **Write** function. **Write** is defined in the **Persistent** base class as a pure virtual function, so all derived classes must include one.

The **Write** function writes the data members from the persistent object by calling the base class's **WriteObject** function for each data member. Figure 8.7 is an example of a persistent class's **Write** function, which is the reciprocal of the **Read** function.

```
void Department::Write()
{
    WriteObject(deptno.DeptNo());
    WriteObject(name);
}
```

Figure 8.7 *The Write Member Function*

The **Write** function calls the base class's **WriteObject** function once for each data member in the object. **WriteObject** understands the format of certain generic abstract data types the same way that **ReadObject** does.

If the object contains variable length fields such as arrays, lists, or data items in dynamic free store allocations, the **Write** function destroys them after it writes them. To tell the PARODY copy of the object how many array or list elements or how much free store it contains for a particular data member, the **Write** function first calls **WriteObject** to write an integer value that represents the dimension. The **Read** function uses that value to read the object into memory when the object is instantiated at some later time.

Defining Keys

Figure 8.3 shows a persistent class definition where the first data member is an object of the **DepartmentNo** abstract base class. A comment on that line of code identified the data member as the class's primary key. How does PARODY know that the data member is a key and that it should be treated differently? The key data items are derived from the **Key** class, which gives the data item the behavior of a key. Its constructor registers it as a key with the **Persistent** class currently being constructed.

Building a Key Class

A key is one of the data members of a persistent class. Its data value can be an object of any fixed-length type, but it must be represented in a class derived from the PARODY **Key** class.

A key's fixed-length property is critical to its operation. This requirement is levied by PARODY's indexing mechanism, the B-tree algorithm, which works with fixed-length key values. This chapter explains the B-tree algorithm at some length in a later section.

Your derived key class specifies its length implicitly the first time it writes a key value to the index file. PARODY depends on your program to maintain a consistent key length for all subsequent reads and writes of key data values of the same key data member for objects of the same class.

Figure 8.8 is an example of a key class.

```
class DepartmentNo : public Key{
    int deptno;
    // ----- overloaded relational operators
    int operator> (Key &key)
        { return deptno > ((DepartmentNo &) key).deptno; }
    int operator== (Key &key)
        { return deptno == ((DepartmentNo &) key).deptno; }
    // ----- overloaded assignment operator
    Key& operator= (Key &key);
    // ----- key input/output
    void Write(fstream& ndx)
        { ndx.write((char *) &deptno, sizeof deptno); }
    void Read(fstream& ndx)
        { ndx.read((char *) &deptno, sizeof deptno); }
    // ----- make a null key
    Key *Make() { return new DepartmentNo(); }
    Bool isNullValue() { return (Bool) (deptno == 0); }
public:
    DepartmentNo(int dn = 0) { deptno = dn; }
    int DeptNo() { return deptno; }
};
```

Figure 8.8 *A Key Class*

Figure 8.8 shows what you must provide in a derived key class. The **deptno** data member represents the data value of the key and is the value by which PARODY will index the object. The **Key** base class defines the private member functions as pure virtual functions, so the derived class must include them.

Overloaded Relational Operators

The overloaded greater than (>) and equal to (==) operator functions logically compare the values in two key objects. These functions use the **Key** class type for the two objects to compare, and they permit the PARODY indexing algorithm to order and locate keys in the B-tree. If your application does similar comparisons, you would include additional overloaded relational operator functions in the public class interface, ones that use your derived class type instead of the base class type. This is so you do not need to cast the operands to the **Key** type when you use the operators.

Overloaded Assignment Operator

The overloaded assignment operator assigns the contents of a derived **Key** object to another one of the same class. If your application also assigns key objects, you might include an overloaded assignment operator in the key class's public interface that uses the derived class type. Do not use your overloaded operator to assign new key objects to the key objects in the persistent class. The ones in the class will have some internal control values that must not be disturbed. It's better to simply work with the key's data values and ignore its **Key** properties outside of the class.

Read and Write Member Functions

The **Read** and **Write** member functions read and write the key's data values from and to a specified **fstream** object. These functions must always read and write the same number of characters for a particular key.

Do not confuse these functions with the **Read** and **Write** functions of the derived persistent class, and do not assume that because you have these functions you do not need to read and write the key data members from within the persistent class. These functions are to support the B-tree index files only. Since the application does not normally have visibility into the index files, you should include the index data values in the **ReadObject** and **WriteObject** calls in the derived persistent class's **Read** and **Write** functions.

The Make Function

Every key must provide a **Make** function. It constructs a null object of the derived key type with the C++ **new** operator and returns its address. The key class does not **delete** this construction. PARODY calls the **Make** function to

build empty buffers of the correct size into which it reads keys and to identify the key class with respect to the class identification of the objects it indexes and the relative position of the key object within the persistent class. The callers of the **Make** function handle the **delete**. Your program should never need to call the **Make** function, but you must provide it, nonetheless.

The isNullValue Function

The **isNullValue** function tells PARODY if a key value is null. The meaning of *null* depends on the data members that constitute the key itself, so the persistent class's keys must provide the information. PARODY does not attempt to verify class relationships when the relating key value is null. This allows you to add objects to the database when two classes are mutually related.

For example, the **Employee** class might be related to the ***Department*** class. An employee is assigned to a department. The **Department** class might be related to the **Employee** class in that each department object has an employee assigned as the manager. You must be able to add objects with null relating keys because the database is empty to start with and you must be able to add either employees without department numbers or departments without manager employee numbers as a first step.

Secondary Keys

The first **Key** object that the persistent class declares is its primary key. All other **Key** objects are secondary keys. The **DepartmentNo** key is the only key in the **Department** class used for examples up until now in this chapter. Figure 8.9 shows a **Project** class that has two **Key** objects.

```
class Project : public Persistent  {
    ProjectNo projno;     // primary key
    String name;
    EmployeeNo manager;   // secondary key
    void Read();
    void Write();
public:
    Project(int pn = 0);
    ~Project();
    void Display();
    void DisplayLine();
};
```

Figure 8.9 *A Persistent Class with Two Keys*

The position of the key data members determines which one is the primary key. All but the first one are secondary. In Figure 8.9, the **Project** class has a **ProjectNo** object as its primary key and an **EmployeeNo** object as a secondary key to record the project manager.

PARODY treats primary and secondary keys differently. A class in the database may have only one object with a particular primary key value. There may be only one project with project number 123, for example. There may be several projects with the same manager, however.

Concatenated Keys

When a database design calls for the relational equivalent of a connector file, the **Persistent** class will include a concatenated key as its primary key. A concatenated key contains two other key objects as its data values. Each of them is a primary key to another class and is treated as a secondary key to the connector class. The application program can retrieve objects based on either key value or the combination of the two.

For example, to represent the many-to-many relationship, where employees can work on many projects and each project can have many employees working on it, you would design an **Assignment** class similar to the one in Figure 8.10.

```
class AssignmentID : public Key  {
    // ---- concatenated key
    EmployeeNo emplno;          // secondary key
    ProjectNo  projno;          // secondary key
    // ...
public:
    AssignmentID(int empl = 0, int proj = 0) :
            emplno(empl), projno(proj);
    // ...
};

class Assignment : public Persistent  {
    AssignmentID assignment;  // primary key
    // ...
public:
    Assignment(int empl = 0, int proj = 0);
    // ...
};
```

Figure 8.10 A Class with a Concatenated Key

You can use the **AssignmentID** object as a primary key and the **EmployeeNo** and **ProjectNo** objects as secondary keys for retrievals. There is one further requirement to complete the definition. Due to the sequence in which C++ executes constructors, the **EmployeeNo** object will appear to PARODY to be the primary key rather than the **AssignmentID** object. You must tell PARODY to override that assumption. The **Key** base class includes the **PrimaryKey** member function to tell a key to assume the role of the primary key in the class, regardless of its position in the class definition. The derived persistent class's constructor must call that function. Figure 8.11 is an example of the **Assignment** class constructor, which uses the **PrimaryKey** function to override the implied primary key.

```
Assignment::Assignment(int empl, int proj) :
                    Persistent(*personnel, ASSIGNMENT),
                   assignment(empl, proj)
{
    assignment.PrimaryKey();  // make assignment primary
    // ...
    LoadObject();
}
```

Figure 8.11 *Using The PrimaryKey Function*

Relating Classes

You have seen that objects in one class are implicitly related to objects in another class by reference. If a class has a secondary key of the same derived **Key** type as the primary key of another class, an implicit relationship exists. You may or may not elect to let PARODY enforce that relationship. For example, it might be acceptable to retain assignments information long after an employee has left or the project has closed down. On the other hand, it might not be acceptable to delete an employee who is still on record as the manager of a department. The degree to which the system enforces the relationships is a function of the application itself. Therefore, PARODY requires that you identify the relationships that you want it to enforce. Figure 8.12 shows how a class specifies a relationship by calling the **Persistent** base class's **Relate** member function from within the constructor.

```
class Project : public Persistent {
    ProjectNo projno;     // primary key
    EmployeeNo manager;   // secondary key
    // ...
public:
    Project(int pn = 0);
    // ...
};


Project::Project(int proj) :
  Persistent(*personnel, PROJECT), projno(proj)
{
    manager.Relate(EMPLOYEE);
    hours_expended = 0;
    LoadObject();
}
```

Figure 8.12 *Relating Classes*

The **Relate** function call in Figure 8.12 tells PARODY that the **Project** class and the **Employee** class are related. PARODY will not allow you to add or change a project object where the **manager** data item is non-null and represents an **Employee** object that is not in the database. The **Relate** function call is all that you need to enforce that relationship.

Conversely, PARODY will not allow you to delete an **Employee** object if any related **Project** objects still exist. However, you might need to take one more step to ensure this enforcement. PARODY builds a table in memory of classes and their relationships as you declare objects. It does not know about the relationships between classes until a constructor calls the **Relate** function. Therefore, if a program deletes an **Employee** object but that program has never declared a **Project** object *during the current execution of the program*, PARODY does not know about the relationship because no **Project** object has called the **Relate** function to record it. PARODY would allow you to delete the **Employee** object regardless of whether or not any **Project** objects exists in the database that refer to the employee being deleted.

To ensure that PARODY enforces all relationships, instantiate an empty object of each class that calls **Relate** from its constructor. Do this at the beginning of the program. You can let the empty objects go out of scope right away. Simply having them declared once is enough to record the relationships

for the duration of the program's execution. Figure 8.13 shows how a program handles that.

```
void main()
{
    personnel = new Parody("PERSONEL");
    {
        // ---- empty objects to declare relationships
        Employee empl;
        Department dept;
        Project proj;
        Assignment assgn;
    }
    // ...
    delete personnel;
}
```

Figure 8.13 *Ensuring Relationships*

Observe that the program in Figure 8.13 declares the four empty objects from within a brace-surrounded block. This form allows them to go immediately out of scope so they will not occupy stack space for the balance of the program's execution.

Multiple-Copy Objects

None of the persistent class examples in this chapter have copy constructors or overloaded assignment operators. This is not an oversight. As a rule, you do not want to make in-memory copies of objects. Making copies of a persistent object has its perils. PARODY would not know which copy to save on the database when the object goes out of scope. Therefore, PARODY must assure that there is only one copy of any particular persistent object in memory at a time.

There are several ways that a C++ program copies objects. The copy constructor and assignment are two. In a third way, the program declares another instance of the same object. Rather than allow these processes to make copies of an object, the **Persistent** class must take measures to prevent it.

For classes where you know that you will be making copies, the **Persistent** class uses what is called the *handle/copy* idiom to implement reference

counting. This technique uses a handle class as the one that a program declares an instance of. The handle class contains a pointer to a copy class, which is the body of the object and which contains all the members. The copy class also contains a count of the current number of copies. When the first copy of the class is constructed, the constructor uses the **new** operator to create a new copy object. Each new handle object gets the address of the copy object in its pointer, and the counter in the copy object gets incremented. When a handle class's destructor runs, it decrements the counter. When the counter is zero, the handle class deletes the copy object. The handle class provides the copy constructor and overloaded assignment operator.

PARODY also monitors for the presence of other instances of the object. When it senses that you are instantiating a copy, it substitutes the master copy.

A **Persistent** class cooperates in its participation in the handle/copy idiom. The class design includes a handle class, which derives from the base **Handle** class and a copy class, which derives from the **Persistent** class. The program's interface to the object is through calls to functions in the handle class. Those functions pass the calls through to the **Persistent** body class. Figure 8.14 shows how the **Employee** class uses the handle/copy idiom.

The **EmployeeBody** class contains everything that must be in a derived **Persistent** class, except that all the members are private, and the **Employee** class is a **friend**. The **Employee** class refers all interface member function calls to the single instance of the **EmployeeBody** object through the **body** pointer. The **ebody** function is for convenience to minimize casting the **Handle** class's **body** pointer, which is a pointer to the **Persistent** type.

The **Employee** constructor calls the **ConstructBody** function, passing it the address of the **EmployeeBody** object created by the **new** operator. If another instance of the same object exists, the **ConstructBody** function will delete the new one and substitute the address of the existing one in the **body** pointer.

Persistent Objects within Persistent Objects

This section explains how to embed a persistent object within another one. You would do that when the application always needs a related object. For example, if you have a **Task** class that derives much of its meaning from the

Project class with which it is associated, you might find that your application fetches the parent **Project** object every time it declares a **Task** object. Rather than doing this every time yourself, you can define the classes so that the **Project** object is embedded in the **Task** class. Whenever you declare a **Task** object, the associated **Project** object is automatically declared, too. Figure 8.15 shows the **Task** class.

```
class EmployeeBody : public Persistent    {
    friend Employee;
    EmployeeNo emplno;
    DepartmentNo deptno;
    String name;
    void Read();
    void Write();
    EmployeeBody(int en);
    ~EmployeeBody();
};

class Employee : public Handle    {
    EmployeeBody *ebody()
        { return ((EmployeeBody *)body); }
public:
    Employee(int en = 0)
      { ConstructBody(new EmployeeBody(en)); }
    ~Employee() {}
    int EmplNo()    { return ebody()->emplno.EmplNo(); }
    int DeptNo()    { return ebody()->deptno.DeptNo(); }
    String &Name() { return ebody()->name; }
};
```

Figure 8.14 *A Handle/Copy Persistent Class*

To create a **Task** object, instantiate it with a task number key that is not in use. The default constructor for the **Project** class will instantiate an empty embedded object. When your program has determined which project object to associate with the new task, use code like that shown in Figure 8.16 to build the embedded project object in the new task object.

```
class Task : public Persistent  {
    TaskNo taskno;   // the primary key
    String name;
    Project proj;    // embedded object
    // ...
public:
    Task(int tno) : Persistent(*personnel, TASK), taskno(tno)
      { LoadObject(); }
    ~Task()
      { SaveObject(); }
    // ...
};

void Task::Read()
{
    int tno;
    ReadObject(tno);
    taskno.SetTaskNo(tno);
    ReadObject(name);
    int projno;
    ReadObject(projno);
    proj.SetProjNo(projno);
    proj.LoadObject();
}

void Task::Write()
{
    WriteObject(taskno.TaskNo());
    WriteObject(name);
    WriteObject(proj.ProjNo());
}
```

Figure 8.15 *An Embedded Object*

```
    proj.SetProjNo(projno);
    proj.LoadObject();
```

Figure 8.16 *Embedding the Object*

If the application might have multiple tasks declared that have the same embedded project object, design the **Project** class to employ the handle/copy idiom to accommodate multiple copies of the same object.

If the application changes the embedded object and wants that change recorded on the database, the application must call **ChangeObject**, discussed later, for the embedded object. Calling **ChangeObject** for the outer object does not automatically call it for any embedded objects.

▼ How PARODY Brings it Together

With derived **Key** and **Persistent** classes defined, the next thing to do is write a program that builds a database and declares and uses persistent objects. This section is an overview of how PARODY works with your classes to make the objects persistent. You will revisit some of the procedures that this chapter already discusses, but, instead, from the viewpoint of how PARODY uses the procedures, and in more detail.

Opening the PARODY Database

The first thing your program does is declare the **Parody** object just as the example in Figure 8.13 does. This declaration opens the database if it exists and creates one if it does not. The database is identified by the filename that you provide in the **Parody** declaration.

Constructing a Persistent Object

When you declare a persistent object, the sequence of constructors is important to how PARODY works. Consider a derived **Persistent** class such as the **Project** class in Figure 8.17.

```
class Project : public Persistent  {
    ProjectNo projno;     // primary key
    EmployeeNo manager;   // secondary key
    // ...
public:
    Project(int pn = 0);
    ~Project();
    // ...
};
```

Figure 8.17 *A Persistent Class*

The Persistent Base Class

When you declare an object of the **Project** class in Figure 8.17, the first constructor that executes is the one for the base **Persistent** class. That constructor initializes the base member variables and it initializes a static global **Persistent** type pointer named **thispers** to point to the object being constructed.

The Keys

Next comes the **Key** base class constructor for the **ProjectNo** object followed by the **ProjectNo** object's constructor. The **Key** constructor uses the **thispers** pointer to add its key object to a list of keys associated with the persistent object.

The **Key** base and **EmployeeNo** derived constructors execute next in turn, and the key appends itself to the list. This sequence is what establishes the **ProjectNo** object as the primary key. Its constructor executes first, so it is first on the list.

The Derived Persistent Class

After all the keys are constructed, the constructor for the derived persistent class executes. Figure 8.18 is an example of that constructor.

```
Project::Project(int proj) :
  Persistent(*personnel, PROJECT), projno(proj)
{
    // ...
    LoadObject();
}
```

Figure 8.18 *Constructor for a Derived Persistent Class*

At a minimum the derived persistent class constructor initializes the primary key with a search value, the other data members with null values, and calls the base class's **LoadObject** member function to load the specified object from the database into the data members in memory.

The LoadObject Function

The first thing the **LoadObject** does is clear the **thispers** pointer so that any further construction of **key** objects will not try to associate themselves with the object in the primary/secondary ranks. This measure is important because the indexing algorithms make copies of keys for different reasons throughout the life of the object, and the key constructors need to know the difference.

LoadObject next looks to see if the program has already instantiated a copy of the object. If so, the function substitutes the existing copy for the one being constructed. This works only if the class uses the handle/copy idiom. If you instantiate a second copy of an object that does not use the idiom, the **LoadObject** function will not load it from the database but will leave you with an empty object.

If no copy of the object is already declared, **LoadObject** uses the primary key value that the object's constructor initialized to search for the object in the database.

If the object exists in the database, **LoadObject** positions the PARODY file system to where the object is stored. **LoadObject** does not know enough about the object itself to load it. This is where the cooperation of the derived persistent class comes in. **LoadObject** calls the derived class's **Read** function so it can specify the size of each data member and the address into which it is read.

If the object does not exist in the database, **LoadObject** records that condition in the object and returns, assuming that the calling constructor has properly initialized the data members to null values.

The Read Function

The derived class's **Read** function calls the base class's **ReadObject** function for each of the data members. Figure 8.19 is an example of a persistent class's **Read** function.

Following the last **ReadObject** call, the **Read** function returns to the **LoadObject** function, which finishes up and returns to the object's constructor.

```
void Project::Read()
{
    int prno;
    ReadObject(prno);          // read the proj #
    projno.SetProjNo(prno);    // put proj # in obj
    ReadObject(name);          // read the name
    int mgr;
    ReadObject(mgr);           // read the mgr's #
    manager.SetEmplNo(mgr);    // put mgr in obj
    ReadObject(hours_expended); // read hours
}
```

Figure 8.19 *A Persistent Class's Read Function*

The ReadObject Function

There are several overloaded **ReadObject** functions, one for each of several data types. They all reduce to a common function that has an address and a character count as its parameters. That function uses the PARODY file input/output system to read the correct number of characters into the specified address. The function assumes that the PARODY file system has already been positioned to the correct address. The read operation is not direct, however. A PARODY database is organized into fixed-length nodes. Each object occupies one or more nodes in a linked thread, depending on the object's length. The **ReadObject** function reads data characters by following the object's thread in the file. This chapter describes PARODY's input/output system of nodes in a later section.

Destroying a Persistent Object

When you destroy a persistent object, the sequence of destructors is as important as the sequence of constructors was to the object's declaration. Consider the destruction of the **Project** class in Figure 8.17.

Destroying the Derived Persistent Class

The first destructor executed is the derived class destructor. Figure 8.20 is the destructor for the **Project** class.

```
Project::~Project()
{
    SaveObject();
    // ...
}
```

Figure 8.20 *Destructor for a Derived Persistent Class*

When the destructor begins to execute, nothing in the memory copy of the object has been destroyed yet: all the data members are intact. The database copy of the object, if one existed, has the same state that it had when the program instantiated the object in memory. The destructor calls the **SaveObject** function before it does anything else.

The SaveObject Function

If the object does not exist on the database when the program instantiated it, and if the program has specified that the object is to be added to the database, a procedure that Chapter 9 addresses, the object needs to be written to the database.

Likewise, if the object exists and the program has specified that the data members have changed, the object needs to be written.

The **SaveObject** function assigns a database address to a new object and adds its key values to the class's indexes. If the object is changed, the **SaveObject** function positions PARODY's file system to the object's address and updates any changed key values in the existing indexes. In either case, the **SaveObject** function then calls the derived class's **Write** function to write the data members.

If the object exists and the program specifies that the object is to be deleted, the **SaveObject** function releases the database file space owned by the object, deletes the key values from the class's indexes, and does *not* call the **Write** function.

The Write Function

The derived class's **Write** function calls the base class's **WriteObject** function for each of the object's data members. The sequence of calls must mimic exactly the sequence of **ReadObject** calls in the **Read** function. Figure 8.21 is an example of a persistent class's **Write** function.

```
void Project::Write()
{
    WriteObject(projno.ProjNo());  // write proj #
    WriteObject(name);             // write name
    WriteObject(manager.EmplNo()); // write mgr's #
    WriteObject(hours_expended);   // write hours
}
```

Figure 8.21 *A Persistent Class's Write Function*

Following the last **WriteObject** call, the **Write** function returns to the **SaveObject** function, which pads and writes the last node owned by the object, finishes up, and returns to the object's destructor.

The WriteObject Function

There are several overloaded **WriteObject** functions, one for each of several data types. They all reduce to a common function that has an address and a character count as its parameters. That function uses the PARODY file input/output system to write the correct number of characters from the specified address. The function assumes that the PARODY file system has already been positioned to the correct address. The **WriteObject** function writes data characters by following the object's node thread, growing new nodes if necessary.

Destroying the Keys

Destructors for the **Key** objects execute after the derived persistent class's destructor, with the derived key destructor executing ahead of its base **Key** class destructor. The base destructor is empty, and PARODY requires nothing from the derived constructor.

Destroying the Persistent Base Class

The base **Persistent** class destructor executes last. It cleans up some tables built by the base class and returns. The object is now destroyed from within the program and properly restored to or deleted from the database, as appropriate.

Object Input/Output: an Overview

Figure 8.22 summarizes the flow of function calls to load and save an object.

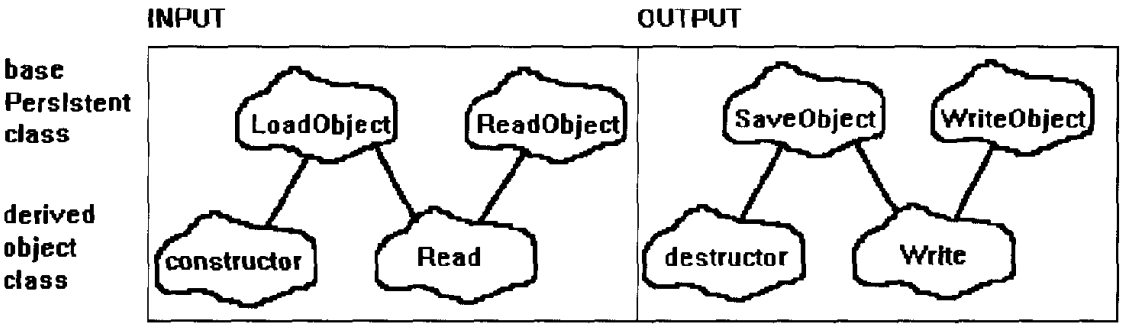

Figure 8.22 *Object Persistence Flow*

To load an object, the derived class's constructor calls the **LoadObject** function in the base **Persistent** class, which calls the **Read** function in the derived class, which makes several calls to the **ReadObject** function in the base class.

To save an object, the derived class's destructor calls **SaveObject** in the base class, which calls **Write** in the derived class, which makes several calls to **WriteObject** in the base class.

Your responsibilities in persistent class design are to provide the derived class's constructor and destructor and its **Read** and **Write** functions. None of these functions needs to understand the complexities of the database file structures. Their responsibilities are focused on the format of the object itself.

Integrity Checks

Chapter 9 describes how the using program calls the **AddObject**, **ChangeObject**, and **DeleteObject** functions to tell the **Persistent** base class how to handle the object in the **SaveObject** process. When those functions are called, they decide whether or not the object, as it is configured at that time, maintains or violates class relationship integrity.

The **AddObject** and **ChangeObject** functions test each of the secondary keys in the object. If the key is related to another persistent class, the functions

search the related class in the data base for an object that matches the value of the key. If no match is found, the functions refuse to set the flags that will tell the **SaveObject** function to write the object to the database when it is being destroyed.

The **DeleteObject** function tests to see if any other classes are related to the class of the object being deleted. If so, the function searches those classes to see if other objects are related to the one being deleted. If so, the function refuses to set the flag that tells **SaveObject** to delete the object.

In all three cases, when the functions refuse to do what the program has requested, the functions return a false value that the calling program can test.

It is important to understand the timing of these integrity checks. If you call one of the three functions, which approves the add, change, or delete, and then you modify the object or the database so that the action would not have been approved, the action will occur anyway.

▼ PARODY File Architecture

This section contains a technical description of the PARODY file structures. Programmers do not need to understand these structures in order to use PARODY, but knowledge of them will be helpful if you get into some detailed database troubleshooting.

A PARODY database consists of two files: the data file, which contains the objects; and the index file, which contains key indexes to the objects. The **Parody** object opens or creates these two files when the program declares the object. The data file has the file extension .DAT, and the index file has the file extension .NDX. The file architecture must support:

- variable-length persistent objects;
- indexes into and objects of multiple classes;
- persistent objects that can grow or shrink in length or be deleted between the time a program declares one and when the program destroys it; and
- reuse of deleted file space.

To support these objectives, PARODY uses a system of file nodes, described in the next section.

Nodes

Both files in a PARODY database are managed by an input/output system of fixed length file nodes in a thread. A file contains a header record followed by node records. Figure 8.23 shows this structure.

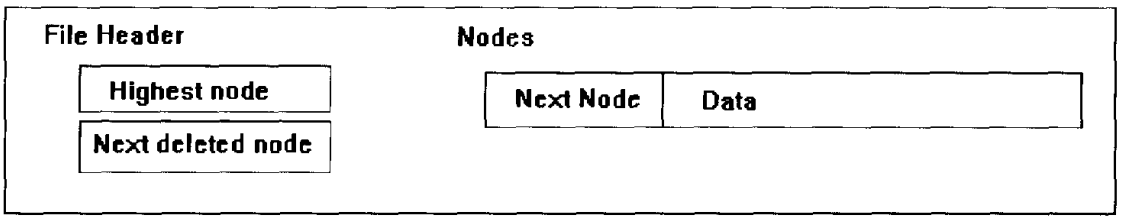

Figure 8.23 *File Header and Node*

Nodes are addressed logically by node numbers starting with 1. PARODY uses the node system two ways. The index files are organized into fixed-length nodes, which inherit the behavior of the **Node** class. The objects are unformatted, variable-length data streams that could extend beyond the boundaries of a node, so they declare **Node** objects and copy their data values into and out of the nodes.

Header Record

The header record contains two node numbers. The first node number points to the first node in a chain of deleted nodes. The second node number points to the highest node that has been allocated. Header records are implemented as objects of the **FileHeader** type.

Node Records

Each node record is a fixed-length disk record with a node number at the front of the record followed by anything PARODY wants to put into the node. The node number is a thread pointer. It points to the next node in a logical node thread. Nodes are implemented as objects of the **Node** type.

Node Threads

A node thread consists of a number of nodes. PARODY maintains a pointer to the first node in the thread. The node pointer at the front of the first node points

to the second node, which points to the third, and so on, until a node is reached that has zero in its node pointer, which marks the end of the thread. The nodes in a thread are in no particular sequence in the file.

Deleting Nodes

When PARODY is done with a node, it tells the **Node** class to delete the node. This action clears the node's data area to zeros and adds it to the deleted node thread, which is pointed to by the deleted node pointer in the header record. The pointer actually points to the most recently deleted node. At first, before any nodes have been deleted, the deleted node pointer is zero. When a node is deleted, it moves the deleted node pointer from the header into its next node pointer and moves its own address into the deleted node pointer in the header.

Allocating Nodes

When PARODY needs a new node, it looks first at the deleted node pointer in the header. If the pointer is nonzero, PARODY appropriates that node for the new usage and writes the next node pointer from the node into the deleted node pointer.

If the deleted node thread has no nodes, PARODY appropriates a node from the end of the file by using the node number of the highest node allocated as recorded in the file header.

Object File

PARODY stores objects in node threads. Each object starts on a node boundary. The node number is the object's address. The first two integers at that address are the class identification of the object and a logical node number, both used in rebuilding the index file, which is described in Chapter 9. The data bytes of the object follow the class identification.

If the object's length is greater than that of a node, the object spills over into the next node in the thread, continuing that way until the entire object is recorded. The balance of the last node in an object's thread is padded with zeros. Figure 8.24 shows the relationship between the index file and the object file.

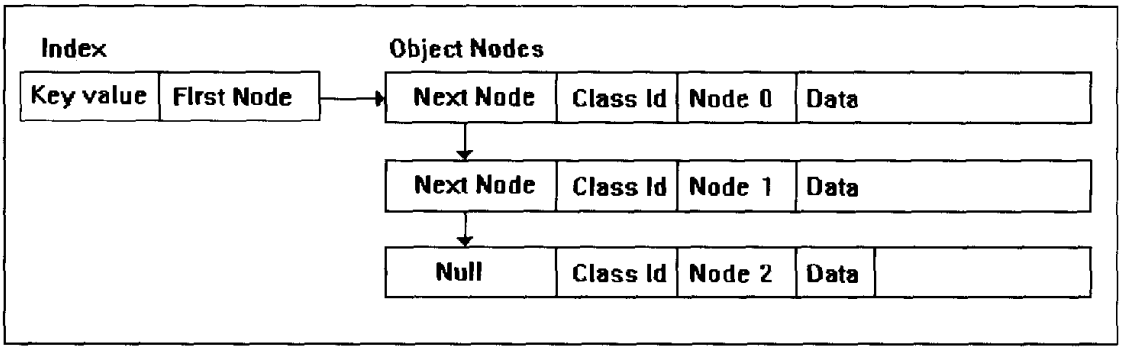

Figure 8.24 *Index and Object File*

The same physical file of nodes holds all the objects of all the classes in a database. Every class can support objects of different sizes and formats, and objects of a class can have variable lengths as well. For example, a class that has a string data type might have objects with many different string lengths. A class with an array can have objects with different numbers of elements in the array.

Without the class definitions and the member functions that support the objects, the object file is meaningless. Nothing that stands alone in the file itself tells the database the types of the objects or anything about their formats. For PARODY to correctly locate an object, it needs the address of the object's first node. That address is maintained by the indexes, described in the following section.

Index File

The database index file contains indexes that associate key values with object node addresses in the object file. An index file holds one logical index table for every primary and secondary key in the persistent classes. The indexes are implementations of the B-tree algorithm described in the next section.

Class Header Tables

Each class is represented by a table of index header records. The first node in the file is the header table for the class that has class identification zero. That node is the first of a thread of index header nodes. The second node in the thread is the table for class identification one, and so on.

Index Header Records

Each class header record is a table of index header records. There is an index header record for each key in the class. The first one is for the primary key, the second one is for the first secondary key, and so on.

Each index header record contains two data items: the node number of the root node for the index and the length in bytes of the index key value. The software assigns the root node number when the index is first built, and it extrapolates the key length from the first time the key class derived from the base **Key** class writes a key index from its **Write** function.

B-Trees

PARODY uses the B-tree data structure to implement the indexes. 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 object node that matches a specified key argument. The tree is itself 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 usually 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 in the heirarchy.

Nodes store keys in key value sequence. Each key consists of the key index value provided by the object and the object's node address in the database provided by PARODY. 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* and do not contain pointers to lower nodes.

Figure 8.25 is an example of a B-tree that uses the letters of the alphabet as keys to locate matching words from the phonetic alphabet. The letters are analogous to the data values for key data elements in a class; the phonetic alphabet words, incomplete in the example, are equivalent to the objects that the B-tree indexes.

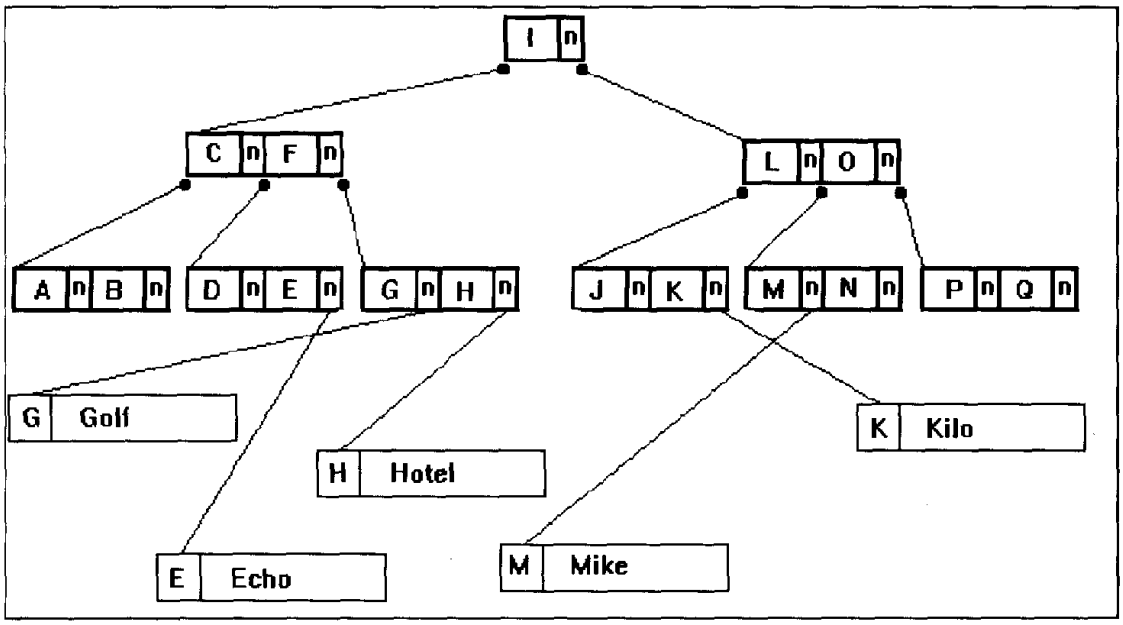

Figure 8.25 *The B-Tree*

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 8.25, 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 will fit.

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.*

▼ Summary

This chapter described PARODY and how you define a PARODY database. It discussed how PARODY implements your database in cooperation with the classes you design, and it provided some details about the internal file structures of a PARODY database.

Chapter 9 is about how an application uses a PARODY database to create, modify, and delete objects.

▼ References

Advanced C++ Programming Styles and Idioms, James O. Coplien, 1992, Reading, MA, Addison-Wesley

Fundamentals of Data Structures, Horowitz and Sahni, 1976, Rockville, MD: Computer Science Press, Inc.

The Art of Computer Programming, Volume 3, Donald E. Knuth, 1973, Reading, MA: Addison-Wesley.

Chapter 9

Using a PARODY Database

"An object in possession seldom retains the same charm that it had in pursuit."

Pliny the Younger

This chapter is about how programs use the object database. Chapter 8 describes how you define the database by designing properly inherited classes. In this chapter you will learn how to declare persistent objects, manipulate them, and tell PARODY what it should do about them.

▼ Building a Database

You build a new PARODY database the first time you declare an object of the **Parody** type. The **String** name that you give the database becomes the file name for the object data and index files with file extensions ".DAT" and ".NDX" assigned when they are created.

The first and subsequent declarations of the same **Parody** object open the database for use by your program. The database remains open as long as the **Parody** object is in scope. You need to make the address of the object known globally so that the constructors for the persistent object classes can associate themselves with the correct database.

You can have more than one database open at the same time.

▼ Managing Persistent Objects

With an open PARODY database you can add persistent objects, retrieve them, change them, and delete them. PARODY will maintain the integrity of the interclass relationships, and you can monitor its actions in this regard. You can navigate among the objects of a class and among related objects of different classes. The examples that follow assume that you are using the **Employee** class as shown in figure 9.1, which assumes that the **emplno** and **deptno** data members are objects of abstract data types derived from the **Key** base class and, therefore, the primary and secondary key of the class.

```
class Employee : public Persistent {
    EmployeeNo emplno;
    DepartmentNo deptno;
    // ...
public:
    Employee(int en = 0);
    ~Employee();
    DepartmentNo& DeptKey() { return deptno; }
    SetDeptNo(int dpno);
    // ...
};
```

Figure 9.1 *An Example Persistent Class*

Retrieving Persistent Objects

You retrieve a persistent object by declaring one that exists in the database. Figure 9.2 shows this procedure.

```
void f()
{
    Employee empl(123);
    if (empl.ObjectExists())   {
        // --- the object exists
        // ...
    }
}
```

Figure 9.2 *Retrieving a Persistent Object*

The **empl** object in Figure 9.2 is constructed from the 123 primary key value, which is the employee number in this example. The **ObjectExists** function returns a true value if PARODY found a matching object on the database.

Creating Persistent Objects

You create a persistent object by first declaring one that does not already exist on the database and then telling PARODY to add it to the database. Figure 9.3 shows this procedure.

```
void f()
{
    Employee empl(123);
    if (!empl.ObjectExists())   {
        // --- the object does not exist
        // ...
        empl.AddObject();
    }
}
```

Figure 9.3 *Adding a Persistent Object*

The **empl** object in Figure 9.3 is constructed from the 123 primary key value, which is the employee number in this example. This time, the PARODY does not find a matching object, so the program calls the **AddObject** function to tell PARODY to add the object to the database.

Changing Persistent Objects

To change a persistent object, declare one that exists, change its data values, and call the **ChangeObject** function. Figure 9.4 is an example of that process.

```
void f()
{
    Employee empl(123);
    if (empl.ObjectExists())   {
        // --- the object exists
        // ... (change the data members) ...
        empl.ChangeObject();
    }
}
```

Figure 9.4 *Changing a Persistent Object*

Deleting Persistent Objects

To delete a persistent object, declare one that exists and call the **DeleteObject** function. Figure 9.5 is an example of that process.

```
void f()
{
    Employee empl(123);
    if (empl.ObjectExists())   {
        // --- the object exists
        // ...
        empl.DeleteObject();
    }
}
```

Figure 9.5 *Deleting a Persistent Object*

The Effect of Adds, Changes, and Deletes

The objects in Figures 9.3 through 9.5 may continue to process after receiving the **AddObject**, **ChangeObject**, or **DeleteObject** messages. The messages tell them to prepare to be added, changed, or deleted. The addition, change, or deletion occurs from the object's destructor when it calls **SaveObject**.

Integrity

The **AddObject**, **ChangeObject**, and **DeleteObject** functions return a true value if they will permit the action and a false value if they will not. The functions will not permit the action if it would violate class integrity. **AddObject** and **ChangeObject** will reject an object that attempts to relate itself to another object that does not exist on the database. **DeleteObject** will reject the deletion of an object if other objects in the database are still related to it. Figure 9.6 shows how those functions work.

```
if (!empl.AddObject())
    // --- add was rejected

if (!empl.ChangeObject())
    // --- change was rejected

if (!empl.DeleteObject())
    // --- delete was rejected
```

Figure 9.6 *Maintaining Object Integrity*

These return values reflect the condition of the database when the functions execute. Nothing prevents you from further changes to the objects after the integrity tests are passed. Therefore, these function calls should be the last thing you do to the object before it goes out of scope.

▼ Navigating the Database

You can navigate a database two ways: by associative access, which uses key index values to address objects, or by navigational access, which uses the addresses of the objects. The first method is relational and the second reflects a more object-oriented approach.

Navigating by Key: Content-Addressable Objects

The relational data model uses key index values to locate objects. PARODY, while not a relational database management system, supports this technique. You have already seen that a program can retrieve a persistent object by instantiating it with a primary key value. You can find objects by using the values of the secondary keys as well. This facility permits the kind of content-addressable access normally associated with relational databases.

FindObject

The **FindObject** function locates an object by using the value assigned by the program to a secondary key. Figure 9.7 shows how that works.

```
Employee empl;
empl.SetDeptNo(1); // initialize the dept#
empl.DeptKey().FindObject();
if (empl.ObjectExists())  {
    // --- at least one object exists
    // ...
}
```

Figure 9.7 *Finding an Object with a Secondary Key*

In this example the program instantiates an **Employee** object with no specified primary key value. The default constructor for the class builds the object with a null key value, and PARODY does not, therefore, try to find a matching object in the database. Thus the importance of the **isNullValue** function described in Chapter 8. The **empl** object has no meaningful data member values yet. The

program assigns one data value, the department number, which in this example, is a secondary key. The program now tells the key object to find the persistent object by using its initialized value. The **FindObject** function does the retrieval.

Observe that you call the function through the key object. If you want to call the function through the persistent object itself, use the format in Figure 9.8.

```
empl.FindObject((Key *)empl.DeptKey());
```

Figure 9.8 *Calling FindObject*

The same convention applies to the navigating functions that follow.

FirstObject and LastObject

You can retrieve the first or last object in a primary or secondary key sequence by using the **FirstObject** and **LastObject** functions. These functions do not require that you initialize the key data value. They retrieve the objects from the front or back of the key sequence. Figure 9.9 is an example of using these functions.

```
Employee empl;
empl.DeptKey().FirstObject();
if (empl.ObjectExists())   {
    // --- the object exists
    // ...
}
// ....
empl.DeptKey().LastObject();
if (empl.ObjectExists())   {
    // --- the object exists
    // ...
}
```

Figure 9.9 *Finding the First and Last Object of a Class*

If you call the **FirstObject** and **LastObject** functions through the persistent object itself without passing a reference to a secondary key, PARODY finds the first or last object in the sequence of the primary key. Figure 9.10 illustrates this usage.

```
Employee empl;
empl.FirstObject();
if (empl.ObjectExists())   {
    // --- the object exists
    // ...
}
// ....
empl.LastObject();
if (empl.ObjectExists())   {
    // --- the object exists
    // ...
}
```

Figure 9.10 First and Last Object by Primary Key

The same convention applies to **NextObject** and **PreviousObject** described in the next section.

NextObject and PreviousObject

The **NextObject** and **PreviousObject** functions return the next and previous sequential objects in the sequence of the key through which they are called, or through the primary key if they are called through the persistent object itself. These functions assume that either themselves, **FirstObject**, **LastObject**, or **FindObject** has positioned the key at an object. If not, **NextObject** works just like **FirstObject** and **PreviousObject** works just like **LastObject**. Figure 9.11 illustrates their use.

```
// ----- navigate forward
Employee empl;
empl.DeptKey().FirstObject();
while (empl.ObjectExists())   {
    // --- process the object
    empl.DeptKey().NextObject();
    // ...
}
// ----- navigate backward
empl.DeptKey().LastObject();
while (empl.ObjectExists())   {
    // --- process the object
    empl.DeptKey().PreviousObject();
    // ...
}
```

Figure 9.11 Navigating Forward and Backward

Navigating Among Classes

Sometimes a program must declare a persistent object and then retrieve, one by one, its related objects of another class. For example, if you wanted to report the employees assigned to a particular department, you could use code similar to that shown in Figure 9.12.

```
Department dept(321);
if (dept.ObjectExists())  {
    Employee empl;
    empl.SetDeptNo(321);
    empl.DeptKey().FindObject();
    while (empl.ObjectExists() &&
            empl.DeptNo()  == dept.DeptNo())  {
        // ...
        empl.DeptKey().NextObject();
    }
}
```

Figure 9.12 Navigating Among Classes

The program in Figure 9.12 instantiates a **Department** object that has 321 as its primary key value. Next it instantiates an empty **Employee** object, sets the same department number into that object's secondary key, and calls **FindObject** through the key. The program processes that employee object and subsequent employee objects retrieved by the **NextObject** function until either the department number in the next employee object changes or the **ObjectExists** function returns a false value indicating the end of the **Employee** class's department number key sequence.

Navigating by Address: Position-Addressable Objects

Having stressed until now the strength of the relational approach to database navigation, this book acknowledges in this chapter that there are times when keyed indexes are not the best solution for database access and navigation. Sometimes you just want to retrieve a persistent object by using some kind of handle that the database manager provides when you create the object. When might that be?

Perhaps you are using the object database to retain some number of single objects between invocations of a program or for passing the object from program to program. For example, a CAD/CAM database might contain one object each of several classes, among them, the design model, some number of canned features, and perhaps some embedded text strings. The application does not need to retrieve objects from a collection on the basis of data values. It needs to retrieve the one and only object or one of a very few objects of a class. Most likely, you will know if this data model applies to your application. As a general rule, if the database will store a large number of objects of the same class, the objects will be indexed by a key value. If the database stores a small number of objects of each class, small enough that the application does need to address them by unique data values, then the objects may be keyless.

PARODY supports the design of a database without keys and the retrieval of objects without indexing.

Designing a Class Without Keys

To design a class without keys, simply leave out the keys. If no data member is derived from the **Key** class, then the objects are *keyless*, and you must retrieve them by specifying their logical PARODY addresses. The class will still require the **Read** and **Write** functions, the constructor must call **LoadObject**, and the destructor must call **SaveObject**.

The keyless class must include a constructor that allows it to be declared with no data values, and one that allows it to be declared with an **ObjAddr** parameter, a type that specifies the object's logical PARODY address. Both constructor requirements may be covered by one constructor that uses a default parameter value to set the **ObjAddr** parameter to zero. The constructor's call to **LoadObject** includes the **ObjAddr** parameter's value as a parameter. Figure 9.13 is a simple keyless class that stores a text string for an application.

Creating a Keyless Object

You create a keyless object by instantiating it with no initializers, assigning some values to its data members and calling **AddObject** just as you do with keyed objects. There are no integrity checks with keyless objects because there are no keys to relate classes.

```
class Text : public Persistent
{
    String text;
    void Read()  { ReadObject(text); }
    void Write() { WriteObject(text); }
public:
    Text(ObjAddr objaddr = 0) : Persistent(*db, TEXT)
        { LoadObject(objaddr); }
    ~Text() { SaveObject(); }
    SetText(String& txt) { text = txt; }
};
```

Figure 9.13 *A Keyless Class*

After the call to **AddObject** you must get the new persistent object's logical PARODY address. The **ObjectAddress** function returns the address of an object once it has been added to the database. It is the responsibility of the application to remember the address for subsequent retrievals of the object. Figure 9.14 creates an object of the **Text** class;

```
void f()
{
    Text txt;
    txt.SetText("Birth of the Cool");
    txt.AddObject();
    ObjAddr objadr = txt.ObjectAddress();
}
```

Figure 9.14 *Creating a Keyless Object*

Retrieving by Object Address

To retrieve a keyless object, you instantiate it with its object address as a parameter. Figure 9.15 is an example of retrieving an object by its address.

```
void f(ObjAddr objadr)
{
    Text txt(objadr);
    // ...
}
```

Figure 9.15 *Creating a Keyless Object*

The example in Figure 9.15 assumes that the caller passes the correct object address to the function, which declares an object from the address. If PARODY determines that the address does not point to an object of that class, it will not call the class's **Read** function, and **ObjectExists**, if you call it, will return false. This condition would probably be a program bug rather than a user error, because the technique assumes that the program can always know the correct addresses for the objects it retrieves.

Navigating by Object Address

You may use the **FirstObject**, **LastObject**, **NextObject**, and **PreviousObject** functions for a keyless class, calling them through the class's persistent object. These functions will navigate the class in the coincidental physical sequence of the objects of the class in the database. This sequence is not necessarily the sequence in which the objects were created. The algorithm that allocates space in the database for new objects reuses deleted object space, so an object that you created yesterday might well be found after one that you create today.

This form of navigation can be inefficient in a large database. PARODY maintains no pointers to or chains of keyless objects. When you call **FirstObject**, for example, PARODY scans the database from the beginning, looking at every node to find the first node of the first object of the class. The other scans work similarly, scanning from the last node of the file or from the address of the object through which you call the function.

Updating and Deleting Keyless Objects

Updating and deleting keyless objects involve retrieving the object to be changed or deleted, changing the data members for a change and calling **ChangeObject** or **DeleteObject** just as you do with keyed objects. There are no integrity checks other than to make sure that the object address points to an object of the class being changed or deleted.

Embedding Keyless Persistent Objects

Embedded keyless objects work very much like the embedded keyed objects described in Chapter 8, except that the outer object uses the object address rather than a key to identify and declare the inner object. This means that the outer object must store the inner object's address from the **Write** function and

rebuild the inner object from the **Read** function. Figure 9.16 is an example of a keyless class that includes a keyless embedded persistent object, in this case an object of the **Text** class shown in Figure 9.13.

```
class Manuscript : public Persistent   {
    Date prepared;
    Text text;  // an embedded keyless persistent object
    void Read();
    void Write();
public:
    Manuscript(Date dp, ObjAddr txobjadr) :
            Persistent(*db, MANUSCRIPT), text(txobjadr)
        { prepared = dp, LoadObject(); }
    Manuscript(ObjAddr objaddr = 0) :
                    Persistent(*db, MANUSCRIPT)
        { LoadObject(objaddr); }
    ~Manuscript() { SaveObject(); }
    // ...
};

void Manuscript::Read()
{
    ReadObject(prepared);
    ObjAddr txtobjaddr;
    ReadObject(txtobjaddr);
    text.LoadObject(txtobjaddr);
}

void Manuscript::Write()
{
    WriteObject(prepared);
    WriteObject(text.ObjectAddress());
}
```

Figure 9.16 *Embedded Keyless Object*

The **Write** function writes the object's data members and the object address of the embedded **Text** object.

The **Read** function calls **ReadObject** for the object's data members and then calls **ReadObject** to read the object address of the **Text** embedded object, which was instantiated as an empty object by the **Manuscript** constructor's

implied call to the **Text** default constructor. Then the **Read** function completes the instantiation of the embedded object by calling **LoadObject** through the empty **Text** object, passing the stored object address.

How does the first instantiation of an embedded object get into the outer object? The embedded object must exist as an established persistent object with an object address before you can create the outer object, and this is where the first constructor in the **Manuscript** class comes in. You declare a new outer object as shown in Figure 9.17.

```
void f()
{
    // ---- build a persistent Text object
    Text txt;
    txt.SetText("Birth of the Cool");
    txt.AddObject();
    // ---- get the text object's address
    ObjAddr txobjadr = txt.ObjectAddress();
    // ---- instantiate a date
    Date dp(6,29,90);
    // ---- build the manuscript with the date
    //      and the embedded text object
    Manuscript man(dp, txobjadr);
}
```

Figure 9.17 *Declaring an Object with an Embedded Object*

Why not simply assign the embedded text object to the persistent manuscript object after they are both instantiated? That approach is possible, but only if the text object uses the handle/copy idiom for multiple copies of the same object.

▼ Converting PARODY Databases

A database designed, implemented, and in use is not necessarily a fixed entity. Requirements for the application will change, and you will, from time to time, need to modify the format of an existing database. Such modifications imply

a conversion of the existing database to the new format. There are several things to consider.

Adding Classes

If you are adding one or more classes to the database, your conversion is simple. Use the next available class identification code for the new class and make no physical conversion at all. The database will accept the new classes without modification.

Removing Classes

To remove a class, you may delete all the objects and retire the class identification code. You do not need to make a physical conversion of the database file, and you can reuse the class identification code if you add a new class later.

Adding Data Items to an Existing Class

You can add data items to an existing class without converting the database if the following conditions are true. First, you must add the data items at the end of the series of calls to **ReadObject** and **WriteObject** from your **Read** and **Write** functions. Second, your constructor should initialize the new data items to zero values before it calls **ReadObject**. Third, the new data items should behave appropriately when they have zero values.

The **ReadObject** function will not read past the last node of an object. The last node of an object is padded with zeros. Therefore, if you initialize the additional data items with zeros, and **ReadObject** either reads padding bytes into the data item or reads nothing because there are no more nodes in the object's node string, then the data item will contain zeros after the persistent object is instantiated.

The **WriteObject** function will allow new or changed objects to grow to their proper lengths.

Removing Data Items from an Existing Class

If you remove a data item from an existing class, you should convert the database as described further down in this section. You can avoid that conversion if the **Read** and **Write** functions continue to allow for the data item's space by reading and writing to and from unused variable space. In practice, however, you will find that such workarounds are inefficient and cloud the design. Residual data values will remain in the database in old objects, and new or changed objects will have useless holes. It's better to do a proper job and convert the database.

Adding Keys to an Existing Class

If you are changing an existing data item into a key, you might not need to convert the database as long as the data item's physical size and proximity to other data items in the class do not change. Replace the data item in the class definition with a data item that derives from the **Key** class, and provide the associated supporting key functions. Then rebuild the indexes for the database as described in a later section.

If you are adding a completely new key data item, use the procedures for adding any data item, provide the support that the **Key** base class requires, and rebuild the indexes for the database.

Converting the Database

The changes you make will most likely affect only a small part of the database, perhaps only one class. You need to convert that class only, and you can do it within the database itself. Leave the original class intact and use a dummy name for the replacement class. For example, if you are changing the **Department** class, you might add the new class with the name **DepartmentNEW**. The conversion consists of the following six steps.

Step 1

Assign a unique class identification code to the **DepartmentNEW** class. This code will remain with the class when it eventually replaces the the **Depart-**

ment class. You will retire and perhaps someday reuse the class identification code that the original **Department** class uses.

Step 2

Design the **DepartmentNEW** class to reflect the changes you are making. Add the class design to the database schema. Leave the original **Department** class design in the schema.

Step 3

Write a conversion constructor to convert the **Department** class to the **DepartmentNEW** class.

Step 4

Write and run a conversion program that reads all the **Department** objects, converts them to **DepartmentNEW** objects, and deletes the **Department** objects. Figure 9.18 is an example of such a conversion program.

```
void main()
{
    // --- scan all the Department objects
    for (;;)   {
        Department dept;
        dept.FirstObject();
        if (!dept.ObjectExists())
            break;
        // --- convert the Department object
        //     to the DepartmentNEW object
        DepartmentNEW deptnew(dept);
        deptnew.AddObject();
        // --- delete the Department object
        dept.DeleteObject();
    }
}
```

Figure 9.18 A Class Conversion Program

Step 5

Rename the **DepartmentNEW** class to **Department** and use it to replace the old **Department** class definition. Change the class identification code for the **Department** class to the one that the **DepartmentNEW** class used during the conversion.

Step 6

Make any modifications to the application that the changed database requires. Compile and link the entire application with the new class definition.

▼ Rebuilding the Index File

The weakest link in a relational database is the integrity between the indexes and the data files. If a system error causes an index to point somewhere other than to the object it indexes, the database is in a state of disintegration and is unstable if not unusable. In other errors, the integrity of the index file itself can be disturbed. If the node pointers in the B-tree file do not point to correct nodes, for example, the index is unusable.

Database Integrity Errors

These errors usually occur when the system fails while the indexes and objects are being changed. Until the changes are completed, the system is unstable. The failure causes the instability to remain when you restart the system. The symptom of such instability is the inability of the application to retrieve valid and logical objects. Therefore, the first measure that a database user should try when those symptoms occur is the rebuilding of the indexes. You should provide such a program for each database that you build. As you learned in the previous section, a database conversion task can also need a utility program to rebuild indexes.

The Index Rebuilding Program

Each database needs its own program that rebuilds indexes. The program is not complex, and Figure 9.19 is an example of such a program.

```
#include "personel.h" // application-specific header
Parody *personnel;
void main()
{
    // ---- delete the old index file
    String dbname("PERSONEL");
    remove(dbname+".NDX");
    // ---- instantiate the database
    personnel = new Parody(dbname);
    ObjectHeader objhdr;
    ObjAddr nd = 1;
    // ---- scan all the nodes
    while (nd < personnel->HighestNode())  {
        // --- get the node's object header
        personnel->GetObjectHeader(nd, objhdr);
        // --- look for first node of any object
        if (objhdr.ndnbr  == 0)   {
            // --- rebuild indexes for object
            personnel->RebuildIndexes(nd);
            // --- instantiate the object
            switch (objhdr.classid)       {
                case EMPLOYEE:
                    Employee empl;
                    break;
                case DEPARTMENT:
                    Department dept;
                    break;
                case PROJECT:
                    Project proj;
                    break;
                case ASSIGNMENT:
                    Assignment assgn;
                    break;
                default:
                    break;
            }
        }
        nd++;
    }
    delete personnel;
}
```

Figure 9.19 *An Index Rebuilding Program*

The example conversion program in Figure 9.19 rebuilds the indexes for a database that contains objects of four classes. It scans all the nodes from number 1 to the highest node used. It reads the object header to see if the node is the first node of an object. If so, the program calls the **RebuildIndexes** function to tell the database to rebuild the indexes to point to the current node for the next persistent object instantiated. Then the program instantiates an object of whatever type the object header indicates. PARODY and the object read the data values from the database through the **Read** and **ReadObject** functions invoked by the object's constructor. PARODY does not try to find the object from its index value because the program told it that it was rebuilding indexes. Instead, it uses the node number that the program passed to the **RebuildIndexes** function to position the database to read the data values. When the object goes out of scope, which it does right away, PARODY rebuilds the object's indexes from the data values in the keys.

▼ Summary

This chapter taught you how to use PARODY's persistent objects, how to perform database conversion, and how to rebuild database indexes. Chapter 10 presents three case studies that are examples of PARODY applications. It also introduces some utility classes that you can use as data types to build data members in your persistent objects.

Chapter 10

Case Studies

"Few things are harder to put up with than the annoyance of a good example."

Mark Twain

This chapter presents two small working applications that use PARODY. You can use these case studies to learn more about how PARODY works and you can use them as examples when you develop your own applications. The first case study uses keyless objects to implement the classic simulation, the *Game of Life*. The second case study, a personnel management system, uses the almost-relational properties of PARODY to implement a personnel management system that maintains indexed objects for departments, projects, employees, and assignments. You will recognize much of the personnel application from the examples in earlier chapters.

Before discussing the example applications, this chapter addresses some supporting software. PARODY uses general-purpose string and linked list classes that you might find useful in your own applications, and there is **GUI**, the generic user interface class that the applications use.

The source code for these classes and the example applications is in Appendix C. The discussions refer to them by their source code file names, and each discussion begins with a list of the files discussed. A table of contents at the beginning of the appendix tells you where to find each of the listings.

▼ Utility Classes

Source files discussed:

```
linklist.cpp
linklist.h
strings.cpp
strings.h
```

Most C++ compilers come equipped with a class library that includes concrete data types that you can incorporate into your classes and template and/or base abstract classes from which you can construct containers and lists of your own types. There is no standard, however, and so each language product provides its own variants. Furthermore, software developers who used C++ when there were no class libraries have often developed their own versions of these classes and have significant numbers of existing programs that use them.

Rather than bind itself to a particular C++ language environment, PARODY provides a **String** concrete class and implements a single-direction linked list with a **LinkedListHead** type and an associated abstract **LinkedListEntry** class.

String

Virtually every new C++ programmer builds a string class, particularly if they advance to C++ from C and were, at one time, a BASIC programmer. BASIC has a powerful string type that C lacks, and C programmers miss it. No wonder they take advantage of the C++ class system to add strings to the language.

Most compilers include a string class in their class libraries, but not all of them implement the same features or interface. Eventually the ANSI X3J16 committee might add a standard string class to the language definition. Until they do, C++ programmers must use what is at hand. This book includes a string class so that readers can use PARODY on a variety of compilers.

The strings.h and strings.cpp source files in Appendix C implement the **String** class. You can declare a string by specifying a **char** pointer, another **String** object, or a length and a fill character. If you leave out the fill character, the default is zero. You can declare a null string as well by using the default constructor.

You can compare strings by using the overloaded relational operators. The overloaded plus (+) operator concatenates strings. There are **right**, **left**, and **mid** member functions to build substrings from existing string objects.

The example applications in this chapter use the **String** class. Figure 10.1 illustrates its use.

Linked List

PARODY uses a linked list class in several places. A linked list is a data structure that links objects in a list that the program can navigate. A linked list consists of a *list head* that points to the first and last objects in the list and *list entries* that are themselves the listed objects. Each list entry includes its own data values, a pointer to the list head, and pointers to the next and previous entries in the list. A program can append entries to the list, insert an entry at a specified position relative to another entry, delete entries, and navigate the list forward and backward starting at either end entry or at a chosen entry in the middle of the list. Figure 10.1 illustrates a typical linked list.

```
// ---- constructors
String name1;                // null string
String name2("Clifford"); // construct with char*
String name3(name2);         // construct with String
String name4(12);             // construct 12-byte,
                             // zero-filled String
String name4(10, '*');        // construct 10-byte,
                              // character-filled String
// ---- assignment
name1 = name2;
// ---- relational
if (name1 == name2)
    // ...
if (name1 != name2)
    // ...
if (name1 < name2)
    // ...
if (name1 > name2)
    // ...
if (name1 <= name2)
    // ...
if (name1 >= name2)
    // ...
// ---- concatenate
name1 = name2 + name3;
name1 += name2
// ---- substrings
name1 = name2.right(4);  // build "ford"
name1 = name2.left(5);   // build "Cliff"
name1 = name2.mid(4, 2)  // build "iffo"
// ---- subscripts
char c = name2[2];       // returns 'i'
name2[2] = 'a';          // build "Clafford"
// ---- length
int ln = name1.StrLen(); // length of string
int ln = name1.Length(); // length of string buffer
```

Figure 10.1 *Uses of the String Class*

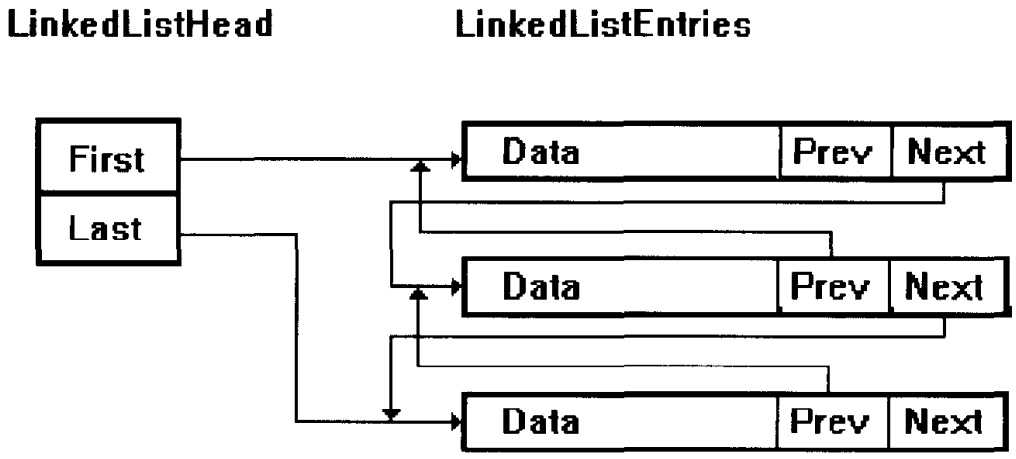

Figure 10.2 *A Typical Linked List*

The linked list that PARODY uses involves two class definitions. The first, **LinkedListHead**, builds the list head. Another class may assume the role of the list head by inheriting from the **LinkedListHead** class, or it may simply include an object of the **LinkedListHead** type as a data member.

The second class is the **LinkedListEntry** abstract base class. Objects that are going to be entries will derive from this class. They may specify the **LinkedListHead** object with which they are associated as a parameter to their constructor or as a parameter when they are added to a list. An object may be a member of only one list at a time.

The **LinkedListEntry** class includes the member functions listed in Figure 10.3.

```
void AppendListEntry(LinkedListHead *lh = NULL);
void PrependListEntry(LinkedListHead *lh = NULL);
void InsertListEntry(void *fe,LinkedListHead *lh=NULL);
void DeleteListEntry();
void *NextListEntry();
void *PrevListEntry();
```

Figure 10.3 *The LinkedListEntry Member Functions*

An object of a class derived from **LinkedListEntry** can use these functions to append itself to the list, insert itself at the front of the list or after any specified other entry, delete itself from the list, and retrieve a pointer to the next or previous entry in the list. If the first three function calls do not include the address of the associated **LinkedListHead** object, then the entry's constructor must have specified the list head object.

Appendix B includes a reference guide to the linked list classes.

▼ GUI: The Generic User Interface

Source files discussed:

```
gui.cpp
gui.h
```

Every software development activity targets a user environment and must, therefore, code to a particular user interface model. The programs in this chapter are no exception. They are launched from the ubiquitous command-line interface, and they use a simple text-mode screen model to display information. To encapsulate the console functions, this book includes the **GUI** class, which manages menus, data and error message displays, and keyboard input through a combination of ANSI protocols and PC-specific BIOS calls.

Do not expect to use the **GUI** class outside of the example applications in this chapter. Most programmers will write programs for far more sophisticated user interface environments. The **GUI** class provides support for the example programs at the lowest common console denominator. If you use something other than a PC-compatible computer with MS-DOS, you must modify the **GUI** class.

The Display

To run the progams without modification to the screen display software, you need an ANSI-compatible terminal or a software driver that emulates the ANSI protocols for cursor positioning and clearing the screen. If you are using

an MS-DOS computer to run these programs, make sure that the CONFIG.SYS file in the root directory of the boot disk contains this statement:

```
DEVICE=ANSI.SYS
```

It might be necessary to add a DOS path to the file name.

The Keyboard

The **GUI** class uses functions from MS-DOS C++ compilers to test for keyboard input and read the keyboard. The standard **cin** object does not satisfactorily provide for single key input, which menu selections use. The object has no way to test for the presence of a keystroke without actually reading it, and the sample applications require such a feature. Furthermore, there is no consensus for the implementation standards of iostreams among C++ compiler vendors. They all behave differently in small ways. The **GUI** class uses iostreams, but not for everything. Instead, it uses function calls into the BIOS of the PC.

The GUI Object

The gui.h and gui.cpp source files define and implement the **GUI** class. There is a global object of that class named **gui**. The class consists mostly of methods that manage keyboard input and screen output. The header file begins by defining some constant values to establish screen dimensions, where to position the menus on the screen, the maximum number of commands on a menu, and the values returned for certain function keys. If you use function key values other than those defined in gui.h, you should add to that list. The discussion on the **GetKBChar** member function explains that process.

The **GUI** class includes functions to manage a standard menu format, data input, two standard dialogs, output to the screen, and keyboard operations.

Menus

An application invokes a menu by calling the **UserMenu** function with a menu title as the first parameter and a NULL-terminated, variable-length list of menu labels and function addresses. Figure 10.4 illustrates such a menu call.

```
void AddEmployee(); // declare menu selection functions
// etc...

gui.UserMenu( "Employees",
              "Add",      AddEmployee,
              "Change",   ChangeEmployee,
              "Delete",   DeleteEmployee,
              "List",     ListEmployees,
              "Compare",  CompareEmployees,
              "Projects", ListEmployeeProjects,
              NULL );
```

Figure 10.4 *Executing a GUI Menu*

The function call in Figure 10.4 will clear the screen and display the menu shown in Figure 10.5.

```
Employees

  1: Add
  2: Change
  3: Delete
  4: List
  5: Compare
  6: Projects
  9: Return

Select>
```

Figure 10.5 *A GUI Menu Displayed on the Screen*

The user selects a menu command by typing its number followed by the Enter key. The program executes the function associated with the command. The function can determine which menu command is executed by calling the **GUI** object's **Selection** member function, which returns the integer value that the user selected.

When the command function returns, the **UserMenu** function clears the screen and displays the menu again. If the command function displays information that the user must see before returning to the menu, the command

function should call the **GUI** object's **AnyKey** function to prompt the user to press a key before proceeding.

You may nest calls to **UserMenu**, but the value returned by the **Selection** member function is valid only before the nested call.

Data Input

The **GUI** class includes several overloaded **UserInput** functions that prompt the user to enter a data value and then read the value into a specified object. Figure 10.6 shows the use of these functions.

```
// ----- character input
char c;
UserInput(c, "Enter a letter");
// ----- integer input
int i;
UserInput(i, "Enter a number");
// ----- string input
String name;
UserInput(name, "Enter your name ", 25);
```

Figure 10.6 *Calling the UserInput Function*

The **UserInput** call to load a **String** object must specify the length of the string to be read. The object's length will be adjusted to reflect the specified length.

Dialogs

The **GUI** class includes three simple dialogs that the applications use. The **YesNo** function displays a question, reads the user's answer, and returns True if the user answered yes and False if the user answered no. The **Error** function displays a specified error message along with an audible alarm and prompts the user to enter any key to continue. The **AnyKey** function is the dialog that prompts the user to enter any key. Figure 10.7 shows the use of these three dialogs.

```
if (!gui.YesNo("Is that OK?"))
    gui.Error("Sorry");
else
    gui.AnyKey();
```

Figure 10.7 *User Dialogs*

Screen Output

The **GUI** class includes four functions to assist the application program with screen output. The **SetCursor** function positions the screen and keyboard cursor at a specified x/y coordinate. The **ClearScreen** function clears the screen. The **WriteChar** function writes a single character at a specified x/y coordinate. The **StatusLine** function writes a line of text at the bottom of the screen, padding the line with spaces. Figure 10.8 shows the use of these functions.

```
gui.ClearScreen();          // clear the screen
gui.SetCursor(5,10);        // position cursor:x=5,y=10
cout << "Hello, Dolly";     // written at 5/10
gui.WriteChar('!',17,10);   // punctuate the message
gui.StatusLine("All done"); // write a status line
```

Figure 10.8 *Screen Output*

Keyboard Input

The **GUI** class includes three functions to assist the application program with keyboard input. The **KBCharWaiting** function returns True if there is a keystroke waiting to be read. The **GetKBChar** function waits for a keystroke, reads it, and returns its value. The **PutBack** function stuffs a keystroke into the input buffer so that it will be returned by the next **GetKBChar** call. Figure 10.9 shows the operation of the keyboard functions.

```
if (gui.KBCharWaiting())
  c = gui.GetKBChar();
gui.PutBack(ESC);
```

Figure 10.9 *Keyboard Input*

When the user presses a function key, the **GetKBChar** function returns the key's ASCII scan value logically OR'd with the value 0x80. The constant integers in gui.h define those values for the PC's cursor arrow keys and the Escape key. If your application needs other function key values, you can add to that list. To determine the values, write a program such as the one in Figure 10.10.

```
// -------- kb.cpp
#include <iostream.h>
#include "gui.h"

void main()
{
    char c;
    cout << "Type keys. Esc when done" << endl;
    while ((c = gui.GetKBChar()) != ESC)
        cout << "That key's value is: "
             << ((int) c & 255) << endl;
}
```

Figure 10.10 *Finding the Key Values*

▼ The Game of Life Application

Source files discussed:

```
life.cpp
```

The Game of Life, used here to demonstrate keyless persistent objects, dates to the early days of computer simulation. It is small enough to lend itself well to an illustration of persistent objects. The game involves relatively few lines of code, and the condition of its simulation at any time is an object that can be stored and retrieved in an object database.

The Concept of Life

Life simulates a universe of neighboring cells. Each cell, identified by its x/y address in the screen coordinate system, may have one of two possible states. The cell is either populated or unpopulated and is surrounded by eight neighboring cells. The game consists of a sequence of generations. Each generation examines each cell to see (1) if it is populated or not and (2) how many neighbors it has. A neighbor is one of the eight adjacent cells in the 3 x 3 array of nine in which the target cell is the center cell. The game's principle

is that if an unpopulated cell has a certain number of populated neighbors, the cell will become populated itself in the next generation. Conversely, if a populated cell has too few or too many neighbors, its population expires in the next generation. The underlying concepts that drive these births and deaths are that (1) a cell is born if there are enough neighbors to spawn it and (2) a cell dies if there are either too few neighbors to support it or too many neighbors with which it must share resources. The implementation of Life in this book uses screen character positions to represent cells, and the universe of cells is limited to the number of screen character positions.

An Evolution of Generations

The Game of Life simulates the evolution of generations. To play the game, you create the universe by specifying which cells are initially populated. Then you run the evolution and observe how each generation modifies the pattern of populated cells. You can run the evolution continuously or by stepping through it a generation at a time. The universe often takes on interesting symetrical patterns as the generations pass. Some patterns result in a totally expired universe after a few generations. Some patterns result in a stable universe. Some patterns endlessly repeat a cycle of births and deaths. A culture of Life players blossomed in the 1970s, and they often published and shared interesting starting patterns.

The Life Program

Life .cpp is the source file that implements the game. It uses a keyless object type to save and retrieve patterns. You can design a pattern and save it as a persistent object, or you can stop an evolution midway and save the current pattern. The program tells you the object address of a pattern when you save it. Later, you can retrieve patterns by their object address and play the game by beginning with the retrieved pattern.

The persistent Life object is built from the state of the in-memory Life object when the program adds it to the database. The object in the database consists of a count of populated cells followed by a list of the x/y coordinates of the populated cells. The persistent object is, therefore, a variable-length data structure that does not resemble the in-memory object at all. The **Life::Write** and **Life::Read** functions define the database object format.

Rules for Survival

The in-memory object includes a table of rules for cell survival. You can modify this table to achieve different results during the evolution of a Life session. The **Life::Rules** array consists of true/false state indicators. The outer dimension contains two inner arrays: The first applies to a cell that is unpopulated; the second applies to a cell that is populated. The inner array has one entry for each possible count of neighbors. If the cell has four neighbors, then the fifth entry specifies whether or not that cell will be populated in the next generation. As published, the table implements these rules. An unpopulated cell will evolve (become populated) if it has three populated neighbors. A populated cell will survive (remain populated) only if it has two neighbors. In this model, the ideal neighborhood of nine cells has three adjacent populated cells. You can change this model by modifying the rules table.

Running the Program

When you run the program, it displays the menu shown in Figure 10.11.

```
The Game of Life

  1: Build
  2: Display
  3: Run
  4: Step
  5: Save
  6: Load
  9: Return
Select>
```

Figure 10.11 *Running the Game of Life*

Select 1 to build a new universe. The program displays a mostly blank screen with a status line at the bottom. Use the cursor arrow keys to move the cursor around, and press the space bar or the Enter key to toggle the population of a cell. A populated cell displays as a happy face character. You can change the display character by changing the value of the **Pop** constant in life.cpp.

When the screen has the population you want, press Esc to return to the menu. Selection 2 displays the screen again. Selection 3 runs the evolution of generations nonstop. You can interrupt it by pressing the Esc key. Selection 4 displays the current generation and allows you to step through subsequent generations a step at a time by pressing the S key.

Selection 5 saves the current generation as a persistent object in the PARODY database named LIFE.DAT. It displays the object address assigned to that object. Make a note of the address so that you can retrieve it later. Selection 6 retrieves a generation object from the PARODY database. It prompts you to enter the object address. Selection 9 returns to DOS.

Life as a Persistent Object

To be a keyless persistent object, the **Life** class derives from the **Persistent** class and provides the necessary functions and function calls. There is no key, so the class does not include a member object that is derived from the **Key** base class.

The constructor accepts an object address as a parameter, which defaults to zero if the caller does not provide it. The constructor sets the class's data members to zero and calls the base **Persistent** object's **LoadObject** function passing the object address as a parameter.

The destructor simply calls **SaveObject**.

The **Read** and **Write** member functions call **ReadObject** and **WriteObject** to read and write the number of populated cells as an integer object and then read and write the coordinates of the populated cells. The **Read** function uses the input coordinates to populate the **Life** object. The **Write** function steps through the **Life** object's cells to derive the coordinates to write.

▼ The Personnel Application

Source files discussed:

```
assign.cpp
assign.h
dept.cpp
dept.h
```

```
employee.cpp
employee.h
personel.cpp
personel.h
project.cpp
project.h
```

This section describes the Personnel system, a PARODY application that maintains a database of departments, employees, projects, and employee-to-project assignments. This system uses PARODY's almost-relational properties to maintain indexed objects and interclass relationships. First you will learn about the persistent classes themselves; then you will learn how the Personnel application uses them.

The Department Class

Dept.h and dept.cpp define and implement the **Department** class. The header file includes definitions for the persistent class and its primary key data member, the **DepartmentNo** object. That class's definition builds most of its member functions as inline functions, the exception being the overloaded assignment operator. You will be able to implement most **Key** classes this way.

The **Department** class is typical of a simple persistent class. Besides its **DeptNo** primary key, it includes data members that define the department name and, as a secondary key, the employee number of the department manager. The **EmployeeNo** class is defined in employee.h, which you will see later, so the dept.h file includes it prior to the **Department** class definition.

There are the usual **Read** and **Write** member functions to support persistence and some member functions to read and change the data values, which are also inline functions.

The **Department** class encapsulates certain aspects of the user interface required to retrieve, display, and modify the object. This approach is consistent in the design of the other classes in the system, and it consists of member functions to display the object, read its data member values from the keyboard, and prompt for and retrieve or build instances of the object. The related member functions are **Display**, **Input**, **InputName**, **InputManager**, **TryChange**, **SelectChange**, **Get**, **GetExisting**, and **Header**.

The Employee Class

Employee.h and employee.cpp define and implement the **Employee** class. This class is more complex than the others because it is an example of a class that uses the handle/copy idiom to allow a program to instantiate more than one copy of the same object. The body of the class is defined in the **EmployeeBody** class, which has no public or protected data members and which specified the **Employee** class as a friend. The **Employee** class is derived from the **Handle** class, and it implements the interface to the body class through member functions.

The **EmployeeNo** class defines the primary key. The **EmployeeBody** class has a secondary key as well, the **DepartmentNo** object, which specifies the department to which the employee is assigned. The only other data member is the employee's name. The rest of the class definitions consist of member functions that access and modify data values and that encapsulate the user interface in the same way that the **Department** class does.

The Project Class

Project.h and project.cpp define and implement the **Project** class, which is similar in construction to the **Department** class. It includes a **ProjectNo** primary key and **EmployeeNo** secondary key to identify the project manager. The other two data members are the project name and an integer that represents the hours expended against the project. The member functions include the usual data member access and modification functions as well as one that allows a using program to add to the hours expended. The user interface is encapsulated just as in the **Department** class.

The Assignment Class

Assign.h and assign.cpp implement the **Assignment** class, an example of a relational connector class that forms the many-to-many relationship between objects of the **Employee** class and objects of the **Project** class. The **AssignmentKey** class is the concatenated key that makes the connection. It is derived from the base **Key** class and it contains two member objects that are themselves derived from the base **Key** class. The **Assignment** class itself contains an object of the **AssignmentKey** class to form its primary key. Since

that class includes two **Key** objects, those objects will be secondary keys. It is necessary, however, for the **AssignmentKey** object to preempt the default primary key designation formed by the sequence of **Key** object constructor execution. The constructor for the **Assignment** class does this by calling the **PrimaryKey** function for the **AssignmentKey** object. The rest of the class definition is similar to the other classes with member functions to access and change data values and an encapsulated user interface.

The Personnel Program

Personel.h and personel.cpp implement the Personnel application. The header file defines the class identification codes for the PARODY database, includes the four header files that define the classes, and declares the external pointer to the database.

Personel.cpp implements the application with a series of **GUI** menus that allow you to enter data into the database and display its contents. The first menu has selections for employees, departments, projects, and assignments. Each of these selections allows you to add, change, list, or delete objects of the chosen class as well as some unique processes for each class. The add, change, list, and delete processes are similar in all the classes, so this discussion covers those functions in general and then discusses the unique processes for the classes that illustrate different ways that an application uses PARODY where objects are related.

Adding an Object

This discussion relates to the **AddEmployee**, **AddDepartment**, **AddProject**, and **AddAssignment** functions in personel.cpp.

The add functions for the classes first call the static **Get** function for the class, which prompts the user to enter a value for the primary key—employee number, for example—and then instantiates an object from the key value. The **Get** functions use the **new** operator to instantiate the objects on the heap and return their pointers, which the caller must destroy with the **delete** operator. If the object's **ObjectExists** function returns True, the add function displays the object and an error message saying that the user is trying to add an object that already exists.

Changing an Object

This discussion relates to the **ChangeEmployee**, **ChangeDepartment**, **ChangeProject**, and **ChangeAssignment** functions in personel.cpp.

The change function calls the object's **GetExisting** function, which prompts the user for a primary key and instantiates an object from the key value. **GetExisting** displays an error message if the object does not exist in the database, destroys the empty object, and returns a NULL pointer. Otherwise it displays the object and returns the address of the object, which the caller must destroy later by using the **delete** operator. The change function calls the **AnyKey** function to give the user time to look at the displayed object. Then it calls the object's **SelectChange** function, which prompts for one of the various data fields in the object and accepts the user's input to change the data, which, if PARODY rejects the change due to an integrity violation, displays an error message.

Listing an Object

This discussion relates to the **ListEmployee**, **ListDepartment**, **ListProject**, and **ListAssignment** functions in personel.cpp.

The list function calls **FirstObject** to retrieve the first object for the class. Then, as long as the object's **ObjectExists** function returns True, the list function calls the object's **Display** function and retrieves the next object in its primary key sequence by calling its **NextObject** function.

Deleting an Object

This discussion relates to the **DeleteEmployee**, **DeleteDepartment**, **DeleteProject**, and **DeleteAssignment** functions in personel.cpp.

The delete function uses **GetExisting** the same way that the change functions do. If **GetExisting** returns a valid object pointer, the delete function asks the user to verify the delete in a **GUI::YesNo** dialog. Then the delete function calls **DeleteObject** to delete the object from the database, displaying an error message if PARODY rejects the deletion due to an integrity violation.

Comparing Employees

The process that compares employees is the only place in the Personnel application where the user can instantiate more than one copy of the same

persistent object. The comparison process demonstrates how that works. It calls **GetExisting** twice to allow the user to specify two employee objects to compare. The program simply displays the two objects. The user is not prevented from specifying the same employee object both times, which causes the program to invoke the multiple-copy logic of the handle/copy idiom. This effect is not obvious in the execution of the example program, except by the fact that it works. If you tried the same thing with one of the other classes, which do not use the handle/copy idiom, the program would not successfully instantiate the second copy but, instead, would instantiate an empty object. You can get a clearer picture of this process by using the compiler's debugger to step through the **CompareEmployees** function and the **Employee** constructors when you specify the same employee number twice.

Listing an Employee's Projects and a Project's Employees

The **ListEmployeeProjects** and **ListProjectEmployees** functions illustrate how a program navigates a connector class by using a secondary key component of its concatenated primary key. The function calls **GetExisting** to get an instance of the project or employee, depending on which secondary key you are searching. Then it instantiates an empty **Assignments** object, adding the partial concatenated key value to the object. Next it calls **FindObject** specifying the secondary part of the concatenated key to find the first instance on the database of an **Assignment** object with the specified project or employee number. Then, as long as the object's **ObjectExists** function returns True and its partial key value is equal to the original value entered by the user, the program instantiates an object of the other type in the concatenated key, displays that object, and then calls **NextObject** for the connector class.

▼ Summary

This chapter provides example applications and utility programs for the PARODY developer. Appendix A shows you how to build the software by using one of the MS-DOS C++ compilers. You will find a programmer's reference guide to PARODY in Appendix B and the complete source code for PARODY and the example applications in Appendix C.

Appendix A

Building the Software

This appendix describes how to build the programs in this book by using one of the three MS-DOS C++ compilers. PARODY compiles with Borland C++ 3.1, Microsoft C++ 7.0, and Zortech C++ 3.0.

The procedures in this appendix use the command-line environments of the three products. All three have integrated development environments as well, and you might wish to install PARODY and the example applications into one of them. However, as published here, PARODY uses the command-line MAKE, compiler, librarian, and linker programs that come with the compiler products.

There are four files, other than the source code itself, that build the software. Each compiler product uses its unique MAKEFILE. All three share a fourth file that contains the commands to build the PARODY object library. The listings for the MAKEFILEs and PARODY.BLD, the librarian command file, are at the end of this appendix.

▼ General Instructions

These instructions apply to all the compiler environments. Follow them first and then proceed to the instructions related to your compiler. If there are additional compilers supported after the publication of this book, or if newer versions of the compilers necessitate changes to these procedures, the companion diskette will describe the changes in a file named README.DOC.

Install Your Compiler

To use any of the compilers, first install it according to the vendor's instructions. Make sure that you have set the DOS path to include the subdirectory where you installed the compiler's executable files. Install the necessary SET commands in your AUTOEXEC.BAT file to set any environment variables required by your compiler. If the compiler has its own configuration file, follow the manufacturer's instructions for building it. The symptoms that occur when your compiler is not properly installed usually point to a missing environment variable or configuration file item. The compilers use these controls to tell them where to find the system's header and library files.

Many of the letters I have received from readers about their problems building the code from C Database Development were, in fact, about problems with their installation of the compiler. Before you assume that a problem is with the PARODY code or with these instructions, verify that your compiler will compile something else. Build the ubiquitous HELLO.C program that most C language texts begin with, and make sure that you can compile, link, and run that program. Use the command-line compiler to build the test program. Often, a compiler is configured correctly for the integrated development environment but is not configured to work from the command line.

Install the PARODY Source Code

Make a subdirectory to contain the source code, and copy the associated MAKEFILE and the PARODY.BLD file into that subdirectory. Copy the source code from Appendix C into the same subdirectory.

Log into the subdirectory where you copied the PARODY source code and proceed to the instructions in this appendix that address your particular compiler.

▼ Borland C++

Modify the MAKEFILE.BCC file (Listing A.1) so that the COMPILER macro points to the subdirectory where you installed Borland C++. The most commonly installed subdirectory is shown in Listing A.1 on line 11. The COMPILER macro identifies the directory below which the TLINK program finds the compiler's startup code and libraries. If the macro value is too long, the MAKEFILE might expand to a command line longer than the maximum 128 characters allowed by DOS, and the make process will not work. In this case, use the DOS SUBST command to assign a virtual disk drive to the subdirectory and assign the virtual disk letter to the COMPILER macro. See your DOS manual for an explanation of the SUBST command.

If you wish to build programs for a memory model other than the large model, change the MODEL macro in MAKEFILE.BCC from 'l' to 's' for the small model, 'c' for the compact model, and 'm' for the medium model. Enter the following command at the DOS command line to run the Borland C++ MAKE program and build the software.

```
make /fmakefile.bcc
```

▼ Microsoft C++

Microsoft C++ uses several DOS environment variables. These are: INCLUDE, SYSTEM, LIB, INIT, and HELPFILES. The installation process puts the SET commands into your AUTOEXEC.BAT file if you allow it to. Make sure that the environment variables are correct.

If you wish to build programs for a memory model other than the large model, change the MODEL macro in MAKEFILE.MSC from 'L' to 'S' for the small model, 'C' for the compact model, and 'M' for the medium model.

Enter the following command at the DOS command line to run the Microsoft C++ NMAKE program and build the software.

```
nmake /fmakefile.msc
```

▼ Zortech C++

Zortech C++ uses two DOS environment variables. These are: INCLUDE and LIB. Make sure that these environment variables are correct.

If you wish to build programs for a memory model other than the large model, change the MODEL macro in MAKEFILE.ZTC from 'L' to 'S' for the small model, 'C' for the compact model, and 'M' for the medium model.

Enter the following command at the DOS command line to run the Borland C++ MAKE program and build the software.

```
make -fmakefile.ztc
```

▼ Testing the Software

Observe the screen while the compiler's MAKE utility builds the software. There should be no error messages. There will probably be no warning messages either, but compiler warnings are not consistent, and future versions of compilers might issue warnings where current ones do not.

Files Built by the Make

If everything went smoothly, the following files will be in the subdirectory where you installed the source code:

```
PARODY.LIB
LIFE.EXE
INDEX.EXE
PERSONEL.EXE
```

The PARODY.LIB file is the object library that contains the member functions for the PARODY and utility classes. It does not contain any object modules for the example programs. You can use the PARODY.LIB library file to link your applications.

The LIFE.EXE and PERSONEL.EXE files are the example applications. The INDEX.EXE file is an example of a program that rebuilds a PARODY database's index file. This example rebuilds the indexes for the Personnel application's object database.

ANSI.SYS

Before you run the applications, make sure that the DOS ANSI.SYS device driver is installed from your CONFIG.SYS program. If it is not, you will see a display similar to the one shown here when you run LIFE.EXE.

```
<-[2J<-[7;21HThe Game of Life<-[9;23H1: Build
<-[10;23H2: Display<-[11;23H3: Run
<-[12;23H4: Step<-[13;23H5: Load<-[14;23H6: Save
<-[15;23H9: Return<-[17;21HSelect>
```

The display just shown contains the ANSI.SYS command protocols that clear the screen and position the cursor. With the ANSI.SYS device driver installed, DOS simply displays those commands on the screen as if they were common text.

Run the Example Application Programs

Run the LIFE.EXE program. Exit it immediately. There should now be two additional files, named LIFE.DAT and LIFE.NDX. They are the PARODY database for the example Life application. They will be only four bytes long each. Because you have not added any objects, the files will not be any larger than they are. Because the Life application uses keyless objects only, the LIFE.NDX file will never grow beyond its initial four-byte length.

Run the PERSONEL.EXE program. Exit it immediately. There should now be two additional files named PERSONEL.DAT and PERSONEL.NDX. They are the PARODY database for the example Personnel application. They will each be only four bytes long. Because you have not added any objects, the files will not be any larger than they are.

Run the PERSONEL.EXE program and add, change, list, and delete employee, department, project, and assignment objects by using the menu commands. Exit and rerun the application to see that the objects are, in fact, persistent.

Run the LIFE.EXE program. Build and save universes of populated cells by using the menu commands. Make a note of the object addresses that the program reports when you save objects. Exit and rerun the application to load those objects. You will see them return to the program as persistent objects.

Rebuild the Indexes

Run the INDEX.EXE program. It will display each of the objects as it finds them and it will rebuild the indexes. You can verify that INDEX.EXE works by deleting or renaming the PERSONEL.NDX file before you run INDEX.EXE. After the program is done, the PERSONEL.NDX file will be reestablished, which you can verify by running PERSONEL.EXE again and retrieving objects.

▼ The Files

Following are the MAKEFILEs for Borland C++, Microsoft C++, and Zortech C++. After that is the PARODY.BLD file that all three MAKEFILEs use to build the PARODY.LIB object library.

Listing A.1 MAKEFILE.BCC

```
 1| # ==============================================
 2| # makefile.bcc - Borland C++ 3.1
 3| # ==============================================
 4|
 5| .cpp.obj:
 6|     bcc -c -m$(MODEL) {$* }
 7|
 8| # ==============================================
 9| # set COMPILER to path where bcc is installed
10| # ==============================================
11| COMPILER=c:\borlandc
12|
13| # ==============================================
14| # set MODEL to the memory model
15| # ==============================================
16| MODEL=l
17|
18| all : index.exe personel.exe life.exe
19|    echo all done
20|
21| # ==============================================
22| # tlink options
23| # ==============================================
24| STARTUP=$(COMPILER)\lib\c0$(MODEL)
```

```
25| LIBS=parody $(COMPILER)\lib\c$(MODEL)
26| TLINK=tlink /m $(STARTUP) $&
27|
28| # ==========================================
29| # define the Personnel system class library
30| # ==========================================
31| PERSONEL=employee.obj dept.obj project.obj assign.obj
32| PLINK=$(PERSONEL),$.,$&
33|
34| index.exe : index.obj $(PERSONEL) parody.lib
35|    $(TLINK) $(PLINK),$(LIBS)
36|
37| personel.exe : personel.obj $(PERSONEL) parody.lib
38|    $(TLINK) $(PLINK),$(LIBS)
39|
40| life.exe : life.obj parody.lib
41|    $(TLINK),$.,$&,$(LIBS)
42|
43| # ==========================================
44| # Build the PARODY.LIB class library
45| # ==========================================
46| parody.lib :   parody.obj   \
47|                btree.obj    \
48|                key.obj      \
49|                tnode.obj    \
50|                node.obj     \
51|                linklist.obj \
52|                strings.obj  \
53|                gui.obj
54|    del parody.lib
55|    tlib parody @parody.bld
```

Listing A.2 MAKEFILE.MSC

```
 1| # ==========================================
 2| # makefile.msc - Microsoft C++ 7.0
 3| # ==========================================
 4|
 5| .cpp.obj:
 6|     cl /DMSC /c /W3 /Gs /A$(MODEL) $*.cpp
 7|
 8| # ==========================================
 9| # set MODEL to the memory model
10| # ==========================================
```

```
11| MODEL=L
12|
13| all : index.exe personel.exe life.exe
14|
15| # ==========================================
16| # link options
17| # ==========================================
18| LINK=link /ST:8192 %|fF
19| LIBS=%|fF.exe,%|fF,parody
20|
21| # ==========================================
22| # define the Personnel system class library
23| # ==========================================
24| PERSONEL=employee.obj dept.obj project.obj assign.obj
25|
26| index.exe : index.obj $(PERSONEL) parody.lib
27|    $(LINK) $(PERSONEL),$(LIBS);
28|
29| personel.exe : personel.obj $(PERSONEL) parody.lib
30|    $(LINK) $(PERSONEL),$(LIBS);
31|
32| life.exe : life.obj parody.lib
33|    $(LINK),$(LIBS);
34|
35| # ==========================================
36| # Build the PARODY.LIB class library
37| # ==========================================
38| parody.lib :   parody.obj   \
39|                btree.obj    \
40|                key.obj      \
41|                tnode.obj    \
42|                node.obj     \
43|                linklist.obj \
44|                strings.obj  \
45|                gui.obj
46|    del parody.lib
47|    lib parody @parody.bld
```

Listing A.3 MAKEFILE.ZTC

```
 1| # ==========================================
 2| # makefile.ztc - Zortech C++ 3.0
 3| # ==========================================
 4|
```

```
 5| .cpp.obj:
 6|     ztc -c -m$(MODEL) -DZTC $*.cpp
 7|
 8| # ==========================================
 9| # set MODEL to the memory model
10| # ==========================================
11| MODEL=L
12|
13| all : index.exe personel.exe life.exe
14|    echo all done
15|
16| # ==========================================
17| # linker options
18| # ==========================================
19| LINK=ztc -DZTC -m$(MODEL) $*.obj
20|
21| # ==========================================
22| # define the Personnel system class library
23| # ==========================================
24| PERSONEL=employee.obj dept.obj project.obj assign.obj
25|
26| index.exe : index.obj $(PERSONEL) parody.lib
27|    $(LINK) $(PERSONEL) parody.lib
28|
29| personel.exe : personel.obj $(PERSONEL) parody.lib
30|    $(LINK) $(PERSONEL) parody.lib
31|
32| life.exe : life.obj parody.lib
33|    $(LINK) parody.lib
34|
35| # ==========================================
36| # Build the PARODY.LIB class library
37| # ==========================================
38| parody.lib :   parody.obj   \
39|                btree.obj    \
40|                key.obj      \
41|                tnode.obj    \
42|                node.obj     \
43|                linklist.obj \
44|                strings.obj  \
45|                gui.obj
46|    del parody.lib
47|    zorlib parody /C /N @parody.bld
```

Listing A.4 PARODY.BLD

```
1| +parody.obj     &
2| +btree.obj      &
3| +key.obj        &
4| +tnode.obj      &
5| +node.obj       &
6| +linklist.obj   &
7| +strings.obj    &
8| +gui
```

▼ Appendix B

PARODY Reference Guide

This appendix is the programmer's reference guide to PARODY. It describes: (1) global constants that alter PARODY's behavior; (2) PARODY classes and their interface member functions; and (3) PARODY data structures.

▼ Global Constants

Table B.1 shows the global constants that you can change to make compile-time modifications to the way PARODY operates.

Table B.1 Global Constants

Constant File	Description	Default	Header
SCREENHEIGHT	Screen height	25	gui.h
SCREENWIDTH	Screen width	80	gui.h
MAXCOMMANDS	Maximum menu commands	9	gui.h
nodelength	Length of a PARODY disk node	128	parody.h

▼ Data Types

These are data types defined as **enum**s and **typedef**s.

Bool (parody.h)

An enumerated Boolean data type that may have one of two values: False = 0 and True = 1.

ObjAddr (parody.h)

An integer type that specifies a persistent object's address.

NodeNbr (parody.h)

An integer type that specifies disk node addresses as serial numbers 1, 2, 3, ...

▼ Class Library Reference

This section describes the application program interface (API) to PARODY. It identifies each of the API classes, the header file where the class is defined, and the member functions that using programs call.

GUI (gui.h)

The **GUI** class implements the generic user interface. There is a global object of type **GUI** named **gui**. The program calls the member functions through this object. The program may also do console input and output with the standard **cin** and **cout** objects.

void UserMenu(char *title, ...)

Displays and executes a menu. The first parameter is the menu title, which is followed by a series of parameter pairs, one for each menu selection. Each pair contains the address of the selection's text and the address of a void function with no parameters that the program calls when the user makes the selection. The last pair is followed by a NULL pointer.

int Selection()

Returns the selection number, beginning with 1, of the user's most recent menu selection.

void UserInput(String& s, char *prompt, int len)
void UserInput(char& c, char *prompt)
void UserInput(int& i, char *prompt)

These functions display the prompt string and read the user's input into the referenced variable.

Bool YesNo(char *question)

Displays the question string and prompts the user to enter 'Y' or 'N' (case-insensitive). The function returns True if the user enters 'Y' and False if the user enters 'N'.

void Error(char *message)

Displays the message string, sounds the console's audible alarm, and calls the **AnyKey** function.

void AnyKey()

Displays an "Any key..." message and waits for the user to type a key.

void SetCursor(int x, int y)

Positions the cursor at the screen character coordinates specified by x and y.

void ClearScreen()

Clears the screen.

void WriteChar(char c, int x, int y)

Writes a single character on the screen at the x/y coordinate.

void StatusLine(char *s)

Displays a status line at the bottom of the screen.

Bool KBCharWaiting()

Tests for keyboard input. Returns True if the user has pressed a key, otherwise returns False.

char GetKBChar()

Reads the next key from the keyboard. If no key is waiting, waits for the next keystroke.

void PutBack(char c)

Sets the character as the next value returned from **GetKBChar**.

String (strings.h)

String()
String(char *s)
String(String& s)
String(int len, char fill = 0)

These constructors construct a **String** object from the parameters provided.

int Strlen()

Returns the length of the null-terminated string.

int Length()

Returns the length of the **String** object's buffer.

String right(int len)

Returns a **String** object with the right-most **len** objects from the original string.

String left(int len)

Returns a **String** object with the left-most **len** objects from the original string.

String mid(int len, int where)

Returns a **String** object with the **len** objects from the original string that begin at the **where** offset.

int FindChar(unsigned char ch)

Returns an offset relative to zero to the first occurrence of the specified character.

String& operator=(String& s)

Overloaded assignment operator.

operator char *()

Returns a pointer to the object's string value in contexts where the compiler expects to see a **char** pointer.

String operator+(String& s)

void operator+=(String& s)

String concatenation operation.

Bool operator==(String& s)
Bool operator!=(String& s)
Bool operator>(String& s)
Bool operator<(String& s)
Bool operator<=(String& s)
Bool operator>=(String& s)

Overloaded relational operators. They use **strcmp** to compare string values.

char& operator[](int n)

Overloaded subscript operator. Permits read/write access to a single character position in a string.

friend ostream& operator<< (ostream& os, String& str)
friend istream& operator>> (istream& is, String& str)

Read and write the string value to **iostream** classes.

LinkedListHead (linklist.h)

void *FirstListEntry()

Returns the address of the first **LinkedListEntry** object in a linked list.

void *LastListEntry()

Returns the address of the last **LinkedListEntry** object in a linked list.

LinkedListEntry (linklist.h)

This abstract base class is used to derive other classes that are to be entries in a linked list. Objects of this class must be associated with a **LinkedListHead** object either when they are constructed or when they are put into a particular list.

LinkedListEntry(LinkedListHead *lh = NULL)

Constructs the **LinkedListEntry** component of the derived class.

void AppendListEntry(LinkedListHead *lh = NULL)

Inserts the entry at the end of the list. If **lh** is NULL, the associated **LinkedListHead** object must have been specified by the constructor.

void PrependListEntry(LinkedListHead *lh = NULL)

Inserts the entry at the beginning of the list. If **lh** is NULL, the associated **LinkedListHead** object must have been specified by the constructor.

void InsertListEntry(void *fe, LinkedListHead *lh = NULL)

Inserts the entry following the entry specified by **fe**. If **lh** is NULL, the associated **LinkedListHead** object must have been specified by the constructor.

void DeleteListEntry()

Deletes the entry from the list.

void *NextListEntry()

Returns the address of the next **LinkedListEntry** object in a linked list.

void *PrevListEntry()

Returns the address of the previous **LinkedListEntry** object in a linked list.

Parody (parody.h)

Bool RebuildingIndexes()

Returns True if the PARODY database is currently rebuilding indexes. Returns False otherwise.

void GetObjectHeader(ObjAddr nd, ObjectHeader &objhdr)

Gets the Object Header record at the node address specified by **nd**.

void RebuildIndexes(ObjAddr nd)

Tells PARODY to rebuild indexes for the current object. The indexes will point to the **nd** node number.

Key (key.h)

The **Key** class is an abstract base class. Derive primary and secondary key classes from **Key**.

virtual int operator> (Key& key) = 0;
virtual int operator== (Key& key) = 0;

Overloaded relational operators provided by classes derived from **Key**.

virtual Key& operator=(Key& key);

Overloaded assignment operator provided by classes derived from **Key**.

virtual void Write(fstream& bfile) = 0;
virtual void Read(fstream& bfile) = 0;

Functions to read and write a fixed-length key value for an index file.

virtual Bool isNullValue() = 0;

Returns True if the derived **Key** object has a null (empty) value. Provided by classes derived from **Key**. The null representation is application dependent.

void PrimaryKey()

Coerces the **Key** object into becoming the primary key without respect to its position in the constructor sequence. Needed to establish a concatenated key as the primary.

void Relate(int cid)

Relates the class that owns the key to the specified class identification, which is presumed to be another class that has the same **Key** type as its primary key.

void FindObject()
void FirstObject()
void LastObject()
void NextObject()
void PreviousObject()

Find and navigate the persistent objects associated with the **Key** object in the sequence of the **Key**.

Persistent (parody.h)

The **Persistent** class is an abstract base class from which persistent object classes are derived.

virtual void Read() = 0;
virtual void Write() = 0;

Functions to read and write the data values of the persistent object. These functions call **ReadObject** and **WriteObject** for each of the data members.

void LoadObject(ObjAddr nd = 0)

Called from the persistent object's constructor to load the object into memory from the PARODY database. If the **nd** parameter is nonzero, the object is assumed to be keyless.

void SaveObject()

Called from the persistent object's destructor to save the object from memory into the PARODY database.

void ReadObject(void *buf, int length)
void WriteObject(void *buf, int length)
void ReadObject(—type— &)
void WriteObject(—type— &)

Called from **Read** and **Write** to read and write data members. The --type-- variants are overloaded functions that specify references to types int, float, long, String, double, and ObjAddr.

Bool AddObject()

Tells PARODY to add a newly created object to the database when the destructor calls **SaveObject**. Returns False if the addition cannot be made due to integrity violations.

Bool ChangeObject()

Tells PARODY to write a changed object back to the database when the destructor calls **SaveObject**. Returns False if the change cannot be made due to integrity violations.

Bool DeleteObject()

Tells PARODY to delete the object from the database when the destructor calls **SaveObject**. Returns False if the delete cannot be made due to integrity violations.

Bool ObjectExists()

Returns True if PARODY found the specified object in the database. Valid after the object is instantiated and after calls to the next five functions discussed.

void FindObject(Key *key)

Finds the object associated with the value in the **key** parameter.

void FirstObject(Key *key = NULL)
void LastObject(Key *key = NULL)
void NextObject(Key *key = NULL)
void PreviousObject(Key *key = NULL)

Finds the object by position relative to the specified key. If the **key** parameter is NULL, uses the primary key. If the class has no key (a keyless object), the retrieval uses the incidental position of the object in the physical database.

ObjAddr ObjectAddress()

Returns the object's address in the PARODY database.

Handle (parody.h)

The **Handle** class is an abstract base class for persistent objects that use the handle/copy idiom for multiple-copy objects. The object body, of which there is one master object instantiated for a particular object, will derive from the **Persistent** class. The object handles, for which there are multiple copies instantiated for a particular object will derive from the **Handle** class.

The **Handle** class has these member functions. They operate the same as the **Persistent** member functions with the same names.

Bool AddObject()
Bool ChangeObject()
Bool DeleteObject()
Bool ObjectExists()
void FindObject(Key *key)
void FirstObject(Key *key = NULL)
void LastObject(Key *key = NULL)
void NextObject(Key *key = NULL)
void PreviousObject(Key *key = NULL)
ObjAddr ObjectAddress()

▼ Disk Data Structures

Data structures on disk are not always obvious in view of the class definitions from which they are written. This discussion describes how PARODY organizes data on disk and the format of each physical record.

Generic Disk Files

PARODY builds two disk files: the object data file and the index file. Both files have the filename specified by the declaration of the **Parody** object. The object

data file has the extension .DAT; the index file has the extension .NDX. Both files are represented by a file header block at the beginning of the file and a variable number of fixed-length data nodes as shown in Figure B.1.

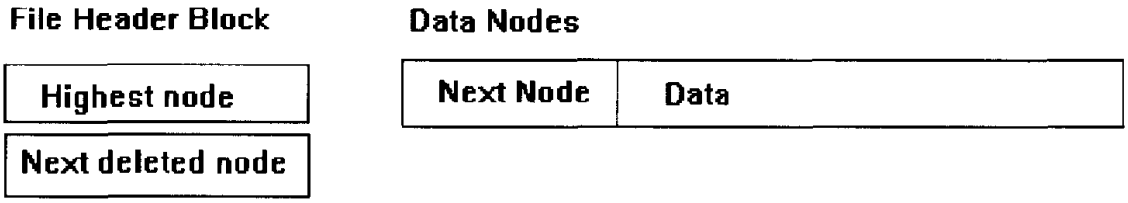

Figure B.1 *File Header Block and Data Nodes.*

File Header Block

Each file contains a header block with two **NodeNbr** variables. The first specifies the node number of the most recently deleted node. The second specifies the highest node currently allocated in the database.

Data Nodes

Each file consists of a variable number of fixed-length nodes. The first node is node number 1 and so on. Each node begins with a **NodeNbr** variable that points to the next node in a logical list of nodes that form a set. The last node in the set has value zero in the **NodeNbr** variable. If the node has been deleted by its user, the set is a list of nodes the first of which is pointed to by the deleted node pointer in the File Header Block.

Persistent Objects

The persistent object .DAT file consists of a File Header Block and Data Nodes. Each persistent object is stored in a set of nodes. Each node in the set has the **NodeNbr** variable mentioned above to form the set followed by the Object Header and Object Data as shown in Figure B.2.

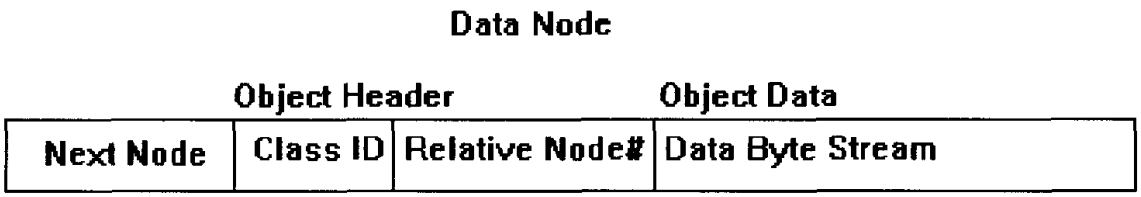

Figure B.2 *Object Header and Data*

Object Header

The Object Header contains the integer class identification code for the object followed by a relative **NodeNbr** variable for the object. The first relative node number is zero to identify that the node is the first node of the object. Subsequent relative node numbers are numbered sequentially. This pattern allows PARODY to identify the first node of an object without an index reference to it.

Object Data

Object data in the node is an unformatted byte stream built by the object's own **Write** member function. If the stream overflows a node, the stream is continued in the data area of the next node in the object's set. PARODY pads the last node in the object's set with zeros.

B-Trees

The B-Tree .NDX file consists of a File Header Block and Data Nodes. There is a set of nodes that contains the Index Header Records, and there are individual one-node sets that contain the B-Tree Node Header and an array of fixed-length key values.

B-Tree Index Header Nodes

Node 1 in the .NDX file is the first node in a set of nodes that contain Index Header Blocks for the persistent classes. Each Index Header Node is an array of Index Header Blocks for one class. The first node in the set contains the array for class identification zero; the second set contains the array for class identification one; and so on. The first entry in the array is for the primary key of the class; the second entry is for the first secondary key; and so on. Figure B.3 shows the Index Header Node set.

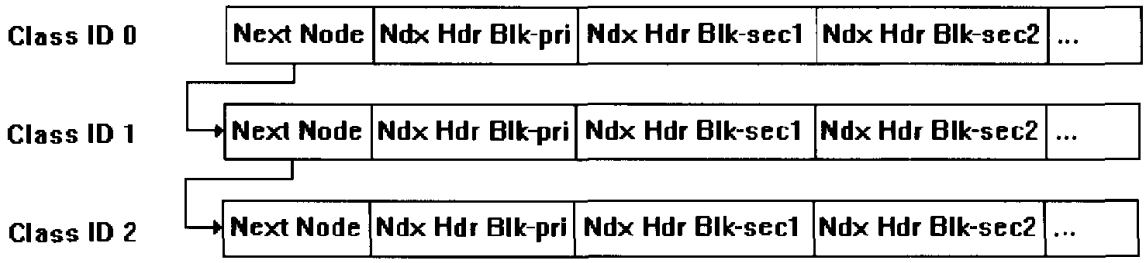

Figure B.3 *Index Header Nodes*

Index Header Blocks

Each Index Header Block for a class contains a **NodeNbr** variable that points to the root node for the B-tree and an integer value that specifies the length of a key in the B-tree.

B-Tree Node Header and Keys

The Data Nodes in a B-tree contain the B-tree Node Header block followed by a fixed-length array of keys.

The B-tree Node Header Block is in the format shown in its structure in Figure B.4.

```
struct TNodeHeader {
    Bool isleaf;            // true if node is a leaf
    NodeNbr parent;         // parent to this node
    NodeNbr leftsibling;    // left sibling node
    NodeNbr rightsibling;   // right sibling node
    int keycount;           // number of keys in this node
    NodeNbr lowernode;      // lower node associated with
                             // keys < keys in this node
};
```

Figure B.4 *The B-tree Node Header*

A key in the array consists of its key data value followed by the **NodeNbr** address of the object in the .DAT file to which the key points. If the node is not a leaf node, each key array entry also includes a **NodeNbr** variable that points to the B-tree node at the next lower level in the tree that contains keys with collating sequence values greater than the current key value and less than the next adjacent key value in the array.

Appendix C

Software Listings

Table of Contents

Listings by Category

Generic User Interface Class

C.1 GUI.H 200
C.2 GUI.CPP 201

String Class

C.3 STRINGS.H 206
C.4 STRINGS.CPP 208

Linked List Class

C.5 LINKLIST.H 210
C.6 LINKLIST.CPP 211

The Game of Life

C.7 LIFE.CPP 213

Personnel Application Classes

C.8 EMPLOYEE.H 221
C.9 EMPLOYEE.CPP 223
C.10 DEPT.H 227
C.11 DEPT.CPP 229
C.12 PROJECT.H 232
C.13 PROJECT.CPP 235
C.14 ASSIGN.H 239
C.15 ASSIGN.CPP 241

Personnel Application

C.16 PERSONEL.H 245
C.17 PERSONEL.CPP 246

PARODY

C.18 PARODY.H 257
C.19 PARODY.CPP 261
C.20 KEY.H 276
C.21 KEY.CPP 277
C.22 BTREE.H 278
C.23 BTREE.CPP 280
C.24 TNODE.CPP 291
C.25 NODE.H 300
C.26 NODE.CPP 301

Example Index Builder

C.27 INDEX.CPP 304

Listings Alphabetically by File Name

Name	Listing	Page
ASSIGN.CPP	C.15	241
ASSIGN.H	C.14	239
BTREE.CPP	C.23	280
BTREE.H	C.22	278
DEPT.CPP	C.11	229
DEPT.H	C.10	227
EMPLOYEE.CPP	C.9	223
EMPLOYEE.H	C.8	221
GUI.CPP	C.1	200
GUI.H	C.2	201
INDEX.CPP	C.27	304
KEY.CPP	C.21	277
KEY.H	C.20	276
LIFE.CPP	C.7	213
LINKLIST.CPP	C.5	210
LINKLIST.H	C.6	211
NODE.CPP	C.26	301
NODE.H	C.25	300
PARODY.CPP	C.19	261
PARODY.H	C.18	257
PERSONEL.CPP	C.17	246
PERSONEL.H	C.16	245
PROJECT.CPP	C.13	235
PROJECT.H	C.12	232
STRINGS.CPP	C.3	206
STRINGS.H	C.4	208
TNODE.CPP	C.24	291

Listing C.1 GUI.H

```
 3| #ifndef USER_H
 4| #define USER_H
 5|
 6| // ==========================
 7| // Generic User Interface (GUI)
 8| // ==========================
 9|
10| #ifdef ZTC
11| #include <iostream.hpp>
12| #else
13| #include <iostream.h>
14| #endif
15| #include <stdarg.h>
16| #include "parody.h"
17|
18| // ---- screen dimensions
19| const int SCREENHEIGHT = 25;
20| const int SCREENWIDTH  = 80;
21|
22| // --- where to display menus
23| const int menux = (SCREENWIDTH/2-SCREENWIDTH/4);
24| const int menuy = (SCREENHEIGHT/2-SCREENHEIGHT/4);
25|
26| const int MAXCOMMANDS = 9; // maximum menu commands
27|
28| // --- function key values
29| const int UPARROW    = 200;
30| const int DOWNARROW  = 208;
31| const int RIGHTARROW = 205;
32| const int LEFTARROW  = 203;
33| const int ESC        = 27;
34|
35| // ---- the Generic User Interface class
36| class GUI   {
37|    int selection;
38|    char putback;
39| public:
40|    GUI() { selection = putback = 0; }
41|    // -- menu processing
42|    void UserMenu(char *title, ...);
43|    int Selection() { return selection; }
44|    // -- data field entry
```

```
45|     void UserInput(String& s, char *prompt, int len);
46|     void UserInput(char& c, char *prompt);
47|     void UserInput(int& i, char *prompt);
48|     // -- standard dialogs
49|     Bool YesNo(char *question);
50|     void Error(char *message);
51|     void AnyKey();
52|     // -- screen operations
53|     void SetCursor(int x, int y);
54|     void ClearScreen();
55|     void WriteChar(char c, int x, int y);
56|     void StatusLine(char *s);
57|     // -- keyboard operations
58|     Bool KBCharWaiting();
59|     char GetKBChar();
60|     void PutBack(char c);
61| };
62|
63| // -- menu selection structure
64| struct MenuSelection {
65|     char *label;
66|     void (*menufunction)(void);
67| };
68|
69| extern GUI gui;       // the GUI object
70|
71| #endif
```

Listing C.2 GUI.CPP

```
 1| // ----- gui.cpp
 2|
 3| // ============================
 4| // Generic User Interface (GUI)
 5| // ============================
 6|
 7| #ifdef ZTC
 8| #include <iostream.hpp>
 9| #include <iomanip.hpp>
10| #else
11| #include <iostream.h>
12| #include <iomanip.h>
13| #endif
```

```
14| #include <stdarg.h>
15| #include <ctype.h>
16| #include <conio.h>
17| #include <bios.h>
18| #include "gui.h"
19| #include "strings.h"
20|
21| // --- the GUI object
22| GUI gui;
23|
24| // ---------------
25| // menu user interface
26| // ---------------
27| void GUI::UserMenu(char *title, ...)
28| {
29|    MenuSelection sels[MAXCOMMANDS];
30|    va_list ap;
31|    va_start(ap, title);
32|    int cmdno = 0;
33|
34|    // -- gather the menu from the caller's parameters
35|    while ((sels[cmdno].label=va_arg(ap, char *)) != NULL) {
36|       if (cmdno == MAXCOMMANDS)
37|          break;
38|       void *fn = va_arg(ap, void *);
39|       sels[cmdno++].menufunction = (void(*)()) fn;
40|    }
41|
42|    for (;;) {
43|       ClearScreen();
44|       SetCursor(menux, menuy);
45|       // --- display the menu
46|       cout << title;
47|       for (int i = 0; i < cmdno; i++)  {
48|          SetCursor(menux+2, menuy+2+i);
49|          cout << (i+1) << ": " << sels[i].label;
50|       }
51|       SetCursor(menux+2, menuy+2+i);
52|       cout << MAXCOMMANDS << ": Return";
53|
54|       // -- get the user's selection
55|       SetCursor(menux, menuy+4+i);
56|       cout << "Select> ";
57|       cin >> selection;
58|
```

```
 59|        if (selection > 0 && selection <= cmdno)  {
 60|           // - call the menu selection function
 61|           ClearScreen();
 62|           cout << sels[selection-1].label << ':'
 63|              << endl << endl;
 64|           (*sels[selection-1].menufunction)();
 65|        }
 66|        else if (selection == MAXCOMMANDS)
 67|           break;   // exiting the menu
 68|        else
 69|           cout << '\a';
 70|     }
 71| }
 72|
 73| // ----------------
 74| // character input
 75| // ----------------
 76| void GUI::UserInput(char& c, char *prompt)
 77| {
 78|    cout << endl << prompt << ": ";
 79|    cout.flush();
 80|    cin >> c;
 81| }
 82|
 83| // ----------------
 84| // integer input
 85| // ----------------
 86| void GUI::UserInput(int& i, char *prompt)
 87| {
 88|    cout << endl << prompt << ": ";
 89|    cout.flush();
 90|    cin >> i;
 91| }
 92|
 93| // ----------------
 94| // String input
 95| // ----------------
 96| void GUI::UserInput(String& s, char *prompt, int len)
 97| {
 98|    cout << endl << prompt << ": ";
 99|    cout.flush();
100|    String str(len);
101|    cin >> str;
102|    s = str;
103| }
```

```
104|
105| // ----------------
106| // yes/no user interface
107| // ----------------
108| Bool GUI::YesNo(char *question)
109| {
110|    char c = ' ';
111|    while (c != 'y' && c != 'n')  {
112|       UserInput(c, question);
113|       c = tolower(c);
114|       if (c != 'y' && c != 'n')
115|          cout << '\a';
116|    }
117|    cout.flush();
118|    return (Bool) (c == 'y');
119| }
120|
121| // ----------------
122| // error message
123| // ----------------
124| void GUI::Error(char *message)
125| {
126|    cout << endl << '\a' << message << endl;
127|    AnyKey();
128| }
129|
130| // ----------------
131| // get a keyboard char
132| // ----------------
133| char GUI::GetKBChar()
134| {
135|    int c;
136|    if (putback)   {
137|       c = putback;
138|       putback = 0;
139|    }
140|    else  {
141|       c = _bios_keybrd(_KEYBRD_READ);
142|       if ((c & 0xff) == 0)
143|          c = (c >> 8) | 0x80;
144|       else
145|          c &= 0xff;
146|    }
147|    return (char) c;
148| }
```

```
149|
150| // ----------------
151| // put back a keyboard char
152| // ----------------
153| void GUI::PutBack(char c)
154| {
155|    putback = c;
156| }
157|
158| // ----------------
159| // Display a status line
160| // ----------------
161| void GUI::StatusLine(char *s)
162| {
163|    SetCursor(0, SCREENHEIGHT-1);
164|    cout << s;
165|    cout.flush();
166|    int len = strlen(s);
167|    while (len++ < SCREENWIDTH-1)
168|       cout << ' ';
169|    cout.flush();
170| }
171|
172|
173| // ----------------
174| // Prompt for and read any key
175| // ----------------
176| void GUI::AnyKey()
177| {
178|    cout << endl << "Any key..." << endl;
179|    GetKBChar();
180| }
181|
182| // ----------------
183| // Position the cursor
184| // ----------------
185| void GUI::SetCursor(int x, int y)
186| {
187|    cout << "\033[" << (y+1) << ';' << (x+1) << 'H';
188|    cout.flush();
189| }
190|
191| // ----------------
192| // Clear the screen
193| // ----------------
```

```
194| void GUI::ClearScreen()
195| {
196|    cout << "\033[2J";
197|    cout.flush();
198| }
199|
200| // ---------------
201| // Write a single character to the screen
202| // ----------------
203| void GUI::WriteChar(char c, int x, int y)
204| {
205|    SetCursor(x, y);
206|    cout << c;
207|    cout.flush();
208| }
209|
210| // ---------------
211| // Test for a keyboard character waiting
212| // ----------------
213| Bool GUI::KBCharWaiting()
214| {
215|    return (Bool) _bios_keybrd(_KEYBRD_READY);
216| }
```

Listing C.3 STRINGS.H

```
  1| // ---- strings.h
  2|
  3| #ifndef STRINGS_H
  4| #define STRINGS_H
  5|
  6| #ifdef ZTC
  7| #include <iostream.hpp>
  8| #else
  9| #include <iostream.h>
 10| #endif
 11| #include <string.h>
 12|
 13| // ============================
 14| // BASIC-like String Class
 15| // ============================
 16| class String    {
 17|    char *sptr;
 18|    int length;
```

```
19|     void putstr(char *s);
20| public:
21|     // ---- construct a null string
22|     String() { sptr = NULL; }
23|     // - construct with char * initializer
24|     String(char *s);
25|     // --- copy constructor
26|     String(String& s);
27|     // ---- construct with a size and fill character
28|     String(int len, char fill = 0);
29|     // --- destructor
30|     ~String() { delete sptr; }
31|     // --- return the length of a string
32|     int Strlen() { return strlen(sptr); }
33|     int Length() { return length; }
34|     // -- substring: right len chars
35|     String right(int len);
36|     // -- substring: left len chars
37|     String left(int len);
38|     // -- substring: middle len chars starting from where
39|     String mid(int len, int where);
40|     int FindChar(unsigned char ch);
41|     // ----- assignment
42|     String& operator=(String& s);
43|     // ----- conversion to char *
44|     operator char *() { return sptr; }
45|     // - concatenation operator (str1 + str2;)
46|     String operator+(String& s);
47|     // - concatenation operator (str1 += str2;)
48|     void operator+=(String& s) { *this = *this + s; }
49|     // --- relational operators
50|     Bool operator==(String& s)
51|        { return (Bool) (strcmp(sptr,s.sptr) == 0); }
52|     Bool operator!=(String& s)
53|        { return (Bool) (strcmp(sptr,s.sptr) != 0); }
54|     Bool operator>(String& s)
55|        { return (Bool) (strcmp(sptr,s.sptr) > 0); }
56|     Bool operator<(String& s)
57|        { return (Bool) (strcmp(sptr,s.sptr) < 0); }
58|     Bool operator<=(String& s)
59|        { return (Bool) (!(*this > s)); }
60|     Bool operator>=(String& s)
61|        { return (Bool) (!(*this < s)); }
62|     // --- subscript
63|     char& operator[](int n) { return sptr[n]; }
```

```
64|     // --- stream I/O
65|     friend ostream& operator<< (ostream& os, String& str);
66|     friend istream& operator>> (istream& is, String& str);
67| };
68|
69| #endif
```

Listing C.4 STRINGS.CPP

```
 1| // ---- strings.cpp
 2|
 3| #include "parody.h"
 4|
 5| // --- put a string into a String
 6| void String::putstr(char *s)
 7| {
 8|     delete sptr;
 9|     length = strlen(s);
10|     sptr = new char[length+1];
11|     strcpy(sptr, s);
12| }
13|
14| // - convert from char *
15| String::String(char *s)
16| {
17|     sptr = NULL;
18|     putstr(s);
19| }
20|
21| // --- copy constructor
22| String::String(String& s)
23| {
24|     sptr = NULL;
25|     putstr(s.sptr);
26| }
27|
28| // ---- construct with a size and fill character
29| String::String(int len, char fill)
30| {
31|     length = len;
32|     sptr = new char[length+1];
33|     memset(sptr, fill, length);
34|     *(sptr+len) = '\0';
35| }
```

```
36|
37| // ---- assignment
38| String& String::operator=(String &s)
39| {
40|    if (this != &s)
41|       putstr(s.sptr);
42|    return *this;
43| }
44|
45| // --- concatenation operator (str1 + str2)
46| String String::operator+(String &s)
47| {
48|    String temp(strlen(s.sptr) + strlen(sptr));
49|    strcpy(temp.sptr, sptr);
50|    strcat(temp.sptr, s.sptr);
51|    return temp;
52| }
53|
54| // --- substring: right len chars
55| String String::right(int len)
56| {
57|    String tmp(sptr + strlen(sptr) - len);
58|    return tmp;
59| }
60|
61| // --- substring: left len chars
62| String String::left(int len)
63| {
64|    String tmp(len);
65|    strncpy(tmp.sptr, sptr, len);
66|    return tmp;
67| }
68|
69| // --- substring: middle len chars starting from where
70| String String::mid(int len, int where)
71| {
72|    String tmp(len);
73|    if (where < (int) strlen(sptr))
74|       strncpy(tmp.sptr,sptr+where,len);
75|    return tmp;
76| }
77|
78| // -- find offset to first instance of specified char
79| int String::FindChar(unsigned char ch)
80| {
```

```
 81|    char *cp = strchr(sptr, ch);
 82|    if (cp == NULL)
 83|       return -1;
 84|    return (int) (cp - sptr);
 85| }
 86|
 87| // --- stream I/O
 88| ostream& operator<< (ostream& os, String& str)
 89| {
 90|    os << str.sptr;
 91|    return os;
 92| }
 93|
 94| istream& operator>> (istream& is, String& str)
 95| {
 96|    *str.sptr = '\0';
 97|    while (*str.sptr == '\0')
 98|       is.getline(str.sptr, str.length+1);
 99|    return is;
100| }
```

Listing C.5 LINKLIST.H

```
  1| // ------ linklist.h
  2|
  3| #ifndef LINKLIST_H
  4| #define LINKLIST_H
  5|
  6| #ifdef ZTC
  7| #include <iostream.hpp>
  8| #else
  9| #include <iostream.h>
 10| #endif
 11|
 12| #ifndef NULL
 13| #define NULL 0
 14| #endif
 15|
 16| class LinkedListEntry;
 17|
 18| // ============================
 19| // Linked Listhead
 20| // ============================
 21| class LinkedListHead {
```

```
22|    friend class LinkedListEntry;
23|    void *first;
24|    void *last;
25| public:
26|    LinkedListHead() { first = last = NULL; }
27|    virtual ~LinkedListHead() { /* null */ }
28|    void *FirstListEntry()    { return first; }
29|    void *LastListEntry()     { return last; }
30| };
31|
32| // ============================
33| // Linked List Entry
34| // ============================
35| class LinkedListEntry   {
36|    LinkedListEntry *prev;
37|    LinkedListEntry *next;
38| protected:
39|    LinkedListHead *listhead;
40|    LinkedListEntry(LinkedListHead *lh = NULL);
41|    virtual ~LinkedListEntry() { DeleteListEntry(); }
42| public:
43|    void AppendListEntry(LinkedListHead *lh = NULL);
44|    void PrependListEntry(LinkedListHead *lh = NULL);
45|    void InsertListEntry(void *fe, LinkedListHead *lh = NULL);
46|    void DeleteListEntry();
47|    void *NextListEntry() { return next; }
48|    void *PrevListEntry() { return prev; }
49| };
50|
51| #endif
```

Listing C.6 LINKLIST.CPP

```
 1| // --------- linklist.cpp
 2|
 3| #include "linklist.h"
 4|
 5| LinkedListEntry::LinkedListEntry(LinkedListHead *lh) :
 6|                              listhead(lh)
 7| {
 8|    prev = next = NULL;
 9|    if (lh != NULL)
10|       AppendListEntry();
11| }
```

```
12|
13| // -- append a list entry to the end of the list
14| void LinkedListEntry::AppendListEntry(LinkedListHead *lh)
15| {
16|    if (lh != NULL)
17|       listhead = lh;
18|    if (listhead != NULL)   {
19|       if (listhead->first == NULL)
20|          listhead->first = this;
21|       prev = (LinkedListEntry *) listhead->last;
22|       if (prev != NULL)
23|          prev->next = this;
24|       listhead->last = this;
25|    }
26| }
27|
28| // -- prepend a list entry to the beginning of the list
29| void LinkedListEntry::PrependListEntry(LinkedListHead *lh)
30| {
31|    if (lh != NULL)
32|       listhead = lh;
33|    if (listhead != NULL)   {
34|       if (listhead->last == NULL)
35|          listhead->last = this;
36|       next = (LinkedListEntry *) listhead->first;
37|       if (next != NULL)
38|          next->prev = this;
39|       listhead->first = this;
40|    }
41| }
42|
43| // - Insert a list entry ahead of a specified (*fe) entry.
44| void LinkedListEntry::InsertListEntry(void *fe,
45|                                       LinkedListHead *lh)
46| {
47|    if (lh != NULL)
48|       listhead = lh;
49|    LinkedListEntry *ahd = (LinkedListEntry *) fe;
50|    if (listhead != NULL)   {
51|       if (ahd == NULL)
52|          AppendListEntry();
53|       else  {
54|          next = ahd;
55|          prev = ahd->prev;
56|          ahd->prev = this;
```

```
57|          if (prev == NULL)
58|             listhead->first = this;
59|          else
60|             prev->next = this;
61|       }
62|    }
63| }
64|
65| // ----- delete a list entry
66| void LinkedListEntry::DeleteListEntry()
67| {
68|    if (listhead != NULL)   {
69|       if (next != NULL)
70|          next->prev = prev;
71|       else
72|          listhead->last = prev;
73|       if (prev != NULL)
74|          prev->next = next;
75|       else
76|          listhead->first = next;
77|       listhead = NULL;
78|    }
79|    next = prev = NULL;
80| }
```

Listing C.7 LIFE.CPP

```
 1| // -------- life.cpp
 2|
 3| // ==========================================
 4| // The Game of Life - a simulation
 5| // ==========================================
 6|
 7| #ifdef ZTC
 8| #include <iostream.hpp>
 9| #include <strstrea.hpp>
10| #include <iomanip.hpp>
11| #else
12| #include <iostream.h>
13| #include <strstrea.h>
14| #include <iomanip.h>
15| #endif
16| #include <ctype.h>
17| #include "gui.h"
```

```
18|
19| #ifdef BCC
20| extern unsigned _stklen = 8192;
21| #endif
22| #ifdef ZTC
23| unsigned _stack = 8192;
24| #endif
25|
26| Parody *life;
27|
28| // --- dimensions of the Life world
29| const int HT = SCREENHEIGHT-1;
30| const int WD = SCREENWIDTH;
31|
32| const char Pop = 2;  // the populated display character
33|
34| // -- collection of neighboring cells
35| class Life : public Persistent {
36|    int world[WD][HT];
37|    int generation;
38|    Bool alive;
39|    int popcount;
40|    static int Rules[2][8];
41|    int CountNeighbors(int x, int y);
42|    void Read();
43|    void Write();
44| public:
45|    Life(ObjAddr objaddr = 0);
46|    ~Life() { SaveObject(); }
47|    void Generation();
48|    Bool isAlive() { return alive; }
49|    void Populate(int x, int y);
50|    void DePopulate(int x, int y);
51|    Bool isPopulated(int x, int y)
52|       { return (Bool) world[x][y]; }
53| };
54|
55| // --- read the persistent Life object
56| void Life::Read()
57| {
58|    int ct, x, y;
59|    ReadObject(ct);
60|    while (ct-)   {
61|       ReadObject(x);
62|       ReadObject(y);
```

```
 63|        Populate(x,y);
 64|     }
 65| }
 66|
 67| // --- write the persistent Life object
 68| void Life::Write()
 69| {
 70|    WriteObject(popcount);
 71|    for (int x = 0; x < WD; x++)  {
 72|       for (int y = 0; y < HT; y++)  {
 73|          if (world[x][y])  {
 74|             WriteObject(x);
 75|             WriteObject(y);
 76|          }
 77|       }
 78|    }
 79| }
 80|
 81| // --- rules for cell survival
 82| int Life::Rules[2][8] = {
 83|    // - cell is unpopulated
 84|    {0,0,0,1,0,0,0,0},
 85|    // - cell is populated
 86|    {0,0,1,0,0,0,0,0}
 87| };
 88|
 89| Life::Life(ObjAddr objaddr) : Persistent(*life, 0)
 90| {
 91|    for (int x = 0; x < WD; x++)
 92|       for (int y = 0; y < HT; y++)
 93|          world[x][y] = 0;
 94|    alive = False;
 95|    generation = 0;
 96|    popcount = 0;
 97|    LoadObject(objaddr);
 98| }
 99|
100| // ----- populate a cell
101| void Life::Populate(int x, int y)
102| {
103|    if (!world[x][y])
104|       popcount++;
105|    world[x][y] = 1;
106|    alive = True;
107| }
```

```
108|
109| // ----- depopulate a cell
110| void Life::DePopulate(int x, int y)
111| {
112|    if (world[x][y])
113|       -popcount;
114|    world[x][y] = 0;
115| }
116|
117| // --- count a cell's neighbors
118| int Life::CountNeighbors(int x, int y)
119| {
120|    int x1, y1, ct = 0;
121|    // -- count 3 rows
122|    for (y1 = y-1; y1 < y+2; y1++)
123|       // -- count 3 columns
124|       for (x1 = x-1; x1 < x+2; x1++)
125|          // - don't count self
126|          if (!(y1 == y && x1 == x))
127|             // - don't exceed margins
128|             if (y1 > -1 && y1 < HT && x1 > -1 && x1 < WD)
129|                // - count populated neighbor cells
130|                if ((world[x1][y1]) & 1)
131|                   ct++;
132|    return ct;
133| }
134|
135| // ---- evolve one generation
136| void Life::Generation()
137| {
138|    gui.SetCursor(0, HT);
139|    char st[81];
140|    strstream stat(st, 80, ios::out);
141|    stat << "Generation: " << generation++ << '\0';
142|    gui.StatusLine(stat.str());
143|    int x, y, newpop, oldpop, nct, cell;
144|    for (x = 0; x < WD; x++)   {
145|       for (y = 0; y < HT; y++)   {
146|          nct = CountNeighbors(x, y);
147|          cell = world[x][y];
148|          oldpop = cell & 1;
149|          newpop = (Rules[oldpop][nct]) << 1;
150|          cell = oldpop | newpop;
151|          world[x][y] = cell;
152|       }
```

```
153|     }
154|     alive = False;
155|     popcount = 0;
156|     for (x = 0; x < WD; x++)    {
157|        for (y = 0; y < HT; y++)    {
158|           cell = world[x][y];
159|           newpop = cell >> 1;
160|           oldpop = cell & 1;
161|           if (newpop != oldpop)    {
162|              if (newpop)
163|                 gui.WriteChar(Pop, x, y);
164|              else
165|                 gui.WriteChar(' ', x, y);
166|           }
167|           if (newpop) {
168|              alive = True;
169|              popcount++;
170|           }
171|           world[x][y] = newpop;
172|        }
173|     }
174| }
175|
176| void BuildLife();
177| void DisplayLife();
178| void RunLife();
179| void StepLife();
180| void LoadLife();
181| void SaveLife();
182|
183| Life *lf;
184|
185| void main()
186| {
187|     life = new Parody("LIFE");
188|     gui.UserMenu( "The Game of Life",
189|                 "Build",    BuildLife,
190|                 "Display",  DisplayLife,
191|                 "Run",      RunLife,
192|                 "Step",     StepLife,
193|                 "Load",     LoadLife,
194|                 "Save",     SaveLife,
195|                 (char*) NULL );
196|     delete lf;
197| }
```

```
198|
199| // --- create a new universe
200| void BuildLife()
201| {
202|    int x = 0, y = 0;
203|    unsigned char lastc = 0;
204|    unsigned char c = 0;
205|    delete lf;
206|    lf = new Life;
207|    gui.ClearScreen();
208|    gui.StatusLine("*, sp, Esc");
209|    while (c != ESC)  {
210|       gui.SetCursor(WD-12, HT);
211|       cout << "x:" << setw(3) << x;
212|       cout << " y:" << setw(3) << y;
213|       cout.flush();
214|       gui.SetCursor(x, y);
215|       c = gui.GetKBChar();
216|       switch (tolower(c))  {
217|          case ' ':
218|          case '\r':
219|             // - toggle a population
220|             if (!lf->isPopulated(x,y)) {
221|                lf->Populate(x,y);
222|                gui.WriteChar(Pop, x, y);
223|             }
224|             else  {
225|                lf->DePopulate(x,y);
226|                gui.WriteChar(' ', x, y);
227|             }
228|             gui.PutBack(lastc);
229|             break;
230|          case UPARROW:
231|             // - up cursor
232|             if (y- == 0)
233|                y = HT-1;
234|             lastc = c;
235|             break;
236|          case DOWNARROW:
237|             // - down cursor
238|             if (++y == HT)
239|                y = 0;
240|             lastc = c;
241|             break;
242|          case LEFTARROW:
```

```
243|             // - left cursor
244|             if (x- == 0)
245|                x = WD-1;
246|             lastc = c;
247|             break;
248|          case RIGHTARROW:
249|             // - right cursor
250|             if (++x == WD)
251|                x = 0;
252|             lastc = c;
253|             break;
254|          default:
255|             break;
256|       }
257|    }
258| }
259|
260| // -- display the universe
261| static void ShowWorld()
262| {
263|    gui.ClearScreen();
264|    if (lf != NULL)
265|       for (int y = 0; y < HT; y++)
266|          for (int x = 0; x < WD; x++)
267|             if (lf->isPopulated(x, y))
268|                gui.WriteChar(Pop, x, y);
269| }
270|
271| // -- display the universe and wait
272| void DisplayLife()
273| {
274|    ShowWorld();
275|    gui.AnyKey();
276| }
277|
278| // --- nonstop evolution
279| void RunLife()
280| {
281|    if (lf != NULL)    {
282|       ShowWorld();
283|       while (lf->isAlive())    {
284|          if (gui.KBCharWaiting())    {
285|             char c = gui.GetKBChar();
286|             if (c == ESC)
287|                break;
```

```
288|          }
289|          lf->Generation();
290|       }
291|    }
292| }
293|
294| // ---- single-step evolution
295| void StepLife()
296| {
297|    if (lf != NULL)   {
298|       ShowWorld();
299|       gui.StatusLine("S-tep or Esc");
300|       int c;
301|       while ((c = gui.GetKBChar()) != ESC)
302|          if (tolower(c) == 's')
303|             lf->Generation();
304|    }
305| }
306|
307| // ---- retrieve a persistent Life object
308| void LoadLife()
309| {
310|    int oa;
311|    gui.UserInput(oa, "Enter Life object #");
312|    delete lf;
313|    lf = new Life(oa);
314|    if (!lf->ObjectExists())
315|       gui.Error("Invalid object #");
316|    else
317|       DisplayLife();
318| }
319|
320| // ---- store a persistent Life object
321| void SaveLife()
322| {
323|    if (lf != NULL)   {
324|       ObjAddr oa = lf->ObjectAddress();
325|       if (oa == 0)   {
326|          lf->AddObject();
327|          oa = lf->ObjectAddress();
328|       }
329|       else
330|          lf->ChangeObject();
331|       // -- force the add/change now
332|       delete lf;
```

```
333|        lf = new Life(oa);
334|        cout << endl << "Life object #: " << oa;
335|        gui.AnyKey();
336|     }
337| }
```

Listing C.8 EMPLOYEE.H

```
  1| // ------- employee.h
  2|
  3| #ifndef EMPLOYEE_H
  4| #define EMPLOYEE_H
  5|
  6| #include "parody.h"
  7| #include "dept.h"
  8|
  9| // ====================================
 10| // EmployeeNo class
 11| // primary key
 12| // ====================================
 13| class EmployeeNo : public Key {
 14|    int emplno;
 15|    int operator> (Key &key)
 16|       { return emplno > ((EmployeeNo &) key).emplno; }
 17|    Key& operator= (Key &key);
 18|    int operator== (Key &key)
 19|       { return emplno == ((EmployeeNo &) key).emplno; }
 20|    void Write(fstream& ndx)
 21|       { ndx.write((char *) &emplno, sizeof emplno); }
 22|    void Read(fstream& ndx)
 23|       { ndx.read((char *) &emplno, sizeof emplno); }
 24|    Key *Make() { return new EmployeeNo(); }
 25|    Bool isNullValue() { return (Bool) (emplno == 0); }
 26|    friend class AssignmentKey;
 27| public:
 28|    EmployeeNo(int en = 0) { emplno = en; }
 29|    int EmplNo() { return emplno; }
 30|    void SetEmplNo(int en) { emplno = en; }
 31| };
 32|
 33| // ====================================
 34| // EmployeeBody class
 35| // single-copy body of the employee object
 36| // ====================================
```

```
37| class EmployeeBody : public Persistent {
38|    friend class Employee;
39|    EmployeeNo emplno;
40|    DepartmentNo deptno;
41|    String name;
42|    void Read();
43|    void Write();
44|    EmployeeBody(int en);
45|    ~EmployeeBody();
46|    void Display();
47|    void Input();
48|    void InputName();
49|    void InputDepartment();
50|    void TryChange();
51|    friend void InputVector();
52|    void SelectChange();
53| };
54|
55| // ====================================
56| // Employee class
57| // multicopy handle of the employee object
58| // ====================================
59| class Employee : public Handle   {
60|    EmployeeBody *emplbody()
61|       { return ((EmployeeBody *)body); }
62| public:
63|    Employee(int en = 0);
64|    ~Employee() {}
65|
66|    int EmplNo()       { return emplbody()->emplno.EmplNo(); }
67|    int DeptNo()       { return emplbody()->deptno.DeptNo(); }
68|    String &Name()            { return emplbody()->name; }
69|    EmployeeNo& EmplKey()     { return emplbody()->emplno; }
70|    DepartmentNo& DeptKey()   { return emplbody()->deptno; }
71|    void SetName(String& nm);
72|    void SetDeptNo(int dn);
73|    // -- user interface functions
74|    void Display()          { emplbody()->Display(); }
75|    void Input()            { emplbody()->Input(); }
76|    void InputName()        { emplbody()->InputName(); }
77|    void InputDepartment()  { emplbody()->InputDepartment(); }
78|    void TryChange()        { emplbody()->TryChange(); }
79|    void SelectChange()     { emplbody()->SelectChange(); }
80|    static void Header();
81|    static Employee *Get();
```

```
82|    static Employee *GetExisting();
83| };
84|
85| inline void Employee::SetName(String& nm)
86| {
87|    emplbody()->name = nm;
88|    emplbody()->ChangeObject();
89| }
90|
91| inline void Employee::SetDeptNo(int dn)
92| {
93|    emplbody()->deptno.SetDeptNo(dn);
94|    emplbody()->ChangeObject();
95| }
96|
97| #endif
```

Listing C.9 EMPLOYEE.CPP

```
 1| // ------- employee.cpp
 2|
 3| #ifdef ZTC
 4| #include <iomanip.hpp>
 5| #else
 6| #include <iomanip.h>
 7| #endif
 8| #include "personel.h"
 9| #include "gui.h"
10|
11| // ====================================
12| // Employee class (Handle)
13| // ====================================
14| Employee::Employee(int en)
15| {
16|    ConstructBody(neweEmployeeBody(en));
17| }
18|
19| // ---- get an existing employee object
20| Employee *Employee::GetExisting()
21| {
22|    Employee *employee = Get();
23|    if (employee != NULL)    {
24|       if (employee->ObjectExists()) {
25|          Header();
```

```
26|          employee->Display();
27|       }
28|       else  {
29|          gui.Error("No such employee on file");
30|          delete employee;
31|          employee = NULL;
32|       }
33|    }
34|    return employee;
35| }
36|
37| // ---- get an employee object
38| Employee *Employee::Get()
39| {
40|    int emplno;
41|    Employee *employee = NULL;
42|    gui.UserInput(emplno,
43|       "Enter employee number (0 to ignore)");
44|    if (emplno != 0)
45|       employee = new Employee(emplno);
46|    return employee;
47| }
48|
49| // --- display employee object header
50| void Employee::Header()
51| {
52|    cout << endl;
53|    cout << "Empl# Employee        Dept#" << endl;
54|    cout << "-- ------- --" << endl;
55|    cout.flush();
56| }
57|
58| // ====================================
59| // EmployeeBody class
60| // ====================================
61|
62| EmployeeBody::EmployeeBody(int en) :
63|       Persistent(*personnel, EMPLOYEE), emplno(en)
64| {
65|    deptno.Relate(DEPARTMENT);
66|    LoadObject();
67| }
68|
69| EmployeeBody::~EmployeeBody()
70| {
```

```
 71|     SaveObject();
 72| }
 73|
 74| // --- employee object assignment
 75| Key& EmployeeNo::operator= (Key &key)
 76| {
 77|     if (this != &key) {
 78|        Key::operator=(key);
 79|        EmployeeNo& empl = (EmployeeNo&) key;
 80|        emplno = empl.emplno;
 81|     }
 82|     return *this;
 83| }
 84|
 85| // --- read a persistent employee object
 86| void EmployeeBody::Read()
 87| {
 88|     int emno, dpno;
 89|     ReadObject(emno);
 90|     emplno.SetEmplNo(emno);
 91|     ReadObject(name);
 92|     ReadObject(dpno);
 93|     deptno.SetDeptNo(dpno);
 94| }
 95|
 96| // --- write a persistent employee object
 97| void EmployeeBody::Write()
 98| {
 99|     WriteObject(emplno.EmplNo());
100|     WriteObject(name);
101|     WriteObject(deptno.DeptNo());
102| }
103|
104| // ----- display an employee object
105| void EmployeeBody::Display()
106| {
107|     cout.width(5);
108|     cout << emplno.EmplNo();
109|     cout.width(0);
110|     cout << ' ';
111|     cout.setf(ios::left, ios::adjustfield);
112|     cout.width(15);
113|     cout << name;
114|     cout.width(0);
115|     cout.setf(ios::right, ios::adjustfield);
```

```
116|    cout << ' ';
117|    cout.width(5);
118|    cout << deptno.DeptNo() << endl;
119|    cout.width(0);
120|    cout.flush();
121| }
122|
123| // -- try to change an employee object
124| void EmployeeBody::TryChange()
125| {
126|    if (!ChangeObject())
127|       gui.Error("Change disallowed");
128| }
129|
130| // --- input employee name from user
131| void EmployeeBody::InputName()
132| {
133|    gui.UserInput(name, "Name", 25);
134|    if (ObjectExists())
135|       TryChange();
136| }
137|
138| // --- input employee department from user
139| void EmployeeBody::InputDepartment()
140| {
141|    int dept;
142|    gui.UserInput(dept, "Department No");
143|    deptno.SetDeptNo(dept);
144|    if (ObjectExists())
145|       TryChange();
146| }
147|
148| // ---- employee input
149| void EmployeeBody::Input()
150| {
151|    InputName();
152|    InputDepartment();
153| }
154|
155| static EmployeeBody *This;
156|
157| static void InputVector()
158| {
159|    if (gui.Selection() == 1)
160|       This->InputName();
```

```
161|    else
162|       This->InputDepartment();
163| }
164|
165| // --- select an employee data member to change
166| void EmployeeBody::SelectChange()
167| {
168|    This = this;
169|    gui.UserMenu( "Change Employee",
170|                "Name",       InputVector,
171|                "Department", InputVector,
172|                (char*) NULL );
173| }
```

Listing C.10 DEPT.H

```
 1| // ------- dept.h
 2|
 3| #ifndef DEPARTMENT_H
 4| #define DEPARTMENT_H
 5|
 6| #include "parody.h"
 7|
 8| // ====================================
 9| // DepartmentNo class
10| // department number primary key
11| // ====================================
12| class DepartmentNo : public Key  {
13|    int deptno;
14|    int operator> (Key &key)
15|       { return deptno > ((DepartmentNo &) key).deptno; }
16|    int operator== (Key &key)
17|       { return deptno == ((DepartmentNo &) key).deptno; }
18|    Key& operator= (Key &key);
19|    void Write(fstream& ndx)
20|       { ndx.write((char *) &deptno, sizeof deptno); }
21|    void Read(fstream& ndx)
22|       { ndx.read((char *) &deptno, sizeof deptno); }
23|    Key *Make() { return new DepartmentNo(); }
24|    Bool isNullValue() { return (Bool) (deptno == 0); }
25| public:
26|    DepartmentNo(int dn = 0) { deptno = dn; }
27|    int DeptNo() { return deptno; }
28|    void SetDeptNo(int dn) { deptno = dn; }
```

```
29| };
30|
31| #include "employee.h"
32|
33| // ====================================
34| // Department class
35| // ====================================
36| class Department : public Persistent   {
37|    DepartmentNo deptno;
38|    String name;
39|    EmployeeNo manager;
40|    void Read();
41|    void Write();
42| public:
43|    Department(int dn = 0);
44|    ~Department();
45|    String &Name()  { return name; }
46|    int DeptNo()    { return deptno.DeptNo(); }
47|    void SetName(String& nm);
48|    void SetManager(int mgr);
49|    // -- user interface functions
50|    void Display();
51|    void Input();
52|    void InputName();
53|    void InputManager();
54|    void TryChange();
55|    void SelectChange();
56|    static Department *Get();
57|    static Department *GetExisting();
58|    static void Header();
59| };
60|
61| inline void Department::SetName(String& nm)
62| {
63|    name = nm;
64|    ChangeObject();
65| }
66|
67| inline void Department::SetManager(int mgr)
68| {
69|    manager.SetEmplNo(mgr);
70|    ChangeObject();
71| }
72|
73| #endif
```

Listing C.11 DEPT.CPP

```
 1| // ------ dept.cpp
 2|
 3| #ifdef ZTC
 4| #include <iomanip.hpp>
 5| #else
 6| #include <iomanip.h>
 7| #endif
 8| #include "personel.h"
 9| #include "gui.h"
10|
11| // ====================================
12| // Department class
13| // ====================================
14| Department::Department(int dn) :
15|             Persistent(*personnel, DEPARTMENT), deptno(dn)
16| {
17|    manager.Relate(EMPLOYEE);
18|    LoadObject();
19| }
20|
21| Department::~Department()
22| {
23|    SaveObject();
24| }
25|
26| // --- read a persistent department object
27| void Department::Read()
28| {
29|    int dpno;
30|    ReadObject(dpno);
31|    deptno.SetDeptNo(dpno);
32|    ReadObject(name);
33|    int mgr;
34|    ReadObject(mgr);
35|    manager.SetEmplNo(mgr);
36| }
37|
38| // --- write a persistent department object
39| void Department::Write()
40| {
41|    WriteObject(deptno.DeptNo());
42|    WriteObject(name);
43|    WriteObject(manager.EmplNo());
```

```
44| }
45|
46| // ---- display department object header
47| void Department::Header()
48| {
49|    cout << endl;
50|    cout << "Dept# Department      Mgr#"  << endl;
51|    cout << "-- ------- --" << endl;
52|    cout.flush();
53| }
54|
55| // ---- display department object
56| void Department::Display()
57| {
58|    cout.width(5);
59|    cout << deptno.DeptNo();
60|    cout.width(0);
61|    cout << ' ';
62|    cout.setf(ios::left, ios::adjustfield);
63|    cout.width(15);
64|    cout << name;
65|    cout.width(0);
66|    cout.setf(ios::right, ios::adjustfield);
67|    cout << ' ';
68|    cout.width(5);
69|    cout << manager.EmplNo();
70|    cout.width(0);
71|    cout << endl;
72|    cout.flush();
73| }
74|
75| // ---- department object assignment
76| Key& DepartmentNo::operator= (Key &key)
77| {
78|    if (this != &key) {
79|       Key::operator=(key);
80|       DepartmentNo& dept = (DepartmentNo&) key;
81|       deptno = dept.deptno;
82|    }
83|    return *this;
84| }
85|
86| static Department *This;
87|
88| static void InputVector()
```

```
 89| {
 90|    if (gui.Selection() == 1)
 91|       This->InputName();
 92|    else
 93|       This->InputManager();
 94| }
 95|
 96| // --- select department data member to change
 97| void Department::SelectChange()
 98| {
 99|    This = this;
100|    gui.UserMenu( "Change Department",
101|                "Name",    InputVector,
102|                "Manager", InputVector,
103|                (char*) NULL );
104| }
105|
106| // ---- read department name from user
107| void Department::InputName()
108| {
109|    gui.UserInput(name, "Name", 25);
110|    if (ObjectExists())
111|       TryChange();
112| }
113|
114| // ---- read department manager from user
115| void Department::InputManager()
116| {
117|    cout << "Manager:" << endl;
118|    cout.flush();
119|    Employee *mgr = Employee::GetExisting();
120|    if (mgr != NULL)  {
121|       SetManager(mgr->EmplNo());
122|       if (ObjectExists())
123|          TryChange();
124|       delete mgr;
125|    }
126| }
127|
128| // --- try to change the department object
129| void Department::TryChange()
130| {
131|    if (!ChangeObject())
132|       gui.Error("Change disallowed");
133| }
```

```
134|
135| // ---- department object user input
136| void Department::Input()
137| {
138|    InputName();
139|    InputManager();
140| }
141|
142| // ---- get an existing department object
143| Department *Department::GetExisting()
144| {
145|    Department *department = Get();
146|    if (department != NULL) {
147|       if (department->ObjectExists())  {
148|          Header();
149|          department->Display();
150|       }
151|       else  {
152|          gui.Error("No such department on file");
153|          delete department;
154|          department = NULL;
155|       }
156|    }
157|    return department;
158| }
159|
160| // ---- get a department object
161| Department *Department::Get()
162| {
163|    int deptno;
164|    Department *department = NULL;
165|    gui.UserInput(deptno,
166|       "Enter department number (0 to ignore)");
167|    if (deptno != 0)
168|       department = new Department(deptno);
169|    return department;
170| }
```

Listing C.12 PROJECT.H

```
 1| // ------ project.h
 2|
 3| #ifndef PROJECT_H
 4| #define PROJECT_H
```

```
 5|
 6| #include "parody.h"
 7| #include "employee.h"
 8|
 9| // ====================================
10| // ProjectNo class
11| // primary key
12| // ====================================
13| class ProjectNo : public Key  {
14|    int projno;
15|    int operator> (Key &key)
16|       { return projno > ((ProjectNo &) key).projno; }
17|    int operator== (Key &key)
18|       { return projno == ((ProjectNo &) key).projno; }
19|    Key& operator= (Key &key);
20|    void Write(fstream& ndx)
21|       { ndx.write((char *) &projno, sizeof projno); }
22|    void Read(fstream& ndx)
23|       { ndx.read((char *) &projno, sizeof projno); }
24|    Key *Make() { return new ProjectNo(); }
25|    Bool isNullValue() { return (Bool) (projno == 0); }
26|    friend class AssignmentKey;
27| public:
28|    ProjectNo(int prno = 0) { projno = prno; }
29|    int ProjNo() { return projno; }
30|    void SetProjNo(int prno) { projno = prno; }
31| };
32|
33| // ====================================
34| // Project class
35| // ====================================
36| class Project : public Persistent   {
37|    ProjectNo projno;
38|    String name;
39|    EmployeeNo manager;
40|    int hours_expended;
41|    void Read();
42|    void Write();
43| public:
44|    Project(int prno = 0);
45|    ~Project();
46|    String &Name()            { return name; }
47|    int ProjNo()              { return projno.ProjNo(); }
48|    int Hours()               { return hours_expended;  }
49|    void SetName(String& nm);
```

```
50|    void SetManager(int mgr);
51|    void SetHours(int hrs);
52|    void AddHours(int hrs);
53|    // --- user interface functions
54|    void Display();
55|    void Input();
56|    void InputName();
57|    void InputManager();
58|    void InputHours();
59|    void TryChange();
60|    void SelectChange();
61|    static Project *Get();
62|    static Project *GetExisting();
63|    static void Header();
64| };
65|
66| inline void Project::SetName(String& nm)
67| {
68|    name = nm;
69|    ChangeObject();
70| }
71|
72| inline void Project::SetManager(int mgr)
73| {
74|    manager.SetEmplNo(mgr);
75|    ChangeObject();
76| }
77|
78| inline void Project::SetHours(int hrs)
79| {
80|    hours_expended = hrs;
81|    ChangeObject();
82| }
83|
84| inline void Project::AddHours(int hrs)
85| {
86|    hours_expended += hrs;
87|    ChangeObject();
88| }
89|
90| #endif
```

Listing C.13 PROJECT.CPP

```
 1| // ------ project.cpp
 2|
 3| #ifdef ZTC
 4| #include <iomanip.hpp>
 5| #else
 6| #include <iomanip.h>
 7| #endif
 8| #include "personel.h"
 9| #include "gui.h"
10|
11| // ====================================
12| // ProjectNo class
13| // ====================================
14|
15| Key& ProjectNo::operator= (Key &key)
16| {
17|    if (this != &key) {
18|       Key::operator=(key);
19|       ProjectNo& proj = (ProjectNo&) key;
20|       projno = proj.projno;
21|    }
22|    return *this;
23| }
24|
25| // ====================================
26| // Project class
27| // ====================================
28|
29| Project::Project(int prno) :
30|             Persistent(*personnel, PROJECT), projno(prno)
31| {
32|    manager.Relate(EMPLOYEE);
33|    hours_expended = 0;
34|    LoadObject();
35| }
36|
37| Project::~Project()
38| {
```

```
39|    SaveObject();
40| }
41|
42| // ---- read a persistent project object
43| void Project::Read()
44| {
45|    int prno;
46|    ReadObject(prno);
47|    projno.SetProjNo(prno);
48|    ReadObject(name);
49|    int mgr;
50|    ReadObject(mgr);
51|    manager.SetEmplNo(mgr);
52|    ReadObject(hours_expended);
53| }
54|
55| // ---- write a persistent project object
56| void Project::Write()
57| {
58|    WriteObject(projno.ProjNo());
59|    WriteObject(name);
60|    WriteObject(manager.EmplNo());
61|    WriteObject(hours_expended);
62| }
63|
64| // ---- display a project object header
65| void Project::Header()
66| {
67|    cout << endl;
68|    cout << "Proj# Project         Mgr#  Hours" << endl;
69|    cout << "-- ------- -- --" << endl;
70|    cout.flush();
71| }
72|
73| // ---- display a project object
74| void Project::Display()
75| {
76|    cout.width(5);
77|    cout << projno.ProjNo();
78|    cout.width(0);
79|    cout << ' ';
80|    cout.setf(ios::left, ios::adjustfield);
81|    cout.width(15);
82|    cout << name;
83|    cout.setf(ios::right, ios::adjustfield);
```

```
 84|    cout.width(0);
 85|    cout << ' ';
 86|    cout.width(5);
 87|    cout << manager.EmplNo();
 88|    cout.width(0);
 89|    cout << ' ';
 90|    cout.width(5);
 91|    cout << hours_expended;
 92|    cout.width(0);
 93|    cout << endl;
 94|    cout.flush();
 95| }
 96|
 97| static Project *This;
 98|
 99| static void InputVector()
100| {
101|    switch (gui.Selection())    {
102|       case 1:
103|          This->InputName();
104|          break;
105|       case 2:
106|          This->InputManager();
107|          break;
108|       case 3:
109|          This->InputHours();
110|          break;
111|       default:
112|          break;
113|    }
114| }
115|
116| // -- select a project data member to change
117| void Project::SelectChange()
118| {
119|    This = this;
120|    gui.UserMenu( "Change Project",
121|                 "Name",    InputVector,
122|                 "Manager", InputVector,
123|                 "Hours",   InputVector,
124|                 (char*) NULL );
125| }
126|
127| // ---- project object input
128| void Project::Input()
```

```
129| {
130|    InputName();
131|    InputManager();
132|    InputHours();
133| }
134|
135| // -- get the project number from the user
136| void Project::InputName()
137| {
138|    gui.UserInput(name, "Name", 25);
139|    if (ObjectExists())
140|       TryChange();
141| }
142|
143| // -- get the project manager from the user
144| void Project::InputManager()
145| {
146|    cout << "Manager: ";
147|    cout.flush();
148|    Employee *employee = Employee::GetExisting();
149|    if (employee != NULL)    {
150|       SetManager(employee->EmplNo());
151|       delete employee;
152|       if (ObjectExists())
153|          TryChange();
154|    }
155| }
156|
157| // -- get the project hours expended from the user
158| void Project::InputHours()
159| {
160|    gui.UserInput(hours_expended, "Hours expended");
161|    if (ObjectExists())
162|       TryChange();
163| }
164|
165| // --- try to change the project object
166| void Project::TryChange()
167| {
168|    if (!ChangeObject())
169|       gui.Error("Change disallowed");
170| }
171|
172| // ---- get an existing project object
173| Project *Project::GetExisting()
```

```
174| {
175|    Project *project;
176|    if ((project = Get()) != NULL)    {
177|       if (project->ObjectExists())   {
178|          Header();
179|          project->Display();
180|       }
181|       else  {
182|          gui.Error("No such project on file");
183|          delete project;
184|       }
185|    }
186|    return project;
187| }
188|
189| // ---- get a project object
190| Project *Project::Get()
191| {
192|    int projno;
193|    Project *project = NULL;
194|    gui.UserInput(projno,
195|       "Enter project number (0 to ignore)");
196|    if (projno != 0)
197|       project = new Project(projno);
198|    return project;
199| }
```

Listing C.14 ASSIGN.H

```
  1| // --- assign.h
  2|
  3| #ifndef ASSIGN_H
  4| #define ASSIGN_H
  5|
  6| #include "parody.h"
  7| #include "personel.h"
  8|
  9| // ====================================
 10| // AssignmentNo class
 11| // concatenated primary key
 12| // ====================================
 13| class AssignmentKey : public Key {
 14|    EmployeeNo emplno;
 15|    ProjectNo  projno;
```

```
16|    int operator> (Key &key);
17|    int operator== (Key &key);
18|    Key& operator= (Key &key);
19|    void Write(fstream& ndx);
20|    void Read(fstream& ndx);
21|    Key *Make() { return new AssignmentKey(); }
22|    Bool isNullValue()
23|       { return (Bool) (EmplNo() == 0 && ProjNo() == 0); }
24|    friend class Assignment;
25| public:
26|    AssignmentKey(int en = 0, int pn = 0) :
27|                    emplno(en), projno(pn) { /* null */}
28|    int EmplNo()            { return emplno.EmplNo(); }
29|    int ProjNo()            { return projno.ProjNo(); }
30|    void SetEmplNo(int en) { emplno.SetEmplNo(en); }
31|    void SetProjNo(int pn) { projno.SetProjNo(pn); }
32| };
33|
34| // ====================================
35| // Assignment class
36| // employee/project assignment
37| // ====================================
38| class Assignment : public Persistent   {
39|    AssignmentKey assignment;
40|    int hours_expended;
41|    void Read();
42|    void Write();
43| public:
44|    Assignment(int en = 0, int pn = 0);
45|    ~Assignment();
46|    int ProjNo()        { return assignment.projno.ProjNo(); }
47|    int EmplNo()        { return assignment.emplno.EmplNo(); }
48|    ProjectNo& ProjKey()  { return assignment.projno; }
49|    EmployeeNo& EmplKey() { return assignment.emplno; }
50|    int Hours()           { return hours_expended; }
51|    void SetHours(int hrs);
52|    void AddHours(int hrs);
53|    void SetEmplNo(int en) { assignment.SetEmplNo(en); }
54|    void SetProjNo(int pn) { assignment.SetProjNo(pn); }
55|    // -- user interface functions
56|    void Display();
57|    void DetailDisplay();
58|    void Input();
59|    static Assignment *Get();
60|    static Assignment *GetExisting();
```

```
61|    static void Header();
62| };
63|
64| inline void Assignment::SetHours(int hrs)
65| {
66|    hours_expended = hrs;
67|    ChangeObject();
68| }
69| inline void Assignment::AddHours(int hrs)
70| {
71|    hours_expended += hrs;
72|    ChangeObject();
73| }
74|
75| #endif
```

Listing C.15 ASSIGN.CPP

```
 1| // ----- assign.cpp
 2|
 3| #ifdef ZTC
 4| #include <iomanip.hpp>
 5| #else
 6| #include <iomanip.h>
 7| #endif
 8| #include "assign.h"
 9| #include "gui.h"
10|
11| // ---- overloaded >
12| int AssignmentKey::operator>(Key& key)
13| {
14|    AssignmentKey& assgnid = (AssignmentKey&)key;
15|    if (emplno > assgnid.emplno)
16|       return 1;
17|    if (emplno == assgnid.emplno && projno > assgnid.projno)
18|       return 1;
19|    return 0;
20| }
21|
22| // ---- overloaded ==
23| int AssignmentKey::operator==(Key& key)
24| {
25|    AssignmentKey& assgnid = (AssignmentKey&)key;
26|    return (emplno == assgnid.emplno &&
```

```
27|        projno == assgnid.projno);
28| }
29|
30| // ---- overloaded assignment
31| Key& AssignmentKey::operator= (Key &key)
32| {
33|    if (this != &key) {
34|       Key::operator=(key);
35|       AssignmentKey& assgnid = (AssignmentKey&)key;
36|       emplno = assgnid.emplno;
37|       projno = assgnid.projno;
38|    }
39|    return *this;
40| }
41|
42| // --- write the key to the index file
43| void AssignmentKey::Write(fstream& ndx)
44| {
45|    emplno.Write(ndx);
46|    projno.Write(ndx);
47| }
48|
49| // --- read the key from the index file
50| void AssignmentKey::Read(fstream& ndx)
51| {
52|    emplno.Read(ndx);
53|    projno.Read(ndx);
54| }
55|
56| // --- construct from employee number, project number
57| Assignment::Assignment(int en, int pn) :
58|        Persistent(*personnel, ASSIGNMENT), assignment(en, pn)
59| {
60|    hours_expended = 0;
61|    assignment.PrimaryKey();
62|    assignment.emplno.Relate(EMPLOYEE);
63|    assignment.projno.Relate(PROJECT);
64|    LoadObject();
65| }
66|
67| Assignment::~Assignment()
68| {
69|    SaveObject();
70| }
71|
```

```
 72| // ---- read the persistent assignment object
 73| void Assignment::Read()
 74| {
 75|    int emplno, projno;
 76|    ReadObject(emplno);
 77|    ReadObject(projno);
 78|    assignment.SetEmplNo(emplno);
 79|    assignment.SetProjNo(projno);
 80|    ReadObject(hours_expended);
 81| }
 82|
 83| // ---- write the persistent assignment object
 84| void Assignment::Write()
 85| {
 86|    WriteObject(assignment.EmplNo());
 87|    WriteObject(assignment.ProjNo());
 88|    WriteObject(hours_expended);
 89| }
 90|
 91| // ---- display an assignment object header
 92| void Assignment::Header()
 93| {
 94|    cout << endl;
 95|    cout << "Empl# Employee        Proj# "
 96|            "Project         Hours" << endl;
 97|    cout << "-- ------- -- "
 98|            "------- --" << endl;
 99|    cout.flush();
100| }
101|
102| // ---- display an assignment object
103| void Assignment::Display()
104| {
105|    cout <<  "Empl #: "
106|         << setw(5)
107|         << assignment.EmplNo()
108|         << setw(0);
109|    cout << " Proj #: "
110|         << setw(5)
111|         << assignment.ProjNo()
112|         << setw(0);
113|    cout << " Hours: "
114|         << setw(5)
115|         << hours_expended
116|         << setw(0)
```

```
117|         << endl;
118|    cout.flush();
119| }
120|
121| // ---- display an assignment object's details
122| void Assignment::DetailDisplay()
123| {
124|    Employee empl(assignment.EmplNo());
125|    Project proj(assignment.ProjNo());
126|
127|    cout << setw(5) << empl.EmplNo();
128|    cout << setw(0);
129|    cout << ' ';
130|    cout.setf(ios::left, ios::adjustfield);
131|    cout << setw(15) << empl.Name();
132|    cout.setf(ios::right, ios::adjustfield);
133|    cout << setw(0);
134|    cout << ' ';
135|    cout << setw(5) << proj.ProjNo();
136|    cout << setw(0);
137|    cout << ' ';
138|    cout.setf(ios::left, ios::adjustfield);
139|    cout << setw(15) << proj.Name();
140|    cout.setf(ios::right, ios::adjustfield);
141|    cout << setw(0);
142|    cout << ' ';
143|    cout << setw(5) << hours_expended << endl;
144|    cout << setw(0);
145|    cout.flush();
146| }
147|
148| // - get an existing assignment object
149| Assignment *Assignment::GetExisting()
150| {
151|    Assignment *assignment;
152|    if ((assignment = Get()) != NULL)   {
153|       if (assignment->ObjectExists())  {
154|          Header();
155|          assignment->DetailDisplay();
156|       }
157|       else  {
158|          gui.Error("No such assignment on file");
159|          delete assignment;
160|          assignment = NULL;
161|       }
```

```
162|     }
163|     return assignment;
164| }
165|
166| // - get an assignment object
167| Assignment *Assignment::Get()
168| {
169|     Assignment *assignment = NULL;
170|     Employee *employee = Employee::GetExisting();
171|     if (employee != NULL)   {
172|        Project *project = Project::GetExisting();
173|        if (project != NULL) {
174|           assignment = new Assignment(employee->EmplNo(),
175|                    project->ProjNo());
176|           delete project;
177|        }
178|        delete employee;
179|     }
180|     return assignment;
181| }
182|
183| // ---- assignment object input
184| void Assignment::Input()
185| {
186|     gui.UserInput(hours_expended, "Hours expended");
187|     if (ObjectExists())
188|        ChangeObject();
189| }
```

Listing C.16 PERSONEL.H

```
  1| // ------ personel.h
  2|
  3| #ifndef PERSONEL_H
  4| #define PERSONEL_H
  5|
  6| // ============================
  7| // Class identification codes
  8| // ============================
  9| enum ClassIdent {
 10|     EMPLOYEE,
 11|     DEPARTMENT,
 12|     PROJECT,
 13|     ASSIGNMENT
```

```
14| };
15|
16| #include "dept.h"
17| #include "employee.h"
18| #include "project.h"
19| #include "assign.h"
20|
21| // ====================================
22| // Personnel Database
23| // ====================================
24| extern Parody *personnel;
25|
26| #endif
```

Listing C.17 PERSONEL.CPP

```
 1| // ------ personel.cpp
 2|
 3| // ---------------------------
 4| // A Personnel Application
 5| // using PARODY,
 6| // the Persistent, (Almost) Relational Object Database
 7| // management system
 8| // ---------------------------
 9|
10| #include "personel.h"
11| #include "gui.h"
12|
13| // ------ Employee prototypes
14| void QueryEmployees();
15| void AddEmployee();
16| void ChangeEmployee();
17| void DeleteEmployee();
18| void ListEmployees();
19| void ListEmployeeProjects();
20| void CompareEmployees();
21|
22| // ------ Department prototypes
23| void QueryDepartments();
24| void AddDepartment();
25| void ChangeDepartment();
26| void DeleteDepartment();
27| void ListDepartments();
28| void ListDepartmentEmployees();
29| void ListAllDepartments();
```

```
30| void ListOneDepartment();
31|
32| // ----- Project prototypes
33| void QueryProjects();
34| void AddProject();
35| void ChangeProject();
36| void DeleteProject();
37| void ListProjects();
38| void ListProjectEmployees();
39|
40| // ----- Assignment prototypes
41| void QueryAssignments();
42| void AddAssignment();
43| void ChangeAssignment();
44| void DeleteAssignment();
45| void ListAssignments();
46| void PostAssignments();
47|
48| #ifdef BCC
49| extern unsigned _stklen = 8192;
50| #endif
51| #ifdef ZTC
52| unsigned _stack = 8192;
53| #endif
54|
55| // ====================================
56| // Personnel Database
57| // The Employee, Department, Project, and Assignment
58| // classes depend on this personnel pointer.
59| // ====================================
60| Parody *personnel;
61|
62| void main()
63| {
64|    personnel = new Parody("PERSONEL");
65|       // -- empty objects to declare relationships
66|    {
67|       Employee emp1;
68|       Department dept;
69|       Project proj;
70|       Assignment assgn;
71|    }
72|
73|    gui.UserMenu( "Personnel Input",
74|                 "Employees",   QueryEmployees,
```

```
 75|                "Departments", QueryDepartments,
 76|                "Projects",    QueryProjects,
 77|                "Assignments", QueryAssignments,
 78|                (char*) NULL );
 79|    delete personnel;
 80| }
 81|
 82| // ------------
 83| // Employee processing
 84| // ------------
 85| void QueryEmployees()
 86| {
 87|    gui.UserMenu( "Employees",
 88|                "Add",      AddEmployee,
 89|                "Change",   ChangeEmployee,
 90|                "Delete",   DeleteEmployee,
 91|                "List",     ListEmployees,
 92|                "Compare",  CompareEmployees,
 93|                "Projects", ListEmployeeProjects,
 94|                (char*) NULL );
 95| }
 96|
 97| // ------ add employee objects
 98| void AddEmployee()
 99| {
100|    Employee *employee;
101|    while ((employee = Employee::Get()) != NULL) {
102|       if (employee->ObjectExists()) {
103|          Employee::Header();
104|          employee->Display();
105|          gui.Error("Employee already on file");
106|       }
107|       else  {
108|          employee->Input();
109|          if (!employee->AddObject())
110|             gui.Error("Add disallowed");
111|       }
112|       delete employee;
113|    }
114| }
115|
116| // ------ change an existing employee object
117| void ChangeEmployee()
118| {
119|    Employee *employee = Employee::GetExisting();
```

```
120|    if (employee != NULL)   {
121|       gui.AnyKey();
122|       employee->SelectChange();
123|       delete employee;
124|    }
125| }
126|
127| // ------ delete an existing employee object
128| void DeleteEmployee()
129| {
130|    Employee *employee = Employee::GetExisting();
131|    if (employee != NULL)   {
132|       if (gui.YesNo("Delete the employee record?"))
133|          if (!employee->DeleteObject())
134|             gui.Error("Delete disallowed");
135|       delete employee;
136|    }
137| }
138|
139| // ---- list all the employees
140| void ListEmployees()
141| {
142|    Employee emp1;
143|    emp1.FirstObject();
144|    Employee::Header();
145|    while (emp1.ObjectExists())   {
146|       emp1.Display();
147|       emp1.NextObject();
148|    }
149|    gui.AnyKey();
150| }
151|
152| // ---- compare two employees
153| //    (demonstrates multiple-copy persistent objects
154| void CompareEmployees()
155| {
156|    Employee *employee1, *employee2;
157|    cout << "Compare Two Employees";
158|    cout.flush();
159|    if ((employee1 = Employee::GetExisting()) != NULL) {
160|       if ((employee2 = Employee::GetExisting()) != NULL) {
161|          Employee::Header();
162|          employee1->Display();
163|          employee2->Display();
164|          delete employee2;
```

```
165|        }
166|        delete employee1;
167|     }
168|     gui.AnyKey();
169| }
170|
171| // --- list the projects an employee is assigned to
172| void ListEmployeeProjects()
173| {
174|     Employee *employee = Employee::GetExisting();
175|     if (employee != NULL)    {
176|        Project::Header();
177|        Assignment assgn;
178|        assgn.SetEmplNo(employee->EmplNo());
179|        assgn.EmplKey().FindObject();
180|        while (assgn.ObjectExists() &&
181|                 assgn.EmplNo() == employee->EmplNo())   {
182|           Project proj(assgn.ProjNo());
183|           if (proj.ObjectExists())
184|              proj.Display();
185|           assgn.NextObject((Key *)&assgn.EmplKey());
186|        }
187|        delete employee;
188|     }
189|     gui.AnyKey();
190| }
191|
192| // ------------
193| // Department processing
194| // ------------
195| void QueryDepartments()
196| {
197|     gui.UserMenu( "Departments",
198|                 "Add",        AddDepartment,
199|                 "Change",     ChangeDepartment,
200|                 "Delete",     DeleteDepartment,
201|                 "List",       ListDepartments,
202|                 "Employees", ListDepartmentEmployees,
203|                 (char*) NULL );
204| }
205|
206| // ----- add department objects
207| void AddDepartment()
208| {
209|     Department *department;
```

```
210|    while ((department = Department::Get()) != NULL)    {
211|       if (department->ObjectExists())  {
212|          Department::Header();
213|          department->Display();
214|          gui.Error("Department already on file");
215|       }
216|       else  {
217|          department->Input();
218|          if (!department->AddObject())
219|             gui.Error("Add disallowed");
220|       }
221|       delete department;
222|    }
223| }
224|
225| // ----- change a department object
226| void ChangeDepartment()
227| {
228|    Department *department = Department::GetExisting();
229|    if (department != NULL) {
230|       gui.AnyKey();
231|       department->SelectChange();
232|       delete department;
233|    }
234| }
235|
236| // ----- delete a department object
237| void DeleteDepartment()
238| {
239|    Department *department = Department::GetExisting();
240|    if (department != NULL) {
241|       if (gui.YesNo("Delete the department record?"))
242|          if (!department->DeleteObject())
243|             gui.Error("Delete disallowed");
244|       delete department;
245|    }
246| }
247|
248| // ----- list the departments
249| void ListDepartments()
250| {
251|    Department dept;
252|    dept.FirstObject();
253|    Department::Header();
254|    while (dept.ObjectExists())    {
```

```
255|        dept.Display();
256|        dept.NextObject();
257|     }
258|     gui.AnyKey();
259| }
260|
261| // ---- list employees in departments
262| void ListDepartmentEmployees()
263| {
264|     gui.UserMenu( "List Employees by Department",
265|                   "All departments", ListAllDepartments,
266|                   "One department",  ListOneDepartment,
267|                   (char*) NULL );
268|     gui.AnyKey();
269| }
270|
271| // --- list employees in all departments
272| void ListAllDepartments()
273| {
274|     Employee emp1;
275|     Employee::Header();
276|     emp1.DeptKey().FirstObject();
277|     while (emp1.ObjectExists())   {
278|        emp1.Display();
279|        emp1.NextObject((Key *)&emp1.DeptKey());
280|     }
281|     gui.AnyKey();
282| }
283|
284| // --- list employees in a selected department
285| void ListOneDepartment()
286| {
287|     Department *department;
288|     while ((department=Department::GetExisting()) != NULL) {
289|        int deptno = department->DeptNo();
290|        Employee::Header();
291|        Employee emp1;
292|        emp1.SetDeptNo(deptno);
293|        emp1.DeptKey().FindObject();
294|        while (emp1.ObjectExists() && emp1.DeptNo()==deptno) {
295|           emp1.Display();
296|           emp1.NextObject((Key *)&emp1.DeptKey());
297|        }
298|        delete department;
299|     }
```

```
300|    gui.AnyKey();
301| }
302|
303| // ------------
304| // Project processing
305| // ------------
306| void QueryProjects()
307| {
308|    gui.UserMenu( "Projects",
309|                 "Add",       AddProject,
310|                 "Change",    ChangeProject,
311|                 "Delete",    DeleteProject,
312|                 "List",      ListProjects,
313|                 "Employees", ListProjectEmployees,
314|                 (char*) NULL );
315| }
316|
317| // ----- a project objects
318| void AddProject()
319| {
320|    Project *project;
321|    while ((project = Project::Get()) != NULL)   {
322|       if (project->ObjectExists())  {
323|          Project::Header();
324|          project->Display();
325|          gui.Error("Project already on file");
326|       }
327|       else  {
328|          project->Input();
329|          if (!project->AddObject())
330|             gui.Error("Add disallowed");
331|       }
332|       delete project;
333|    }
334| }
335|
336| // ---- change a project object
337| void ChangeProject()
338| {
339|    Project *project = Project::GetExisting();
340|    if (project != NULL) {
341|       gui.AnyKey();
342|       project->SelectChange();
343|       delete project;
344|    }
```

```
345| }
346|
347| // ---- delete a project object
348| void DeleteProject()
349| {
350|    Project *project = Project::GetExisting();
351|    if (project != NULL) {
352|       if (gui.YesNo("Delete the project record?"))
353|          if (!project->DeleteObject())
354|             gui.Error("Delete disallowed");
355|       delete project;
356|    }
357| }
358|
359| // ------- list the project objects
360| void ListProjects()
361| {
362|    Project proj;
363|    proj.FirstObject();
364|    Project::Header();
365|    while (proj.ObjectExists())   {
366|       proj.Display();
367|       proj.NextObject();
368|    }
369|    gui.AnyKey();
370| }
371|
372| // ---- list employees assigned to a project
373| void ListProjectEmployees()
374| {
375|    Project *project = Project::GetExisting();
376|    if (project != NULL) {
377|       Employee::Header();
378|       Assignment assgn;
379|       assgn.SetProjNo(project->ProjNo());
380|       assgn.ProjKey().FindObject();
381|       while (assgn.ObjectExists() &&
382|                assgn.ProjNo() == project->ProjNo())   {
383|          Employee empl(assgn.EmplNo());
384|          if (empl.ObjectExists())
385|             empl.Display();
386|          assgn.NextObject((Key *)&assgn.ProjKey());
387|       }
388|       delete project;
389|    }
```

```
390|    gui.AnyKey();
391| }
392|
393| // -----------
394| // Assignments
395| // The Assignments class is a "connector" file
396| // -----------
397| void QueryAssignments()
398| {
399|    gui.UserMenu( "Assignments",
400|                "Add",        AddAssignment,
401|                "Change",     ChangeAssignment,
402|                "Delete",     DeleteAssignment,
403|                "List",       ListAssignments,
404|                "Post hours",   PostAssignments,
405|                (char*) NULL );
406| }
407|
408| // ----- add an assignment object
409| void AddAssignment()
410| {
411|    for (;;) {
412|       Assignment *assignment = Assignment::Get();
413|       if (assignment == NULL)
414|          break;
415|       if (assignment->ObjectExists())  {
416|          Assignment::Header();
417|          assignment->DetailDisplay();
418|          gui.Error("Assignment already on file");
419|       }
420|       else  {
421|          assignment->Input();
422|          assignment->AddObject();
423|       }
424|       delete assignment;
425|    }
426| }
427|
428| // ------ change an assignment object
429| void ChangeAssignment()
430| {
431|    Assignment *assignment = Assignment::GetExisting();
432|    if (assignment != NULL) {
433|       gui.AnyKey();
434|       assignment->Input();
```

```
435|        delete assignment;
436|     }
437| }
438|
439| // ----- delete an assignment object
440| void DeleteAssignment()
441| {
442|     Assignment *assignment = Assignment::GetExisting();
443|     if (assignment != NULL) {
444|        if (gui.YesNo("Delete the assignment?"))
445|           assignment->DeleteObject();
446|        delete assignment;
447|     }
448| }
449|
450| // -- post hours expended by the employee to the project
451| void PostAssignments()
452| {
453|     Assignment *assignment = Assignment::GetExisting();
454|     if (assignment != NULL) {
455|        int hours;
456|        gui.UserInput(hours, "Hours expended");
457|        if (hours != 0)    {
458|           assignment->AddHours(hours);
459|           Project project(assignment->ProjNo());
460|           if (project.ObjectExists())
461|              project.AddHours(hours);
462|        }
463|        delete assignment;
464|     }
465| }
466|
467| // ---- list the assignment objects
468| void ListAssignments()
469| {
470|     Assignment assgn;
471|     assgn.FirstObject();
472|     Assignment::Header();
473|     while (assgn.ObjectExists())  {
474|        assgn.DetailDisplay();
475|        assgn.NextObject();
476|     }
477|     gui.AnyKey();
478| }
```

Listing C.18 PARODY.H

```
 1| // ---- parody.h
 2|
 3| #ifndef DATABASE_H
 4| #define DATABASE_H
 5|
 6| #ifdef ZTC
 7| #include <fstream.hpp>
 8| #else
 9| #include <fstream.h>
10| #endif
11|
12| enum Bool { False, True };
13|
14| // ============================
15| // min/max macros
16| // ============================
17| inline int max(int a, int b)
18| {
19|    return a > b ? a : b;
20| }
21|
22| inline int min(int a, int b)
23| {
24|    return a < b ? a : b;
25| }
26|
27|
28| #include "strings.h"
29| #include "btree.h"
30|
31| // ============================
32| // Object Address
33| // ============================
34| typedef NodeNbr ObjAddr;
35|
36| // ============================
37| // Persistent Object Header Rcd
38| // ============================
39| struct ObjectHeader  {
40|    int classid;       // invariant: class identification
41|    NodeNbr ndnbr;     // relative node number within object
42|    ObjectHeader(int cid = 0, NodeNbr nd = 0)
43|       { classid = cid; ndnbr = nd; }
```

```
44| };
45|
46| // ============================
47| // the Parody database
48| // ============================
49| class Parody : public FileHeader {
50|    Index index;             // invariant: the b-tree file
51|    LinkedListHead objects; // list of instantiated objects
52|    LinkedListHead btrees;  // the b-trees in the database
53|    ObjAddr rebuildnode;    // node of object being rebuilt
54| public:
55|    Parody(String name);
56|    ~Parody();
57|    Index& IndexFile()          { return index; }
58|    LinkedListHead& Btrees()    { return btrees; }
59|    LinkedListHead& Objects()   { return objects; }
60|    Bool RebuildingIndexes()
61|       { return (Bool) (rebuildnode != 0); }
62|    ObjAddr RebuildNode()       { return rebuildnode; }
63|    void GetObjectHeader(ObjAddr nd, ObjectHeader &objhdr);
64|    void RebuildIndexes(ObjAddr nd);
65| };
66|
67| // ====================================
68| // Persistent object abstract base class
69| // ====================================
70| class Persistent  {
71|    friend class Key;
72|    friend class ObjectList;
73|    ObjectHeader objhdr;
74|    ObjAddr objectaddress;  // Node address for this object
75|    Parody& parody;         // database for this object
76|    int indexcount;         // number of keys in the object
77|    LinkedListHead keys;    // the keys in the object
78|    LinkedListHead orgkeys; // original keys in the object
79|    ObjectList *objectlist; // object list entry this object
80|    Node *node;          // current node for reading/writing
81|    int offset;          // current char position
82|    Bool changed;        // True if user changed the object
83|    Bool deleted;        // True if user deleted the object
84|    Bool newobject;      // True if user is adding the object
85|    // - pointers to associate keys with objects
86|    Persistent *oldthispers;
87|    static Persistent *thispers;
88|    // -- methods used from within Persistent class
```

```
 89|    void ObjectOut();
 90|    void RecordObject();
 91|    void RemoveObject();
 92|    void AddIndexes();
 93|    void DeleteIndexes();
 94|    void UpdateIndexes();
 95|    void PositionNode();
 96|    void ReadObjectHeader();
 97|    void WriteObjectHeader();
 98|    void SearchIndex(Key *key);
 99|    void ReadDataMembers();
100|    Btree *FindIndex(Key *key);
101|    Bool TestRelationships();
102|    void ScanForward(NodeNbr nd);
103|    void ScanBackward(NodeNbr nd);
104| protected:
105|    Persistent(Parody &db, int cid);
106|    // -- derived class methods for reading members
107|    void ReadObject(void *buf, int length);
108|    void ReadObject(String& str);
109|    void ReadObject(int& i)
110|       { ReadObject(&i, sizeof(int)); }
111|    void ReadObject(long& l)
112|       { ReadObject(&l, sizeof(long)); }
113|    void ReadObject(float& f)
114|       { ReadObject(&f, sizeof(float)); }
115|    void ReadObject(double& d)
116|       { ReadObject(&d, sizeof(double)); }
117|    void ReadObject(ObjAddr& o)
118|       { ReadObject(&o, sizeof(ObjAddr)); }
119|    // -- derived class methods for writing members
120|    void WriteObject(void *buf, int length);
121|    void WriteObject(String& str);
122|    void WriteObject(int i)
123|       { WriteObject(&i, sizeof(int)); }
124|    void WriteObject(long l)
125|       { WriteObject(&l, sizeof(long)); }
126|    void WriteObject(float f)
127|       { WriteObject(&f, sizeof(float)); }
128|    void WriteObject(double d)
129|       { WriteObject(&d, sizeof(double)); }
130|    void WriteObject(ObjAddr o)
131|       { WriteObject(&o, sizeof(ObjAddr)); }
132|    // - provided by derived class
133|    virtual void Write() = 0;
```

```
134|    virtual void Read() = 0;
135| public:
136|    virtual ~Persistent();
137|    // -- called from derived class's constructor
138|    void LoadObject(ObjAddr nd = 0);
139|    // -- called from derived class's destructor
140|    void SaveObject();
141|    // ----- handle/copy variables
142|    int count;              // count of handle objects
143|    Persistent *objectcopy; // for multiple instantiations
144|    // - class interface methods for modifying database
145|    Bool AddObject();
146|    Bool ChangeObject();
147|    Bool DeleteObject();
148|    Bool ObjectExists()
149|       { return (Bool) (objectaddress != 0); }
150|    // -- class interface methods for searching database
151|    void FindObject(Key *key);
152|    void FirstObject(Key *key = NULL);
153|    void LastObject(Key *key = NULL);
154|    void NextObject(Key *key = NULL);
155|    void PreviousObject(Key *key = NULL);
156|    ObjAddr ObjectAddress() { return objectaddress; }
157| };
158|
159| // ============================
160| // list of instantiated objects
161| // ============================
162| class ObjectList : public LinkedListEntry {
163|    friend Persistent;
164|    Persistent *object;
165|    ObjectList(LinkedListHead *l1, Persistent *obj) :
166|          LinkedListEntry(l1)
167|       { object = obj; }
168| };
169|
170| // ==================================
171| // Handle class for handle/copy idiom
172| // ==================================
173| class Handle   {
174| protected:
175|    Persistent *body;
176|    Handle() { body = NULL; }
177|    virtual ~Handle();
178|    void ConstructBody(Persistent *pbody);
```

```
179| public:
180|    Handle(Handle& handle);
181|    Handle& operator=(Handle& empl);
182|    // - class interface methods for modifying database
183|    Bool AddObject()      { return body->AddObject(); }
184|    Bool ChangeObject()   { return body->ChangeObject(); }
185|    Bool DeleteObject()   { return body->DeleteObject(); }
186|    Bool ObjectExists()   { return body->ObjectExists(); }
187|    // -- class interface methods for searching database
188|    void FindObject(Key *key)
189|       { body->FindObject(key); }
190|    void FirstObject(Key *key = NULL)
191|       { body->FirstObject(key); }
192|    void LastObject(Key *key = NULL)
193|       { body->LastObject(key); }
194|    void NextObject(Key *key = NULL)
195|       { body->NextObject(key); }
196|    void PreviousObject(Key *key = NULL)
197|       { body->PreviousObject(key); }
198|    ObjAddr ObjectAddress()
199|       { return body->ObjectAddress(); }
200| };
201|
202| #include "key.h"
203|
204| #endif
```

Listing C.19 PARODY.CPP

```
  1| // ----- parody.cpp
  2|
  3| // ========================================
  4| // Parody persistent object member functions
  5| // ========================================
  6|
  7| #ifdef ZTC
  8| #include <new.hpp>
  9| #else
 10| #include <new.h>
 11| #endif
 12| #include <stdlib.h>
 13| #include "parody.h"
 14|
 15| #ifdef MSC
```

```
16| int NoMemory(unsigned int)
17| {
18|    exit(-1);
19|    return 0;
20| }
21| #else
22| void NoMemory()
23| {
24|    exit(-1);
25| }
26| #endif
27|
28| // ======================================
29| // Parody member functions
30| // ======================================
31|
32| // ----- construct a Parody database
33| Parody::Parody(String name) :
34|          FileHeader(name + String(".DAT")),
35|             index(name + String(".NDX"))
36| {
37| #ifdef MSC
38|    _set_new_handler(&NoMemory);
39| #else
40|    set_new_handler(NoMemory);
41| #endif
42|    rebuildnode = 0;
43| }
44|
45| // ----- close the Parody database
46| Parody::~Parody()
47| {
48|    Btree *bt = (Btree *) btrees.FirstListEntry();
49|    while (bt != NULL)   {
50|       delete bt;
51|       bt = (Btree *) btrees.FirstListEntry();
52|    }
53| }
54|
55| // --- read an object header record
56| void Parody::GetObjectHeader(
57|             ObjAddr nd,ObjectHeader &objhdr)
58| {
59|    Node *node = new Node(this, nd);
60|    nfile.read((char *)&objhdr, sizeof(ObjectHeader));
```

```
61|    delete node;
62| }
63|
64| void Parody::RebuildIndexes(ObjAddr nd)
65| {
66|    rebuildnode = nd;
67| }
68|
69| // ========================================
70| // Persistent base class member functions
71| // ========================================
72|
73| Persistent *Persistent::thispers = NULL;
74|
75| // --- constructor
76| Persistent::Persistent(Parody& db, int cid) :
77|                          parody(db), objhdr(cid)
78| {
79|    oldthispers = thispers;
80|    thispers = this;
81|    changed = False;
82|    deleted = False;
83|    newobject = False;
84|    offset = 0;
85|    indexcount = 0;
86|    count = 0;
87|    node = NULL;
88|    objectlist = NULL;
89|    objectcopy = NULL;
90|    objectaddress = 0;
91| }
92|
93| // --- destructor
94| Persistent::~Persistent()
95| {
96|    RemoveObject();
97| }
98|
99| // --- search the collected Btrees for this key's index
100| //        add it if it does not exist
101| Btree *Persistent::FindIndex(Key *key)
102| {
103|    Btree *bt = NULL;
104|    if (key == NULL)
105|       key = (Key *) keys.FirstListEntry();
```

```
106|    if (key != NULL)  {
107|       bt = (Btree *) parody.Btrees().FirstListEntry();
108|       while (bt != NULL)   {
109|          Key *tkey = bt->NullKey();
110|          if (tkey->classid == key->classid &&
111|                tkey->indexno == key->indexno)
112|             break;
113|          bt = (Btree *) (bt->NextListEntry());
114|       }
115|       if (bt == NULL)   {
116|          // - key does not have declared Btree index yet
117|          bt = new Btree(parody.IndexFile(), key);
118|          bt->AppendListEntry(&parody.Btrees());
119|       }
120|    }
121|    return bt;
122| }
123|
124| //  -------- record the object's state
125| void Persistent::RecordObject()
126| {
127|    // -- remove this obj in case it was recorded already
128|    RemoveObject();
129|    // -- put the object's address in a parody list of
130|    //      instantiated objects
131|    objectlist = new ObjectList(&parody.Objects(), this);
132|    // -- make copies of the original keys for later update
133|    Key *key = (Key *) keys.FirstListEntry();
134|    while (key != NULL)  {
135|       Key *ky = key->Make();
136|       *ky = *key;
137|       ky->Key::operator=(*key);
138|       ky->AppendListEntry(&orgkeys);
139|       // - instantiate the index b-tree (if not already)
140|       FindIndex(ky);
141|       key = (Key *) (key->NextListEntry());
142|    }
143| }
144|
145| //  -- remove the record of the object's state
146| void Persistent::RemoveObject()
147| {
148|    // - remove object's address from the parody list
149|    delete objectlist;
150|    objectlist = NULL;
```

```
151|    // -- remove copies of the original keys
152|    Key *ky = (Key *) orgkeys.FirstListEntry();
153|    while (ky != NULL)   {
154|       delete ky;
155|       ky = (Key *) orgkeys.FirstListEntry();
156|    }
157| }
158|
159| // - called from derived constructor after all construction
160| void Persistent::LoadObject(ObjAddr nd)
161| {
162|    thispers = NULL;
163|    objectaddress = nd;
164|    if (parody.RebuildingIndexes())
165|       objectaddress = parody.RebuildNode();
166|    if (objectaddress == 0)
167|       // - position at object's node
168|       SearchIndex((Key *) keys.FirstListEntry());
169|    if (objectaddress != 0) {
170|       // - search for a previous instance of this object
171|       ObjectList *obj =
172|          (ObjectList *) parody.Objects().FirstListEntry();
173|       while (obj != NULL)  {
174|          if (objectaddress == obj->object->objectaddress)  {
175|             // -- object already instantiated
176|             objectcopy = obj->object;
177|             break;
178|          }
179|          obj = (ObjectList *) obj->NextListEntry();
180|       }
181|       if (obj == NULL)  {
182|          PositionNode();
183|          ReadDataMembers();
184|       }
185|    }
186|    thispers = oldthispers;
187| }
188|
189| // --- write the object to the database
190| void Persistent::ObjectOut()
191| {
192|    // - tell object to write its data members
193|    Write();
194|    // - pad the last node
195|    int padding = nodedatalength - offset;
```

```
196|    if (padding)   {
197|       String pads(padding);
198|       parody.Nfile().write(pads, padding);
199|    }
200|    NodeNbr nx = node->NextNode();
201|    node->SetNextNode(0);
202|    delete node;
203|    node = NULL;
204|    // - if node was linked, object got shorter
205|    while (nx != 0)   {
206|       Node nd(&parody, nx);
207|       nx = nd.NextNode();
208|       nd.MarkNodeDeleted();
209|    }
210| }
211|
212| // -- write the object's node header
213| void Persistent::WriteObjectHeader()
214| {
215|    // - write the relative node number and class id
216|    fstream& df = parody.Nfile();
217|    df.write((char *) &objhdr, sizeof(ObjectHeader));
218|    offset = sizeof(ObjectHeader);
219|    objhdr.ndnbr++;
220| }
221|
222| // -- write the object's node header
223| void Persistent::ReadObjectHeader()
224| {
225|    // - write the relative node number and class id
226|    fstream& df = parody.Nfile();
227|    df.read((char *) &objhdr, sizeof(ObjectHeader));
228|    offset = sizeof(ObjectHeader);
229| }
230|
231| // - called from derived destructor before all destruction
232| //        a new or existing object is being saved
233| void Persistent::SaveObject()
234| {
235|    if (parody.RebuildingIndexes())
236|       AddIndexes();
237|    else if (newobject)  {
238|       if (!deleted && ObjectExists())  {
239|          PositionNode();
240|          AddIndexes();
```

```
241|            ObjectOut();
242|            RecordObject();
243|         }
244|      }
245|      else if (deleted || changed)  {
246|         if (!ObjectExists())
247|            return;
248|         // - position the parody file at the object's node
249|         PositionNode();
250|         if (deleted)   {
251|            // - delete the object's nodes from the database
252|            while (node != NULL) {
253|               node->MarkNodeDeleted();
254|               NodeNbr nx = node->NextNode();
255|               delete node;
256|               if (nx)
257|                  node = new Node(&parody, nx);
258|               else
259|                  node = NULL;
260|            }
261|            DeleteIndexes();
262|         }
263|         else  {
264|            // - tell object to write its data members
265|            ObjectOut();
266|            // -- update the object's indexes
267|            UpdateIndexes();
268|            RecordObject();
269|         }
270|      }
271| }
272|
273| // - read one data member of the object from the database
274| void Persistent::ReadObject(void *buf, int length)
275| {
276|      fstream& df = parody.Nfile();
277|      while (node != NULL && length > 0)  {
278|         if (offset == nodedatalength) {
279|            NodeNbr nx = node->NextNode();
280|            delete node;
281|            node = nx ? new Node(&parody, nx) : NULL;
282|            ReadObjectHeader();
283|         }
284|         if (node != NULL) {
285|            int len = min(length, nodedatalength-offset);
```

```
286|            df.read((char *)buf, len);
287|            buf = (char *)buf + len;
288|            offset += len;
289|            length -= len;
290|         }
291|      }
292| }
293|
294| // - write one data member of the object to the database
295| void Persistent::WriteObject(void *buf, int length)
296| {
297|      fstream& df = parody.Nfile();
298|      while (node != NULL && length > 0)  {
299|         if (offset == nodedatalength) {
300|            NodeNbr nx = node->NextNode();
301|            if (nx == 0)
302|               nx = parody.NewNode();
303|            node->SetNextNode(nx);
304|            delete node;
305|            node = new Node(&parody, nx);
306|            WriteObjectHeader();
307|         }
308|         int len = min(length, nodedatalength-offset);
309|         df.write((char *)buf, len);
310|         buf = (char *)buf + len;
311|         offset += len;
312|         length -= len;
313|      }
314| }
315|
316| // ------ read a string
317| void Persistent::ReadObject(String& str)
318| {
319|      int len;
320|      ReadObject(&len, sizeof(int));
321|      String s(len);
322|      ReadObject((char *)s, len);
323|      str = s;
324| }
325|
326| // ------ write a string
327| void Persistent::WriteObject(String& str)
328| {
329|      int len = str.Strlen();
330|      WriteObject(&len, sizeof(int));
```

```
331|    WriteObject((char *)str, len);
332| }
333|
334| // -- add the index values to the object's index btrees
335| void Persistent::AddIndexes()
336| {
337|    Key *key = (Key *) keys.FirstListEntry();
338|    while (key != NULL)  {
339|       Btree *bt = FindIndex(key);
340|       key->fileaddr = objectaddress;
341|       bt->Insert(key);
342|       key = (Key *) (key->NextListEntry());
343|    }
344| }
345|
346| // -- update the index values in the object's index btrees
347| void Persistent::UpdateIndexes()
348| {
349|    Key *oky = (Key *) orgkeys.FirstListEntry();
350|    Key *key = (Key *) keys.FirstListEntry();
351|    while (key != NULL)  {
352|       if (!(*oky == *key)) {
353|          // - key value has changed, update the index
354|          // - delete the old
355|          Btree *bt = FindIndex(oky);
356|          oky->fileaddr = objectaddress;
357|          bt->Delete(oky);
358|          // - insert the new
359|          key->fileaddr = objectaddress;
360|          bt->Insert(key);
361|       }
362|       oky = (Key *) (oky->NextListEntry());
363|       key = (Key *) (key->NextListEntry());
364|    }
365| }
366|
367| // - delete the index values from the object's index btrees
368| void Persistent::DeleteIndexes()
369| {
370|    Key *key = (Key *) orgkeys.FirstListEntry();
371|    while (key != NULL)  {
372|       Btree *bt = FindIndex(key);
373|       key->fileaddr = objectaddress;
374|       bt->Delete(key);
375|       key = (Key *) (key->NextListEntry());
```

```
376|    }
377| }
378|
379| // -- position the file to the specifed node number
380| void Persistent::PositionNode()
381| {
382|    if (objectaddress)   {
383|       node = new Node(&parody, (NodeNbr) objectaddress);
384|       offset = sizeof(ObjectHeader);
385|       fstream& nf = parody.Nfile();
386|       ObjectHeader oh;
387|       nf.read((char *)&oh, sizeof(ObjectHeader));
388|       nf.seekp(nf.tellg());
389|       if (oh.classid != objhdr.classid || oh.ndnbr != 0)
390|          objectaddress = 0;
391|    }
392| }
393|
394| // --- search the index for a match on the key
395| void Persistent::SearchIndex(Key *key)
396| {
397|    objectaddress = 0;
398|    if (key != NULL && !key->isNullValue())   {
399|       Btree *bt = FindIndex(key);
400|       if (bt != NULL && bt->Find(key)) {
401|          if (key->indexno != 0)  {
402|             Key *bc;
403|             do
404|                bc = bt->Previous();
405|             while (bc != NULL && *bc == *key);
406|             key = bt->Next();
407|          }
408|          objectaddress = key->fileaddr;
409|       }
410|    }
411| }
412|
413| // ---- find an object by a key value
414| void Persistent::FindObject(Key *key)
415| {
416|    SearchIndex(key);
417|    PositionNode();
418|    ReadDataMembers();
419| }
420|
```

```
421| // - scan nodes forward to the first one of next object
422| void Persistent::ScanForward(NodeNbr nd)
423| {
424|    ObjectHeader oh;
425|    while (++nd < parody.HighestNode()) {
426|       parody.GetObjectHeader(nd, oh);
427|       if (oh.classid == objhdr.classid && oh.ndnbr == 0) {
428|          objectaddress = nd;
429|          break;
430|       }
431|    }
432| }
433|
434| // - scan nodes back to first one of the previous object
435| void Persistent::ScanBackward(NodeNbr nd)
436| {
437|    ObjectHeader oh;
438|    while (-nd > 0)  {
439|       parody.GetObjectHeader(nd, oh);
440|       if (oh.classid == objhdr.classid && oh.ndnbr == 0) {
441|          objectaddress = nd;
442|          break;
443|       }
444|    }
445| }
446|
447| // - retrieve the first object in a key sequence
448| void Persistent::FirstObject(Key *key)
449| {
450|    objectaddress = 0;
451|    Btree *bt = FindIndex(key);
452|    if (bt == NULL)
453|       // -- keyless object
454|       ScanForward(0);
455|    else if ((key = bt->First()) != NULL)
456|       objectaddress = key->fileaddr;
457|    if (objectaddress != 0) {
458|       PositionNode();
459|       ReadDataMembers();
460|    }
461| }
462|
463| // - retrieve the last object in a key sequence
464| void Persistent::LastObject(Key *key)
465| {
```

```
466|    objectaddress = 0;
467|    Btree *bt = FindIndex(key);
468|    if (bt == NULL)
469|       // -- keyless object
470|       ScanBackward(parody.HighestNode());
471|    else if ((key = bt->Last()) != NULL)
472|       objectaddress = key->fileaddr;
473|    if (objectaddress != 0) {
474|       PositionNode();
475|       ReadDataMembers();
476|    }
477| }
478|
479| // - retrieve the next object in a key sequence
480| void Persistent::NextObject(Key *key)
481| {
482|    ObjAddr oa = objectaddress;
483|    objectaddress = 0;
484|    Btree *bt = FindIndex(key);
485|    if (bt == NULL)
486|       // -- keyless object
487|       ScanForward(oa);
488|    else if ((key = bt->Next()) != NULL)
489|       objectaddress = key->fileaddr;
490|    if (objectaddress != 0) {
491|       PositionNode();
492|       ReadDataMembers();
493|    }
494| }
495|
496| // - retrieve the previous object in a key sequence
497| void Persistent::PreviousObject(Key *key)
498| {
499|    ObjAddr oa = objectaddress;
500|    objectaddress = 0;
501|    Btree *bt = FindIndex(key);
502|    if (bt == NULL)
503|       // -- keyless object
504|       ScanBackward(oa);
505|    else if ((key = bt->Previous()) != NULL)
506|       objectaddress = key->fileaddr;
507|    if (objectaddress != 0) {
508|       PositionNode();
509|       ReadDataMembers();
510|    }
```

```
511| }
512|
513| // --- read an object's data members
514| void Persistent::ReadDataMembers()
515| {
516|    if (objectaddress != 0) {
517|       // - tell object to read its data members
518|       Read();
519|       delete node;
520|       node = NULL;
521|       // - put the secondary keys into the table
522|       RecordObject();
523|    }
524| }
525|
526| // ---- add an object to the Parody database
527| Bool Persistent::AddObject()
528| {
529|    newobject =
530|       (Bool) (objectaddress == 0 && TestRelationships());
531|    if (newobject) {
532|       delete node;  // (just in case)
533|       node = new Node(&parody, parody.NewNode());
534|       objectaddress = node->GetNodeNbr();
535|       WriteObjectHeader();
536|    }
537|    return newobject;
538| }
539|
540| // ----- mark a persistent object for change
541| Bool Persistent::ChangeObject()
542| {
543|    changed = TestRelationships();
544|    return changed;
545| }
546|
547| // ----- mark a persistent object for delete
548| Bool Persistent::DeleteObject()
549| {
550|    Key *key = (Key *) keys.FirstListEntry();
551|    Key *ky = key->Make();
552|    *ky = *key;
553|    ky->Key::operator=(*key);
554|
555|    if (!ky->isNullValue()) {
```

```
556|        // - scan for other objects related to this one
557|        Btree *bt = (Btree*) parody.Btrees().FirstListEntry();
558|        while (bt != NULL)    {
559|           Key *tkey = bt->NullKey();
560|           if (tkey->relatedclass == ky->classid) {
561|              ky->Key::operator=(*tkey);
562|              if (bt->Find(ky) == True)   {
563|                 // - another object is related,
564|                 //     so the delete is rejected
565|                 delete ky;
566|                 return False;
567|              }
568|           }
569|           bt = (Btree *) (bt->NextListEntry());
570|        }
571|     }
572|     delete ky;
573|     deleted = True;
574|     return deleted;
575| }
576|
577| // --- test an object's relationships
578| //        return false if it is related to a
579| //        nonexistent object
580| Bool Persistent::TestRelationships()
581| {
582|     Key *key = (Key *) keys.FirstListEntry();
583|     while (key != NULL)   {
584|        Key *ky = key->Make();
585|        *ky = *key;
586|        ky->Key::operator=(*key);
587|        if (!ky->isNullValue()) {
588|           if (ky->relatedclass != -1)    {
589|              ky->indexno = 0;
590|              ky->classid = ky->relatedclass;
591|              ky->fileaddr = 0;
592|              Btree *bt = FindIndex(ky);
593|              if (bt->Find(ky) == False)
594|                 break;
595|           }
596|        }
```

```
597|        delete ky;
598|        key = (Key *) (key->NextListEntry());
599|     }
600|     return (Bool) (key == NULL);
601| }
602|
603| // ====================================
604| // Handle class (Handle)
605| // ====================================
606| Handle::Handle(Handle& handle)
607| {
608|     body = handle.body;
609|     body->count++;
610| }
611|
612| Handle::~Handle()
613| {
614|     if (-body->count == 0)
615|        delete body;
616| }
617|
618| Handle& Handle::operator=(Handle& emp1)
619| {
620|     if (-body->count == 0)
621|        delete body;
622|     body = emp1.body;
623|     body->count++;
624|     return *this;
625| }
626|
627| void Handle::ConstructBody(Persistent *pbody)
628| {
629|     if (pbody->objectcopy != NULL)    {
630|        // - LoadObject found another instantiated
631|        //     copy of the object
632|        body = pbody->objectcopy;
633|        delete pbody;
634|     }
635|     else
636|        body = pbody;
637|     body->count++;
638| }
```

Listing C.20 KEY.H

```
 1| // ------ key.h
 2|
 3| #ifndef KEY_H
 4| #define KEY_H
 5|
 6| #include "parody.h"
 7|
 8| // ============================
 9| // Key abstract base class
10| // ============================
11| class Key : public LinkedListEntry  {
12|    NodeNbr fileaddr;    // object node address -> by this key
13|    NodeNbr lowernode;   // lower node of keys > this key
14|    int classid;         // class id for this key
15|    int indexno;         // 0=primary key, > 0=secondary key
16|    int relatedclass;    // id of class related by this key
17|    Persistent *object;  // object being indexed
18|    virtual Key *Make() = 0;
19|    friend class Btree;
20|    friend class TNode;
21|    friend class Persistent;
22| protected:
23|    Key(NodeNbr fa = 0);
24|    virtual ~Key() { /* null */ }
25|    virtual int operator> (Key& key) = 0;
26|    virtual int operator== (Key& key) = 0;
27|    virtual Key& operator=(Key& key);
28|    virtual void Write(fstream& bfile) = 0;
29|    virtual void Read(fstream& bfile) = 0;
30|    virtual Bool isNullValue() = 0;
31| public:
32|    void PrimaryKey();
33|    void Relate(int cid)  { relatedclass = cid; }
34|    void FindObject()     { object->FindObject(this); }
35|    void FirstObject()    { object->FirstObject(this); }
36|    void LastObject()     { object->LastObject(this); }
37|    void NextObject()     { object->NextObject(this); }
38|    void PreviousObject() { object->PreviousObject(this); }
39| };
40|
41| #endif
```

Listing C.21 KEY.CPP

```
 1| // ------ key.cpp
 2|
 3| #include "parody.h"
 4|
 5| // ==========================================
 6| // base Key class member functions
 7| // ==========================================
 8|
 9| Key::Key(NodeNbr fa) : LinkedListEntry(NULL)
10| {
11|    fileaddr = fa;
12|    lowernode = 0;
13|    relatedclass = -1;
14|    classid = 0;
15|    indexno = 0;
16|    if (Persistent::thispers != NULL)   {
17|       // - register the key with the object being built
18|       object = Persistent::thispers;
19|       AppendListEntry(&object->keys);
20|       classid = object->objhdr.classid;
21|       indexno = object->indexcount++;
22|    }
23| }
24|
25| // --- overloaded =
26| Key& Key::operator=(Key& key)
27| {
28|    if (this != &key) {
29|       fileaddr = key.fileaddr;
30|       lowernode = key.lowernode;
31|       indexno = key.indexno;
32|       classid = key.classid;
33|       relatedclass = key.relatedclass;
34|    }
35|    return *this;
36| }
37|
38| // ---- a key declares itself primary
39| void Key::PrimaryKey()
40| {
41|    if (indexno != 0 && Persistent::thispers != NULL)  {
```

```
42|        DeleteListEntry();
43|        PrependListEntry();
44|        Key *key = (Key *) NextListEntry();
45|        while (key != NULL)  {
46|           if (++(key->indexno) == indexno)
47|              break;
48|           key = (Key *)(key->NextListEntry());
49|        }
50|        indexno = 0;
51|        Persistent::thispers->RecordObject();
52|     }
53| }
```

Listing C.22 BTREE.H

```
 1| // ------ btree.h
 2|
 3| #ifndef BTREE_H
 4| #define BTREE_H
 5|
 6| #ifdef ZTC
 7| #include <fstream.hpp>
 8| #else
 9| #include <fstream.h>
10| #endif
11| #include "linklist.h"
12| #include "node.h"
13|
14| class Key;
15|
16| // ---- b-tree index file
17| class Index : public FileHeader  {
18| public:
19|    Index(String name) : FileHeader(name) {}
20|    Bool NewFile() { return newfile; }
21| };
22|
23| class TNode;
24|
25| // ---- b-tree header record
26| struct TreeHeader {
27|    NodeNbr rootnode; // node number of the root
28|    int keylength;    // length of a key in this b-tree
29|    TreeHeader() { rootnode = keylength = 0; }
```

```
30| };
31|
32| // ---- b-tree index
33| class Btree : public LinkedListEntry   {
34|    TreeHeader header;    // btree header
35|    NodeNbr currnode;     // current node number
36|    TNode *trnode;        // -> current node value
37|    Key *nullkey;         // for padding nodes and calling Make
38|    Index& index;         // index file this tree lives in
39|    long headeraddr;      // offset where header lives
40| public:
41|    Btree(Index& ndx, Key *ky);
42|    ~Btree();
43|    void Insert(void *keypointer);
44|    void Delete(void *keypointer);
45|    Bool Find(void *keypointer);
46|    Key *Current();
47|    Key *First();
48|    Key *Last();
49|    Key *Next();
50|    Key *Previous();
51|    void SetKeyLength(int kl) { header.keylength = kl; }
52|    Index& IndexFile()        { return index; }
53|    Key *NullKey()            { return nullkey; }
54|    NodeNbr Root()            { return header.rootnode; }
55|    NodeNbr KeyLength()       { return header.keylength; }
56| };
57|
58| // ------ b-tree TNode class
59| class TNode : public Node  {
60|    friend Btree;
61|    struct TNodeHeader   {
62|       Bool isleaf;         // true if node is a leaf
63|       NodeNbr parent;      // parent to this node
64|       NodeNbr leftsibling; // left sibling node
65|       NodeNbr rightsibling;// right sibling node
66|       int keycount;        // number of keys in this node
67|       NodeNbr lowernode;   // lower node associated with
68|                            // keys < keys in this node
69|       TNodeHeader()
70|          { isleaf = False; parent = leftsibling =
71|             rightsibling = keycount = lowernode = 0; }
72|    } header;
73|    Key *currkey;           // current key
74|    Btree *btree;           // btree that owns this node
```

```
75|    LinkedListHead keys;     // the keys in this node
76| public:
77|    TNode(Btree *bt, NodeNbr node);
78|    ~TNode();
79|    Bool SearchNode(Key *keyvalue);
80|    void Insert(Key *keyvalue);
81|    int m();
82|    void WriteKey(Key *thiskey);
83|    void Adopt(NodeNbr node);
84|    void Adoption();
85|    Bool isLeaf()           { return header.isleaf; }
86|    NodeNbr Parent()        { return header.parent; }
87|    NodeNbr LeftSibling()   { return header.leftsibling; }
88|    NodeNbr RightSibling()  { return header.rightsibling; }
89|    int KeyCount()          { return header.keycount; }
90|    NodeNbr LowerNode()     { return header.lowernode; }
91|    Bool Redistribute(NodeNbr sib);
92|    Bool Implode(TNode &right);
93|    int NodeHeaderSize()
94|       { return sizeof(TNodeHeader)+Node::NodeHeaderSize(); }
95|    TNode& operator=(TNode &tnode);
96| };
97|
98| #endif
```

Listing C.23 BTREE.CPP

```
 1| // -------- btree.cpp
 2|
 3| #include <string.h>
 4| #include "parody.h"
 5|
 6| // ----- constructor to open a btree
 7| Btree::Btree(Index& ndx, Key *ky) :
 8|             index(ndx), LinkedListEntry(NULL)
 9| {
10|    currnode = 0;
11|    trnode = NULL;
12|    nullkey = ky->Make();
13|    nullkey->Key::operator=(*ky);
14|    NodeNbr nd = 0;
15|    int clno = ky->classid;
16|
17|    if (!index.NewFile())   {
```

```
18|        Node node(&index, 1);
19|        // --- locate the class header
20|        while (clno)   {
21|           NodeNbr nx = node.NextNode();
22|           if (nx == 0)   {
23|              // need to link this one with new one(s) added
24|              nd = index.NewNode();
25|              node.SetNextNode(nd);
26|              break;
27|           }
28|           node = Node(&index, nx);
29|           -clno;
30|        }
31|        if (clno == 0) {
32|           // --- position to the tree header
33|           index.Nfile().seekg((long)
34|              (ky->indexno * sizeof(TreeHeader)), ios::cur);
35|           headeraddr = index.Nfile().tellp();
36|           // --- read the btree header
37|           index.Nfile().read((char *)
38|                    &header, sizeof(TreeHeader));
39|           return;
40|        }
41|     }
42|     // --- build the class headers for new node or class
43|     index.ResetNewFile();
44|     if (nd == 0)
45|        nd = index.NewNode();
46|     else
47|        -clno;
48|     TreeHeader th;
49|     const int treect = nodedatalength / sizeof(TreeHeader);
50|     while (clno)   {
51|        Node node(&index, nd);
52|        for (int i = 0; i < treect; i++)
53|           index.Nfile().write((char*)&th,sizeof(TreeHeader));
54|        nd = index.NewNode();
55|        node.SetNextNode(nd);
56|        -clno;
57|     }
58|
59|     Node node(&index, nd);
60|     // --- position to the tree header
61|     for (int i = 0; i < ky->indexno; i++)
62|        index.Nfile().write((char *)&th, sizeof(TreeHeader));
```

```
 63|    headeraddr = index.Nfile().tellp();
 64|    // --- write the btree header
 65|    index.Nfile().write((char *)&header, sizeof(TreeHeader));
 66|    // --- pad out the rest of the node
 67|    for (i++; i < treect; i++)
 68|       index.Nfile().write((char *)&th, sizeof(TreeHeader));
 69|    // -- pad any residual node space
 70|    int residual = nodedatalength-treect*sizeof(TreeHeader);
 71|    if (residual)  {
 72|       char *residue = new char[residual];
 73|       memset(residue, 0, residual);
 74|       index.Nfile().write(residue, residual);
 75|       delete residue;
 76|    }
 77|    node.MarkNodeChanged();
 78| }
 79|
 80| // ----- destructor for a btree
 81| Btree::~Btree()
 82| {
 83|    index.Nfile().seekg(headeraddr);
 84|    index.Nfile().write((char *)&header, sizeof(TreeHeader));
 85|    delete nullkey;
 86| }
 87|
 88| // -------- insert a key into a btree
 89| void Btree::Insert(void *keypointer)
 90| {
 91|    // -- don't insert duplicate keys
 92|    if (!Find(keypointer))  {
 93|
 94|       Key *newkey = ((Key *)keypointer)->Make();
 95|       *newkey = *(Key *)keypointer;
 96|
 97|       NodeNbr rootnode = 0, leftnode = 0, rightnode = 0;
 98|       Bool RootisLeaf = True;
 99|
100|       Bool done = False;
101|       // ---- insert key into btree
102|       while (currnode)  {
103|          int em = trnode->m();
104|          trnode->Insert(newkey);
105|                         // first insertion is into leaf
106|                         // if split, subsequent insertions
107|                         // are into parents (non-leaves)
```

```
108|          if (!trnode->header.isleaf)
109|             trnode->currkey->lowernode = rightnode;
110|
111|          done = (Bool) (trnode->header.keycount <= em);
112|          if (!done)
113|             // -- node is full,
114|             //      try to redistribute keys among siblings
115|             done = trnode->Redistribute(
116|                      trnode->header.leftsibling);
117|          if (!done)
118|             done = trnode->Redistribute(
119|                      trnode->header.rightsibling);
120|
121|          if (done)
122|             break;
123|          // -- cannot redistribute filled node, split it
124|          RootisLeaf = False;
125|
126|          rightnode = index.NewNode();
127|          leftnode = currnode;
128|
129|          TNode right(this, rightnode);
130|          right.SetNodeNbr(rightnode);
131|          right.MarkNodeChanged();
132|
133|          // - establish sibling and parent relationships
134|          //     between current node and new right sibling
135|          right.header.rightsibling =
136|                  trnode->header.rightsibling;
137|          trnode->header.rightsibling = rightnode;
138|          right.header.leftsibling = currnode;
139|          right.header.parent = trnode->header.parent;
140|
141|          // -- if the current node is a leaf,
142|          //        so is the new sibling
143|          right.header.isleaf = trnode->header.isleaf;
144|
145|          // -- compute new key counts for the two nodes
146|          trnode->header.keycount = (em + 1) / 2;
147|          right.header.keycount = em-trnode->header.keycount;
148|
149|          // --- locate the middle key in the current node
150|          Key *middlekey = (Key *)
151|             (trnode->keys.FirstListEntry());
152|          for (int i = 0; i < trnode->header.keycount; i++)
```

```
153|            middlekey = (Key *) middlekey->NextListEntry();
154|
155|         // -- set the pointer to keys less than
156|         //      those in new node
157|         if (!right.header.isleaf)
158|            right.header.lowernode = middlekey->lowernode;
159|
160|         // -- point to the keys to move (1 past middle)
161|         Key *movekey = (Key *) middlekey->NextListEntry();
162|
163|         // -- middle key inserts into parent
164|         middlekey->DeleteListEntry();
165|         *newkey = *middlekey;
166|         delete middlekey;
167|
168|         // -- move keys from current to new right node
169|         for (i = 0; i < right.header.keycount; i++)  {
170|            Key *nkey = (Key *) movekey->NextListEntry();
171|            movekey->DeleteListEntry();
172|            movekey->AppendListEntry(&right.keys);
173|            movekey = nkey;
174|         }
175|
176|         // -- prepare to insert key
177|         //      into parent of split nodes
178|         currnode = trnode->header.parent;
179|         if (!currnode) {
180|            // -- no parent node, splitting the root node
181|            rootnode = index.NewNode();
182|            right.header.parent = rootnode;
183|            trnode->header.parent = rootnode;
184|         }
185|
186|         // - the former right sibling of the current node
187|         //     is now the right sibling of the split node
188|         //     and must record the new node as left sibling
189|
190|         if (right.header.rightsibling)   {
191|            TNode farright(this, right.header.rightsibling);
192|            farright.header.leftsibling = rightnode;
193|            farright.MarkNodeChanged();
194|         }
195|
196|         // - children of the new split node point to
197|         //     the current split node as parent. They must
```

```
198|          //     be adopted by the new split node
199|
200|          if (!right.header.isleaf)
201|             right.Adoption();
202|
203|          // -- if splitting other than root, read parent
204|          //       position currkey to key where split node
205|          //       key will be inserted
206|
207|          if (currnode)  {
208|             delete trnode; // writes the split node to disk
209|             // - get the parent of the split nodes
210|             trnode = new TNode(this, currnode);
211|             // - position currkey where new key will insert
212|             trnode->SearchNode(newkey);
213|          }
214|       }
215|
216|       if (!done)  {
217|          // --- new root node --- */
218|          delete trnode;
219|          if (rootnode == 0)
220|             rootnode = index.NewNode();
221|          trnode = new TNode(this, rootnode);
222|          trnode->header.isleaf = RootisLeaf;
223|          currnode = header.rootnode = rootnode;
224|          trnode->SetNodeNbr(rootnode);
225|          trnode->Insert(newkey);
226|          trnode->header.parent = 0;
227|          trnode->header.keycount = 1;
228|          if (!RootisLeaf)    {
229|             trnode->header.lowernode = leftnode;
230|             trnode->currkey->lowernode = rightnode;
231|          }
232|          trnode->MarkNodeChanged();
233|       }
234|       delete newkey;
235|    }
236|    delete trnode;
237|    trnode = NULL;
238| }
239|
240| // -------- find a key in a btree
241| Bool Btree::Find(void *keypointer)
242| {
```

```
243|    currnode = header.rootnode;
244|    while (currnode)  {
245|       delete trnode;
246|             trnode = new TNode(this, currnode);
247|
248|       if (trnode->SearchNode((Key *)keypointer))   {
249|          // -- search key is equal to a key in the node
250|          ((Key*)keypointer)->fileaddr =
251|                   trnode->currkey->fileaddr;
252|          return True;
253|       }
254|       if (trnode->header.isleaf)
255|          break;
256|       if (trnode->currkey == trnode->keys.FirstListEntry())
257|          // -- search key is < lowest key in node
258|          currnode = trnode->header.lowernode;
259|       else if (trnode->currkey)
260|          // - search key is < current key in node
261|          currnode = ((Key *)
262|             (trnode->currkey->PrevListEntry()))->lowernode;
263|       else
264|          // - search key is > highest key in node
265|          currnode =
266|          ((Key *)(trnode->keys.LastListEntry()))->lowernode;
267|    }
268|    return False;
269| }
270|
271| // -------- delete a key from a btree
272| void Btree::Delete(void *keypointer)
273| {
274|    if (Find(keypointer))   {
275|       if (!trnode->header.isleaf)   {
276|
277|          // - if not found in leaf node, go down to leaf
278|          TNode *leaf =
279|             new TNode(this, trnode->currkey->lowernode);
280|          while (!leaf->header.isleaf)  {
281|             NodeNbr lf = leaf->header.lowernode;
282|             delete leaf;
283|             leaf = new TNode(this, lf);
284|          }
285|
286|          // -- Move the left-most key from the leaf
287|          //    to where deleted key was in higher node
```

```
288|          Key *movekey = (Key *) leaf->keys.FirstListEntry();
289|          movekey->DeleteListEntry();
290|          leaf->header.keycount-;
291|          leaf->MarkNodeChanged();
292|
293|          movekey->InsertListEntry(
294|                trnode->currkey, &trnode->keys);
295|          movekey->lowernode = trnode->currkey->lowernode;
296|          trnode->currkey->DeleteListEntry();
297|          delete trnode->currkey;
298|          trnode->MarkNodeChanged();
299|          delete trnode;
300|
301|          trnode = leaf;
302|          trnode->currkey =
303|             (Key *) trnode->keys.FirstListEntry();
304|          currnode = trnode->GetNodeNbr();
305|       }
306|       else  {
307|          // -- delete the key from the node
308|          trnode->currkey->DeleteListEntry();
309|          delete trnode->currkey;
310|          trnode->header.keycount-;
311|          trnode->MarkNodeChanged();
312|          if (trnode->header.keycount == 0)
313|             header.rootnode = 0;
314|       }
315|       // -- if the node shrinks to half capacity,
316|       //      try to combine it with a sibling node
317|       while (trnode->header.keycount > 0 &&
318|                trnode->header.keycount <= trnode->m() / 2) {
319|          if (trnode->header.rightsibling) {
320|             TNode *right =
321|                new TNode(this, trnode->header.rightsibling);
322|             if (trnode->Implode(*right))  {
323|                delete right;
324|                NodeNbr parent = trnode->header.parent;
325|                if (parent == 0)  {
326|                   header.rootnode = trnode->GetNodeNbr();
327|                   break;
328|                }
329|                delete trnode;
330|                trnode = new TNode(this, parent);
331|                continue;
332|             }
```

```
333|            delete right;
334|         }
335|         if (trnode->header.leftsibling)  {
336|            TNode *left =
337|               new TNode(this, trnode->header.leftsibling);
338|            if (left->Implode(*trnode))   {
339|               delete trnode;
340|               NodeNbr parent = left->header.parent;
341|               if (parent == 0)  {
342|                  header.rootnode = left->GetNodeNbr();
343|                  trnode = left;
344|                  break;
345|               }
346|               delete left;
347|               trnode = new TNode(this, parent);
348|               continue;
349|            }
350|            delete left;
351|         }
352|
353|         // - could not combine with either sibling,
354|         //     try to redistribute
355|         if (!trnode->Redistribute(
356|                  trnode->header.leftsibling))
357|            trnode->Redistribute(
358|                  trnode->header.rightsibling);
359|         break;
360|      }
361|   }
362|   delete trnode;
363|   trnode = NULL;
364| }
365|
366| // --- return the address of the current key
367| Key *Btree::Current()
368| {
369|   if (trnode == NULL)
370|      return NULL;
371|   return trnode->currkey;
372| }
373|
374| // --- return the address of the first key
375| Key *Btree::First()
376| {
377|   currnode = header.rootnode;
```

```
378|    if (currnode)  {
379|       delete trnode;
380|             trnode = new TNode(this, currnode);
381|       while (!trnode->header.isleaf)   {
382|          currnode = trnode->header.lowernode;
383|          delete trnode;
384|          trnode = new TNode(this, currnode);
385|       }
386|       trnode->currkey =
387|          (Key *) (trnode->keys.FirstListEntry());
388|    }
389|    return Current();
390| }
391|
392| // --- return the address of the last key
393| Key *Btree::Last()
394| {
395|    currnode = header.rootnode;
396|    if (currnode)  {
397|       delete trnode;
398|             trnode = new TNode(this, currnode);
399|       while (!trnode->header.isleaf)   {
400|          currnode =
401|             ((Key *)(trnode->keys.LastListEntry()))->
402|                   lowernode;
403|          delete trnode;
404|          trnode = new TNode(this, currnode);
405|       }
406|       trnode->currkey =
407|          (Key *) (trnode->keys.LastListEntry());
408|    }
409|    return Current();
410| }
411|
412| // --- return the address of the next key
413| Key *Btree::Next()
414| {
415|    if (trnode == NULL || trnode->currkey == NULL)
416|       return First();
417|    if (!trnode->header.isleaf)   {
418|       // - current key is not in a leaf
419|       currnode = trnode->currkey->lowernode;
420|       delete trnode;
421|             trnode = new TNode(this, currnode);
422|       // -- go down to the leaf
```

```
423|        while (!trnode->header.isleaf)   {
424|           currnode = trnode->header.lowernode;
425|           delete trnode;
426|                 trnode = new TNode(this, currnode);
427|        }
428|        // -- use the first key in the leaf as the next one
429|        trnode->currkey =
430|           (Key *) (trnode->keys.FirstListEntry());
431|     }
432|     else  {
433|        // --- current key is in a leaf
434|        Key *thiskey = nullkey->Make();
435|        *thiskey = *(trnode->currkey);
436|
437|        // -- point to the next key in the leaf
438|        trnode->currkey =
439|           (Key *) (trnode->currkey->NextListEntry());
440|        while (trnode->currkey == NULL &&
441|                    currnode != header.rootnode)  {
442|           // - current key was the last one in the leaf
443|           TNode pnode(this, trnode->Parent());
444|           pnode.SearchNode(thiskey);
445|           currnode = pnode.GetNodeNbr();
446|           *trnode = pnode;
447|        }
448|        delete thiskey;
449|     }
450|     return Current();
451| }
452|
453| // --- return the address of the previous key
454| Key *Btree::Previous()
455| {
456|     if (trnode == NULL || trnode->currkey == NULL)
457|        return Last();
458|     if (!trnode->header.isleaf)   {
459|        // - current key is not in a leaf
460|        Key *ky = (Key *) (trnode->currkey->PrevListEntry());
461|        if (ky != NULL)
462|           currnode = ky->lowernode;
463|        else
464|           currnode = trnode->header.lowernode;
465|        delete trnode;
466|              trnode = new TNode(this, currnode);
467|        // -- go down to the leaf
```

```
468|        while (!trnode->header.isleaf)    {
469|           currnode =
470|              ((Key *)
471|              (trnode->keys.LastListEntry()))->lowernode;
472|           delete trnode;
473|                 trnode = new TNode(this, currnode);
474|        }
475|        // -- use the last key in the leaf as the next one
476|        trnode->currkey =
477|           (Key *) (trnode->keys.LastListEntry());
478|     }
479|     else  {
480|        // --- current key is in a leaf
481|        Key *thiskey = nullkey->Make();
482|        *thiskey = *(trnode->currkey);
483|
484|        // -- point to the previous key in the leaf
485|        trnode->currkey =
486|           (Key *) (trnode->currkey->PrevListEntry());
487|        while (trnode->currkey == NULL &&
488|                    currnode != header.rootnode)  {
489|           // - current key was the first one in the leaf.
490|           TNode pnode(this, trnode->Parent());
491|           pnode.SearchNode(thiskey);
492|           if (pnode.currkey == NULL)
493|              pnode.currkey =
494|                 (Key *) pnode.keys.LastListEntry();
495|           else
496|              pnode.currkey =
497|              (Key *) (pnode.currkey->PrevListEntry());
498|           currnode = pnode.GetNodeNbr();
499|           *trnode = pnode;
500|        }
501|        delete thiskey;
502|     }
503|     return Current();
504| }
```

Listing C.24 TNODE.CPP

```
  1| // ------ tnode.cpp
  2|
  3| // ==============================
  4| // B-tree Tnode class
```

```
 5| // ==============================
 6|
 7| #include "parody.h"
 8|
 9| TNode::TNode(Btree *bt, NodeNbr nd) :
10|                        Node(&(bt->IndexFile()), nd)
11| {
12|    btree = bt;
13|    currkey = NULL;
14|    fstream& nf = btree->IndexFile().Nfile();
15|    // --- read the header
16|    long nad = NodeAddress() + Node::NodeHeaderSize();
17|    nf.seekg(nad);
18|    nf.read((char *) &header, sizeof(TNodeHeader));
19|    if (nf.eof() || nf.fail()) {
20|       nf.clear();
21|       // -- appending a new node
22|       nf.seekp(nad);
23|       nf.write((char *) &header, sizeof(TNodeHeader));
24|    }
25|    else  {
26|       // -- reading an existing node, read the keys
27|       for (int i = 0; i < header.keycount; i++) {
28|          // -- get memory for and read a key
29|          Key *thiskey = btree->NullKey()->Make();
30|          thiskey->Key::operator=(*btree->NullKey());
31|          thiskey->Read(nf);
32|
33|          // -- read the key's file address
34|          NodeNbr fa;
35|          nf.read((char *)&fa, sizeof(NodeNbr));
36|          thiskey->fileaddr = fa;
37|
38|          if (!header.isleaf)  {
39|             NodeNbr lnode;
40|             nf.read((char *)&lnode, sizeof(NodeNbr));
41|             thiskey->lowernode = lnode;
42|          }
43|          thiskey->AppendListEntry(&keys);
44|       }
45|    }
46| }
47|
48| // ---- write a key to the node's disk record
49| void TNode::WriteKey(Key *thiskey)
```

```
50| {
51|    long kl;
52|    fstream& nf = btree->IndexFile().Nfile();
53|    if (btree->KeyLength() == 0)
54|       kl = nf.tellp();
55|    // ---- write the key value
56|    thiskey->Write(nf);
57|    // -- write the key's file address
58|    NodeNbr fa = thiskey->fileaddr;
59|    nf.write((char *)&fa, sizeof(NodeNbr));
60|    // --- post the key length in the btree header
61|    if (btree->KeyLength() == 0)  {
62|       long km = nf.tellp();
63|       btree->SetKeyLength((int) (km - kl));
64|    }
65|    if (!header.isleaf)  {
66|       // - write the lower node pointer for non-leaf keys
67|       NodeNbr lnode = thiskey->lowernode;
68|       nf.write((char *)&lnode, sizeof(NodeNbr));
69|    }
70| }
71|
72| TNode::~TNode()
73| {
74|    if (header.keycount == 0)
75|       // -- this node is to be deleted
76|       deletenode = True;
77|    else  {
78|       fstream& nf = btree->IndexFile().Nfile();
79|       if (nodechanged)  {
80|          long nad = NodeAddress() + Node::NodeHeaderSize();
81|          nf.seekp(nad);
82|          // --- write the node header
83|          nf.write((char *) &header, sizeof(TNodeHeader));
84|       }
85|       // --- write the keys
86|       Key *thiskey = (Key *) keys.FirstListEntry();
87|       while (thiskey != NULL) {
88|          if (nodechanged)
89|             WriteKey(thiskey);
90|          Key *nkey = (Key *) (thiskey->NextListEntry());
91|          delete thiskey;
92|          thiskey = nkey;
93|       }
94|       if (nodechanged)  {
```

```
 95|          // --- pad the node
 96|          int em = m();
 97|          for (int i = header.keycount; i < em; i++)
 98|             WriteKey(btree->NullKey());
 99|
100|          int nodesize =
101|             NodeHeaderSize() + em * btree->KeyLength();
102|          if (!header.isleaf)
103|             nodesize += em * sizeof(NodeNbr);
104|          int residual = nodelength - nodesize;
105|          if (residual > 0) {
106|             char *fill = new char[residual];
107|             memset(fill, 0, residual);
108|             nf.write(fill, residual);
109|             delete fill;
110|          }
111|       }
112|    }
113| }
114|
115| // --- assignment operator
116| TNode& TNode::operator=(TNode &tnode)
117| {
118|    Key *thiskey;
119|    // -- if receiver has any keys, delete them
120|    while (header.keycount > 0)   {
121|       thiskey = (Key *) keys.FirstListEntry();
122|       thiskey->DeleteListEntry();
123|       delete thiskey;
124|       -header.keycount;
125|    }
126|    Node::operator=(tnode);
127|    header = tnode.header;
128|    currkey = NULL;
129|    // --- copy the keys
130|    thiskey = (Key *) tnode.keys.FirstListEntry();
131|    while (thiskey != NULL) {
132|       Key *newkey = btree->NullKey()->Make();
133|       *newkey = *thiskey;
134|       newkey->AppendListEntry(&keys);
135|       if (thiskey == tnode.currkey)
136|          currkey = newkey;
137|       thiskey = (Key *) (thiskey->NextListEntry());
138|    }
139|    return *this;
```

```
140| }
141|
142| // ---- compute m value of node
143| int TNode::m()
144| {
145|    int keyspace = nodelength - NodeHeaderSize();
146|    int keylen = btree->KeyLength();
147|    if (!header.isleaf)
148|       keylen += sizeof(NodeNbr);
149|    return keyspace / keylen;
150| }
151|
152| // ----- search a node for a match on a key
153| Bool TNode::SearchNode(Key *keyvalue)
154| {
155|    currkey = (Key *) keys.FirstListEntry();
156|    while (currkey != NULL) {
157|       if (*currkey > *keyvalue)
158|          break;
159|       if (*currkey == *keyvalue) {
160|          if (keyvalue->indexno == 0)
161|             return True;
162|          if (currkey->fileaddr == keyvalue->fileaddr)
163|             return True;
164|          if (keyvalue->fileaddr == 0)
165|             return True;
166|          if (currkey->fileaddr > keyvalue->fileaddr)
167|             break;
168|       }
169|       currkey = (Key *) (currkey->NextListEntry());
170|    }
171|    return False;
172| }
173|
174| void TNode::Insert(Key *keyvalue)
175| {
176|    // ---- insert the new key
177|    Key *ky = keyvalue->Make();
178|    *ky = *keyvalue;
179|    if (currkey == NULL)
180|       ky->AppendListEntry(&keys);
181|    else
182|       ky->InsertListEntry(currkey, &keys);
183|    header.keycount++;
184|    nodechanged = True;
```

```
185|    currkey = ky;
186| }
187|
188| // -- a node "adopts" all its children by telling
189| //      them to point to it as their parent
190| void TNode::Adoption()
191| {
192|    Adopt(header.lowernode);
193|    Key *thiskey = (Key *) keys.FirstListEntry();
194|    for (int i = 0; i < header.keycount; i++) {
195|       Adopt(thiskey->lowernode);
196|       thiskey = (Key *) (thiskey->NextListEntry());
197|    }
198| }
199|
200| // - adopt a child node
201| void TNode::Adopt(NodeNbr node)
202| {
203|    if (node)   {
204|       TNode nd(btree, node);
205|       nd.header.parent = nodenbr;
206|       nd.nodechanged = True;
207|    }
208| }
209|
210| // -- redistribute keys among two sibling nodes
211| Bool TNode::Redistribute(NodeNbr sib)
212| {
213|    if (sib == 0)
214|       return False;
215|    TNode sibling(btree, sib);
216|    if (sibling.header.keycount >= sibling.m() ||
217|          sibling.header.parent != header.parent)
218|       return False;
219|    // -- assign left and right associations
220|    TNode *left, *right;
221|    if (sib == header.leftsibling)   {
222|       right = this;
223|       left = &sibling;
224|    }
225|    else  {
226|       right = &sibling;
227|       left = this;
228|    }
229|    // --- compute number of keys to be in left node
```

```
230|    int leftct =
231|       (left->header.keycount + right->header.keycount) / 2;
232|    // --- if no redistribution would occur
233|    if (leftct == left->header.keycount)
234|       return False;
235|    // --- compute number of keys to be in right node
236|    int rightct =
237|       (left->header.keycount+right->header.keycount)-leftct;
238|    // --- get the parent
239|    TNode parent(btree, left->header.parent);
240|    // - position parent's currkey
241|    //     to one that points to siblings
242|    parent.SearchNode((Key *)(left->keys.FirstListEntry()));
243|    // -- will move keys from left to right or right to
244|    //     left depending on which node has the greater
245|    //     number of keys to start with.
246|    if (left->header.keycount < right->header.keycount)   {
247|       // -- moving keys from right to left
248|       int mvkeys = right->header.keycount - rightct - 1;
249|       // -- move key from parent to end of left node
250|       left->currkey = parent.currkey;
251|       parent.currkey =
252|          (Key *) parent.currkey->NextListEntry();
253|       left->currkey->DeleteListEntry();
254|       left->currkey->AppendListEntry(&left->keys);
255|       if (!left->header.isleaf)
256|          left->currkey->lowernode = right->header.lowernode;
257|       // - point to the keys to move
258|       //     (at front of right node)
259|       Key *movekey = (Key *) right->keys.FirstListEntry();
260|       // -- move keys from right to left node
261|       for (int i = 0; i < mvkeys; i++) {
262|          Key *nkey = (Key *) movekey->NextListEntry();
263|          movekey->DeleteListEntry();
264|          movekey->AppendListEntry(&left->keys);
265|          movekey = nkey;
266|       }
267|       // - move separating key from right node to parent
268|       movekey->DeleteListEntry();
269|       movekey->InsertListEntry(parent.currkey,&parent.keys);
270|       if (!right->header.isleaf)
271|          right->header.lowernode = movekey->lowernode;
272|       movekey->lowernode = right->nodenbr;
273|       right->header.keycount = rightct;
274|       left->header.keycount  = leftct;
```

```
275|        if (!left->header.isleaf)
276|           left->Adoption();
277|     }
278|     else  {
279|        // ---- moving from left to right
280|        int mvkeys = left->header.keycount - leftct - 1;
281|        // -- move key from parent to right node
282|        right->currkey = parent.currkey;
283|        parent.currkey =
284|           (Key *) parent.currkey->NextListEntry();
285|        right->currkey->DeleteListEntry();
286|        right->currkey->InsertListEntry(
287|                 right->keys.FirstListEntry(), &right->keys);
288|        if (!right->header.isleaf)
289|           right->currkey->lowernode=right->header.lowernode;
290|        // -- locate the first key to move in the left node
291|        Key *movekey = (Key *) (left->keys.FirstListEntry());
292|        for (int i = 0; i < leftct; i++)
293|           movekey = (Key *) movekey->NextListEntry();
294|        // -- move key from left node up to parent
295|        Key *nkey = (Key *) movekey->NextListEntry();
296|        movekey->DeleteListEntry();
297|        movekey->InsertListEntry(parent.currkey,&parent.keys);
298|        right->header.lowernode = movekey->lowernode;
299|        movekey->lowernode = right->nodenbr;
300|        movekey = nkey;
301|        // - move keys from the left node to the right node
302|        Key *inskey = (Key *) right->keys.FirstListEntry();
303|        for (i = 0; i < mvkeys; i++)  {
304|           Key *nkey = (Key *) movekey->NextListEntry();
305|           movekey->DeleteListEntry();
306|           movekey->InsertListEntry(inskey, &right->keys);
307|           movekey = nkey;
308|        }
309|        right->header.keycount = rightct;
310|        left->header.keycount  = leftct;
311|        if (!right->header.isleaf)
312|           right->Adoption();
313|     }
314|     nodechanged =
315|     sibling.nodechanged =
316|     parent.nodechanged = True;
317|     return True;
318| }
```

```
319|
320| // --- implode the keys of two sibling nodes
321| Bool TNode::Implode(TNode &right)
322| {
323|    int totkeys = right.header.keycount+header.keycount;
324|    if (totkeys >= m() ||
325|          right.header.parent != header.parent)
326|       return False;
327|    nodechanged = right.nodechanged = True;
328|    header.rightsibling = right.header.rightsibling;
329|    header.keycount += right.header.keycount+1;
330|    right.header.keycount = 0;
331|    // -- get the parent of the imploding nodes
332|    TNode parent(btree, header.parent);
333|    parent.nodechanged = True;
334|    // -- position parent's currkey to
335|    //      key that points to siblings
336|    parent.SearchNode((Key *) keys.FirstListEntry());
337|    // -- move the parent's key to the left sibling
338|    parent.currkey->DeleteListEntry();
339|    parent.currkey->AppendListEntry(&keys);
340|    parent.currkey->lowernode = right.header.lowernode;
341|    parent.header.keycount-;
342|    if (parent.header.keycount == 0)
343|       // - combined the last two leaf nodes into a new root
344|       header.parent = 0;
345|    // - move the keys from the right sibling into the left
346|    Key *movekey = (Key *) right.keys.FirstListEntry();
347|    while (movekey != NULL) {
348|       Key *nkey = (Key *) movekey->NextListEntry();
349|       movekey->DeleteListEntry();
350|       movekey->AppendListEntry(&keys);
351|       movekey = nkey;
352|    }
353|    if (header.rightsibling)   {
354|       // - point right sibling of old right to imploded node
355|       TNode farright(btree, header.rightsibling);
356|       farright.header.leftsibling = GetNodeNbr();
357|       farright.nodechanged = True;
358|    }
359|    Adoption();
360|    return True;
361| }
```

Listing C.25 NODE.H

```
 1| // ------ node.h
 2|
 3| #ifndef NODE_H
 4| #define NODE_H
 5|
 6| typedef unsigned NodeNbr;
 7| const int nodelength = 128;
 8| const int nodedatalength = nodelength - sizeof(NodeNbr);
 9| const int fileheadersize = sizeof(NodeNbr) * 2;
10|
11| // ============================
12| // Node File Header Record
13| // ============================
14| class FileHeader  {
15|    NodeNbr deletednode;      // first deleted node
16|    NodeNbr highestnode;      // highest assigned node
17|    NodeNbr origdeletednode; // original first deleted node
18|    NodeNbr orighighestnode; // original highest assigned node
19| protected:
20|    fstream nfile;
21|    Bool newfile;             // true if building new node file
22| public:
23|     FileHeader(String filename);
24|    virtual ~FileHeader();
25|    void SetDeletedNode(NodeNbr node) { deletednode = node; }
26|    NodeNbr DeletedNode() { return deletednode; }
27|    void SetHighestNode(NodeNbr node) { highestnode = node; }
28|    NodeNbr HighestNode() { return highestnode; }
29|    NodeNbr NewNode();
30|    fstream& Nfile() { return nfile; }
31|    void ResetNewFile() { newfile = False; }
32| };
33|
34| // ============================
35| // Node Record
36| // ============================
37| class Node  {
38|    NodeNbr nextnode;
39| protected:
40|    FileHeader *owner;
41|    NodeNbr nodenbr;  // current node number
42|    Bool nodechanged; // true if the node changed
```

```
43|    Bool deletenode;  // true if the node is being deleted
44| public:
45|    Node(FileHeader *hd = NULL, NodeNbr node = 0);
46|    virtual ~Node();
47|    Node& operator=(Node &node);
48|    void SetNextNode(NodeNbr node)
49|             { nextnode = node; MarkNodeChanged(); }
50|    NodeNbr NextNode()       { return nextnode; }
51|    void SetNodeNbr(NodeNbr node) { nodenbr = node; }
52|    NodeNbr GetNodeNbr()     { return nodenbr; }
53|    void MarkNodeDeleted()   { deletenode = True; }
54|    void MarkNodeChanged()   { nodechanged = True; }
55|    Bool NodeChanged()       { return nodechanged; }
56|    long NodeAddress();
57|    virtual int NodeHeaderSize() { return sizeof(NodeNbr); }
58| };
59|
60| #endif
```

Listing C.26 NODE.CPP

```
 1| // ----- node.cpp
 2|
 3| // ======================================
 4| // Parody file Node class member functions
 5| // ======================================
 6|
 7| #include <io.h>
 8|
 9| #include "parody.h"
10|
11| // ---- construct a file header
12| FileHeader::FileHeader(String filename)
13| {
14|    deletednode = highestnode = 0;
15|    newfile = (Bool) (access(filename, 0) != 0);
16|    // -- open the file
17|    nfile.open(filename, ios::in | ios::out
18| #ifndef ZTC
19|             | ios::binary
20| #endif
21|             );
22|    if (!newfile)  {
```

```
23|        // --- read the header
24|        nfile.read((char *)&highestnode, sizeof highestnode);
25|        nfile.read((char *)&deletednode, sizeof deletednode);
26|        nfile.seekp(nfile.tellg());
27|     }
28|     else  {
29|        // --- write the header
30|        nfile.write((char *)&highestnode, sizeof highestnode);
31|        nfile.write((char *)&deletednode, sizeof deletednode);
32|        nfile.seekg(nfile.tellp());
33|     }
34|     origdeletednode = deletednode;
35|     orighighestnode = highestnode;
36| }
37|
38| FileHeader::~FileHeader()
39| {
40|     if (deletednode != origdeletednode ||
41|           highestnode != orighighestnode)  {
42|        // -- the file header has changed
43|        nfile.seekp(0);
44|        nfile.write((char *)&highestnode, sizeof highestnode);
45|        nfile.write((char *)&deletednode, sizeof deletednode);
46|     }
47|     nfile.close();
48| }
49|
50| // --- appropriate a new node
51| NodeNbr FileHeader::NewNode()
52| {
53|     NodeNbr newnode;
54|     if (deletednode)  {
55|        newnode = deletednode;
56|        Node node(this, newnode);
57|        deletednode = node.NextNode();
58|        node.SetNextNode(0);
59|     }
60|     else
61|        newnode = ++highestnode;
62|     return newnode;
63| }
64|
65| // ----- construct a new node
66| Node::Node(FileHeader *hd, NodeNbr node)
67| {
```

```
68|     nextnode = 0;
69|     nodechanged = deletenode = False;
70|     nodenbr = node;
71|     owner = hd;
72|     if (nodenbr)    {
73|        fstream& nf = owner->Nfile();
74|
75|        long nad = NodeAddress();
76|        nf.seekg(nad);
77|        // --- read the header
78|        nf.read((char *) &nextnode, sizeof nextnode);
79|        nf.seekp(nf.tellg());
80|
81|        if (nf.eof() || nf.fail()) {
82|           nf.clear();
83|           // -- appending a new node
84|           nf.seekp(nad);
85|           nf.write((char *) &nextnode, sizeof nextnode);
86|           nf.seekg(nf.tellp());
87|        }
88|     }
89| }
90|
91| // --- assignment operator
92| Node& Node::operator=(Node &node)
93| {
94|     nextnode = node.nextnode;
95|     owner = node.owner;
96|     nodenbr = node.nodenbr;
97|     nodechanged = node.nodechanged;
98|     deletenode = node.deletenode;
99|     return *this;
100| }
101|
102| // --- destroy the node
103| Node::~Node()
104| {
105|     if (owner && nodenbr && (nodechanged || deletenode))  {
106|        fstream& nf = owner->Nfile();
107|        if (deletenode)    {
108|           nextnode = owner->DeletedNode();
109|           owner->SetDeletedNode(nodenbr);
110|        }
111|        nf.seekp(NodeAddress());
112|        // --- write the header
```

```
113|        nf.write((char *) &nextnode, sizeof nextnode);
114|        if (deletenode)   {
115|           // --- zero fill the deleted node
116|           char fill[nodedatalength];
117|           memset(fill, 0, nodedatalength);
118|           nf.write(fill, nodedatalength);
119|        }
120|     }
121| }
122|
123| // ----- compute the disk address of a node
124| long Node::NodeAddress()
125| {
126|    long adr = nodenbr-1;
127|    adr *= nodelength;
128|    adr += fileheadersize;
129|    return adr;
130| }
```

Listing C.27 INDEX.CPP

```
  1| // ----- index.cpp
  2|
  3| // ==================================
  4| // A program to rebuild the indexes
  5| // for a PARODY database
  6| // ==================================
  7|
  8| #include "stdio.h"
  9| #include "personel.h"   // application-specific header
 10|
 11| // ==================================
 12| // Personnel Database
 13| // The Employee, Department, Project, and Assignment
 14| // classes depend on this personnel pointer.
 15| // ==================================
 16| Parody *personnel;
 17|
 18| void main()
 19| {
 20|    String dbname("PERSONEL");
 21|    remove(dbname+String(".NDX"));   // delete old index file
 22|    personnel = new Parody(dbname);  // open database
 23|    ObjectHeader objhdr;
 24|
 25|    ObjAddr nd = 1;
```

```
26|     // --- scan Parody nodes
27|     while (nd < personnel->HighestNode())  {
28|        // - read the object header for this node
29|        personnel->GetObjectHeader(nd, objhdr);
30|        // - object relative node# 0 is 1st node of object
31|        if (objhdr.ndnbr == 0)  {
32|           // - tell Parody to rebuild indexes this object
33|           personnel->RebuildIndexes(nd);
34|           // -- rebuild depending on class type
35|           switch (objhdr.classid) {
36|              case EMPLOYEE:
37|              {
38|                 Employee empl;
39|                 cout << "Empl: ";
40|                 empl.Display();
41|                 break;
42|              }
43|              case DEPARTMENT:
44|              {
45|                 Department dept;
46|                 cout << "Dept: ";
47|                 dept.Display();
48|                 break;
49|              }
50|              case PROJECT:
51|              {
52|                 Project proj;
53|                 cout << "Proj: ";
54|                 proj.Display();
55|                 break;
56|              }
57|              case ASSIGNMENT:
58|              {
59|                 Assignment assgn;
60|                 cout << "Asgn: ";
61|                 assgn.Display();
62|                 break;
63|              }
64|              default:
65|                 break;
66|           }
67|        }
68|        nd++;
69|     }
70|     delete personnel;
71| }
```

▼ Glossary

This glossary defines terms that have specific meaning in the realms of C++, object-oriented programming, and database management. To help you separate the meanings, the definitions are prefixed with (OOP) for object-oriented programming terms, (C++) for terms that apply specifically to the C++ language, and (DB) for terms of databases and database management.

Abstract base class (OOP) A class definition that will always be a base class for other classes to be derived from. No specific objects of the base class will be declared by the program. A C++ abstract base classes is one that has a pure virtual function, a protected constructor, or a protected destructor.

Abstract data type (OOP) Also called ADT. A user-defined data type built as a C++ class. The details of implementation are not necessarily a part of the ADT. See also primitive data type and concrete data type.

Abstraction (OOP) Defining an abstract data type by designing a class.

Anonymous object (C++) An internal, temporary object created by the compiler.

Argument (C++) The value passed to a function. Its type must match that of the function's corresponding parameter as declared in the function's prototype. See parameter.

Base class (OOP) A class from which other classes derive characteristics. All the characteristics of the base are inherited by the derived class. Also called superclass.

Catalog (DB) The file of database format and rules statements that constitute the definition of its schema.

Child record (DB) A record in a database that is related to higher records in the database. The higher records are "parent" records.

Class (OOP) A user-defined data type that may consist of data members and member functions.

Class hierarchy (OOP) A system of base and derived classes.

Column (DB) A data element field in a relational database table of rows and columns. A column represents all the instances of the same data element in the file/table.

Concrete data type A user-defined or library data type complete with interface and implementation. The CDT is meant to be instantiated as an object and is not intended solely to be derived from.

Constructor (C++) The function executed by the compiler when the program declares an instance of a class. See also destructor.

Data definition language (DB) The language that describes the schema for a database.

Data element (DB) A single entity of data usually one item of a data type. Collections of data elements form files in a database.

Data manipulation language (DB) The language that applications programs use to store, retrieve, and maintain data in a database. Sometimes referred to by the less-specific term, *application program interface.*

Data member (C++) A data component of a class. It may be any valid data type including class objects and references.

Database (DB) A collection of data files integrated loosely to support a common application.

Database management system (DB) General-purpose systems software that manages the storage and retrieval of data in a database.

Declaration (C++) As opposed to definition. A declaration is the statement that declares the existence of an object. A declaration reserves memory.

Definition (C++) As opposed to declaration. A definition is the statement that defines the format of a type; a definition reserves no memory.

Derived class (OOP) A class that inherits some of its characteristics from a base class. Also called a subclass.

Destructor (C++) The function executed by the compiler when a declared instance of a class goes out of scope. See also constructor.

Domain (DB) A column in a relational database that adheres to specified rules in the schema with respect to content and format. Similar to a data element except that the catalog rules restrict the allowable data values.

Encapsulation (OOP) The activity of defining a class with its data members and member functions encapsulated into the definition. Encapsulation implies an implementation, which is hidden from the class user, and an interface, which is visible to the class user.

Extraction Operator (C++) The overloaded >> operator that reads (extracts) values from an input stream. See also insertion operator.

File (DB) A collection of records of a common format in a database.

Free store (C++) The C++ heap. A dynamic memory pool that programs use to allocate and release temporary memory buffers.

Friend (C++) A function that has access to the private members of a class but that is not a member function of that class. The class definition declares the function to be a friend.

Hierarchical data model (DB) A database of parent and child files. Each record in a parent file may have multiple child records. Each record in a child file may have only one parent record. A child file may be parent-to-lower-child files, forming a multiple-level hierarchy. See also network data model and relational data model.

Hierarchy (OOP) See class hierarchy.

Inheritance (OOP) The ability for one class to inherit the characteristics of another. The inherited class is said to be derived from the base class. Also called subclassing.

Implementation (OOP) The private members of a class. The implementation defines the details of how the class implements the behavior of the abstract base type. See also interface.

Index (DB) The file that associates key data values with the records that they index.

Inline function (C++) A function that the compiler compiles as in-line code every time the function is called.

Insertion operator (C++) The overloaded << operator that writes (inserts) values to an output stream. See also extraction operator.

Instantiate (OOP) Declare an object of a data type, usually a class.

Interface (OOP) The public members of a class, which define the class user's interface to the class's data and its behavior. Usually implemented as member functions. See also implementation.

Intrinsic data type (OOP) See primitive data type.

Join (DB) A relational database operator that creates a new table by logically joining two other tables on the basis of common values in a common column.

Key (DB) A data element used to index a record in a database.

Linkage specification (C++) Notation that tells the C++ compiler that a function was or is to be compiled with the linkage conventions of another language.

Manipulator (C++) A value that a program sends to a stream to tell the stream to modify one of its modes.

Member (C++) A component of a class, either a data member or a member function.

Member function (C++) A function component of a class, also called a method. A member function may be virtual.

Message (OOP) A message is the invocation of a class's member function in the name of a declared object of the class. The message is said to be sent to the object to tell it to perform its function. The message includes the function call and the arguments that accompany it.

Method (OOP) A method in C++ is a member function of a class. Programs send messages to objects by invoking methods.

Multiple inheritance (OOP) The ability for a derived class to inherit the characteristics of more than one base class.

Network data model (DB) A database of parent and child files. Each record in a parent file may have multiple child records. Each record in a child file may have multiple parent records. A child file may be parent-to-lower-child files, forming a multiple-level network. See also hierarchical data model and relational data model.

Object (OOP) A declared instance of a data type, including standard C++ data types as well as objects of classes.

Object database (OOP) A collection of persistent objects.

Overloaded function (C++) A function that has the same name as one or more other functions but that has a different parameter list. The compiler selects which function to call based on the data types and number of arguments in the call.

Overloaded operator (C++) A function that executes when a C++ operator is seen in a defined context with respect to a class object.

Overriding function (C++) A function in a derived class that has the same name, return type, and parameter list as a function in the base class. The compiler calls the overriding function when the program calls that function in the name of an object of the derived class. If the function in the base class is virtual, the compiler calls the derived class's function even when the call is through a pointer or reference to the base class. See also pure virtual function.

Parameter (C++) The declaration of a data item that a function expects to be passed to it. This declaration includes the item's type and name and appears in the function's declaration block at the beginning of the function. When the parameter appears in the function's prototype, the parameter's name may be omitted. See argument and prototype.

Parameter list (C++) The list of parameter types and names in a function declaration block. Also the same list, which may exclude the names, in a function prototype.

Parent record (DB) A record in a database that is related to lower records in the database. The lower records are "child" records.

Persistence (OOP) The ability of an object to succeed its creator and to subsequently exist in space other than the space in which it was created.

Persistent object (OOP) An object that exhibits persistence.

Polymorphism (OOP) The ability for methods in a class hierarchy to exhibit different behavior for the same message, depending on the type of the object for which the method is invoked and without regard to the class type of the reference to the object.

Primary key (DB) The key data value that uniquely identifies a record in a database file. See also secondary key.

Primitive data type (OOP) A data type known to the compiler. Primitive data types in C++ are char, int, float, double, and pointer. The integer types may be further qualified as long, short, and unsigned. All types may be organized into arrays of like types and structures and unions of varying types. Also called intrinsic data types.

Private class members (C++) Members of a class for which access is granted only to the class's member functions and to friend functions of the class.

Project (DB) A relational database operator that creates a different "view" of a "table" from a subset of its columns. See also select and join.

Protected class members (C++) Members of a class that are private except to member functions of derived classes.

Prototype (C++) The definition of a function's name, return type, and parameter list.

Public class members (C++) Members of a class to which access is granted to all functions within the scope of an object of the class.

Pure virtual function (C++) A virtual function in a base class that must have a matching function in a derived class. A program may not declare an instance of a class that has a pure virtual function. A program may not declare an instance of a derived class if that derived class has not provided an overriding function for each pure virtual function in the base.

Query (DB) A process that creates a new "view" of the contents of a relational database by applying and combining the "select," project," and "join" operators. The new view is presented as another table.

Record (DB) An aggregate of "data elements" that form to support one functional aspect of a database's application. A collection of records of common format forms a "file."

Reference (C++) A variable name that is an alias for another variable.

Relation (DB) A relational database term that is often interchanged with "file" and "table," although "relation" does not imply any physical implementation.

Relational data model (DB) A representation of data as tables of rows and columns. Any relationships between tables are formed by common values in common columns. See also hierarchical data model and network data model.

Repeating group (DB) An array with fixed or variable dimension of similar data elements in a database record. Not permitted in a relational database.

Row (DB) Relational database term that is analogous to "record" in traditional databases.

Schema (DB) The formal definition of the format of a database. The schema defines the files, their format, and their relationships.

Secondary key (DB) A key data value that indexes a file on other than its "primary key." The value of a secondary key does not have to be unique. Multiple records can have the same value. When a secondary key is the primary key of another file, the two files have a many-to-one relationship.

Select (DB) A relational database operator that selects rows from a table based on the criteria of a "query." See also project and join.

Stream (C++) A category of character-oriented data files or devices where the data characters exist in an input or output stream.

Subclass (OOP) See derived class.

Subclassing (OOP) See inheritance.

Superclass (OOP) See base class.

Table (DB) In a relational database, a "file" of rows and columns.

This (C++) A pointer that exists in all nonstatic member functions. The pointer is a pointer to an object of the class. It points to the object for which the function is being executed.

Tuple (DB) A relational database term that refers to a particular row in a table of rows and columns. Analogous to "record" in traditional databases and "object" in object databases.

Type (OOP) The type of a program constant or variable, which can be of a primitive or an abstract data type.

Type conversion (C++) The conversion of one type to another. The compiler has built-in type conversions, and a class may define its own conversions for converting from an object of the class to another type and from another type to an object of the class.

Type-safe linkage (C++) A technique that insures that functions and function calls in separately compiled program modules use consistent parameter lists.

View (DB) A user's perception of the data in a database, which may or may not coincide with the actual data representation. A view is prepared for the user by a program, typically a query program.

Virtual function (C++) A member function in a class from which other classes may be derived. If the derived class has a function with the same name and parameter list, the derived class's function is always executed for objects of the derived class. See also pure virtual function and overriding function.

Index

A

abstract base class, 20, 102
abstract data type, 13, 14, 36
abstraction, 13, 14
AddAssignment, 169
AddDepartment, 169
AddEmployee, 169
adding an object, 169
adding objects, 71
AddObject, 72, 73, 125, 136, 137, 142, 191, 192
AddProject, 169
aggregate, 67
allocating a node, 128
ANSI, 55, 155
ANSI protocols, 158
ANSI.SYS, 159, 177
AnyKey, 161, 170, 185
API, 5
AppendListEntry, 157, 188
applications program interface, 5
ARM, 55
assignment, 60
Assignment, 112, 168
AssignmentID, 112
associative access, 57, 102, 138
AUTOEXEC.BAT, 175

B

B-tree, 109, 129, 194
base class, 20
basis, database design, 79
Bayer, R., 129
behavior, 15
BIOS, 158
Bool, 184
Borland C++, 7, 173, 175

C

catalog, 67
change a persistent object, 136
ChangeAssignment, 170
ChangeDepartment, 170
ChangeEmployee, 170
ChangeObject, 72, 73, 119, 125, 136, 137, 144, 191, 192
ChangeProject, 170
changing an object, 170
changing objects, 72
child record, 2, 32
cin, 159
class, 3, 14
class header record, 129
class relationship, 59, 95, 113
classes:
  Assignment, 112, 168
  AssignmentID, 112
  Department, 104, 167
  DepartmentNo, 108, 167
  Employee, 134, 168
  EmployeeBody, 116, 168
  GUI, 158, 185
  Handle, 116, 168, 192
  Key, 108, 120, 129, 134, 166, 167, 189
  LinkedListEntry, 157, 188
  LinkedListHead, 157, 188
  Node, 127
  Parody, 103, 189
  Persistent, 104, 106, 120, 166, 190
  Project, 111, 168
  String, 155, 185
ClearScreen, 162, 185
COBOL, 33
CODASYL, 33, 36
Codd, E. F., 66
CompareEmployees, 171

CompuServe Information System, 7
concatenated key, 35, 94, 112, 168
CONFIG.SYS, 159, 177
connector file, 35, 94, 112, 168
const:
  SCREENWIDTH, 184
  MAXCOMMANDS, 184
  nodelength, 184
  NodeNbr, 193
  SCREENHEIGHT, 184
  SCREENWIDTH, 184
ConstructBody, 116
constructor:
  sequence of, 119
  base class, 44
  derived persistent class, 120, 125
  Key class, 108, 120
  Persistent class, 120
  persistent object, 105
conversion, database, 146
copy class, 116
copy construction, 60
copy constructor, 115
create a persistent object, 134

D

DAT file, 104, 134, 193
data aggregate, 85, 87
data definition language, 29, 30, 69
data description language, 5
data element, 28
data items, identification, 84
data management language, 5
data manipulation language, 29, 30
data member dictionary, 86
data members, 15, 85
data node, 193
data type, 28, 36
data type method, 18
database, 28
database conversion, 146
database management system, 28, 29
database models, 30
DDL, 5
DeleteAssignment, 170
DeleteDepartment, 170
DeleteEmployee, 170
DeleteListEntry, 157, 188
DeleteObject, 72, 73, 126, 136, 137, 144, 191, 192
DeleteProject, 170
deleting a node, 128
deleting objects, 72, 136, 170
Department, 104, 167
DepartmentNo, 108, 167
derived class, 20
derived persistent class constructor, 120, 125
derived persistent class destructor, 122, 125
design notation, 37
destructor:
  sequence of, 122
  base class, 44
  derived class, 45
  derived persistent class, 122, 125
  persistent object, 106
dialogs, 161
Display, 167, 170
DML, 5

E

embedded object, 48, 116, 144
Employee, 134, 168
EmployeeBody, 116, 168
empty object, instantiating, 73
encapsulation, 13, 15
Error, 161, 185
Extern "C", 64

F

file, 28, 85
file header block, 193
file names, 104, 134
FindChar, 187
FindObject, 73, 74, 138, 140, 141, 190, 191, 192
FirstListEntry, 188
FirstObject, 74, 139, 140, 144, 170, 190, 191, 192
flat file, 34

fstream, 47
functions, examples:
AddAssignment, 169
AddDepartment, 169
AddEmployee, 169
AddProject, 169
ChangeAssignment, 170
ChangeDepartment, 170
ChangeEmployee, 170
ChangeObject, 72, 73, 119, 125, 136, 137, 144, 191, 192
ChangeProject, 170
CompareEmployees, 171
DeleteAssignment, 170
DeleteDepartment, 170
DeleteEmployee, 170
DeleteListEntry, 157, 188
DeleteProject, 170
Display, 167, 170
Get, 167, 169
GetExisting, 167, 170, 171
Header, 167
Input, 167
InputManager, 167
InputName, 167
ListAssignment, 170
ListDepartment, 170
ListEmployee, 170
ListEmployeeProjects, 171
ListProject, 170
ListProjectEmployees, 171
SelectChange, 167, 170
TryChange, 167
functions: GUI class
AnyKey, 161, 170, 185
ClearScreen, 162, 185
Error, 161, 185
GetKBChar, 159, 162, 185
KBCharWaiting, 162, 185
PutBack, 162, 185
Selection, 160, 185
SetCursor, 162, 185
StatusLine, 162, 185
UserInput, 161, 185
UserMenu, 159, 185
WriteChar, 162, 185
YesNo, 161, 170, 185
functions, linked list classes
AppendListEntry, 157, 188
FirstListEntry, 188
InsertListEntry, 157, 188
LastListEntry, 188
NextListEntry, 157, 189
PrependListEntry, 157, 188
PrevListEntry, 157, 189
functions: PARODY
AddObject, 72, 73, 125, 136, 137, 142, 191, 192
ConstructBody, 116
DeleteObject, 72, 73, 126, 136, 137, 144, 191, 192
FindObject, 73, 74, 138, 140, 141, 190, 191, 192
FirstObject, 74, 139, 140, 144, 170, 190, 191, 192
GetObjectHeader, 189
isNullValue, 111, 138, 190
LastObject, 74, 139, 140, 144, 190, 191, 192
LoadObject, 105, 106, 107, 121, 125, 142, 146, 166, 190
Make, 110
NextObject, 74, 140, 141, 144, 170, 171, 190, 191, 192
ObjectAddress, 143, 192
ObjectExists, 134, 144, 169, 170, 171, 191, 192
PreviousObject, 74, 140, 144, 190, 191, 192
PrimaryKey, 113, 169, 190
Read, 104, 106, 110, 121, 125, 142, 144, 147, 148, 164, 166, 167, 189, 190
ReadObject, 106, 107, 110, 121, 122, 125, 145, 147, 166, 191
RebuildIndexes, 152, 189
RebuildingIndexes, 189
Relate, 113, 190
SaveObject, 106, 107, 123, 125, 126, 166, 190
Write, 104, 107, 108, 110, 123, 125, 129, 142, 144, 147, 148, 164, 166, 167, 189, 190
WriteObject, 108, 110, 124, 125, 147, 166, 191
functions, String class
FindChar, 187
left, 155, 187
Length, 185
mid, 155, 187

right, 155, 187
Strlen, 185
functional method, 17
functional requirements, 79, 82

G

Game of Life, 163
generic user interface, 6
Get, 167, 169
GetExisting, 167, 170, 171
GetKBChar, 159, 162, 185
GetObjectHeader, 189
GUI, 6, 158, 185

H

Handle, 116, 168, 192
handle class, 116
handle/copy idiom, 115, 118, 146, 168, 171
HASA relationship, 21, 38
Header, 167
header record, 127
hierarchical data model, 4, 31
hierarchy, 2

I

implementation, 13
implicit conversion method, 18
implmentation, 15
index file, 129
index file, rebuild, 150
index header block, 194
index header record, 129
INDEX.EXE, 176, 178
indexes, 57
inheritance, 13, 20
Input, 167
InputManager, 167
InputName, 167
InsertListEntry, 157, 188
instantiate, 15
integrity, 125, 137
integrity, object, 58
interclass relationships, 71
interface, 13, 15
INTERNET, 7
iostream, 6
ISA relationship, 21, 38
isNullKey, 138
isNullValue, 111, 190

J

JOIN operator, 67

K

KBCharWaiting, 162, 185
Key, 108, 120, 129, 134, 166, 167, 189
Key class, 69
Key class constructor, 120
key data member, 93
key length, 109
keyless class, 142
keyless objects, 163

L

LastListEntry, 188
LastObject, 74, 139, 140, 144, 190, 191, 192
left, 155, 187
Length, 185
Life, 163
LIFE.DAT, 177
LIFE.EXE, 176
LIFE.NDX, 177
linkage specification, 64
linked list, 155
LinkedListEntry, 157, 188
LinkedListHead, 157, 188
linker, 97
ListAssignment, 170
ListDepartment, 170
ListEmployee, 170
ListEmployeeProjects, 171
ListProject, 170
ListProjectEmployees, 171
LoadObject, 105, 106, 107, 121, 125, 142, 146, 166, 190

M

Make, 110
MAKEFILE.BCC, 175, 178

MAKEFILE.MSC, 175, 179
MAKEFILE.ZTC, 176, 180
many-to-many, 35, 168
MAXCOMMANDS, 184
McCreight, E., 129
member function, 19
member functions, 15
memory model, 175, 176
message, 17
method, 17, 97
methods: see functions
Microsoft C++, 7, 173, 175
mid, 155, 187
MS-DOS, 6, 158, 173
multiple inheritance, 21
Myers, Terry, 50

N

navigating a class, 73
navigating among classes, 74
navigational access, 57, 102, 138
NDX file, 104, 134, 193
network, 2
network data model, 4, 32
NextListEntry, 157, 189
NextObject, 74, 140, 141, 144, 170, 171, 190, 191, 192
Node, 127
node allocation, 128
node deletion, 128
node record, 127
node thread, 127
nodelength, 184
NodeNbr, 184, 193
nonpersistent members, 47
normalization, 35
null key, 111

O

ObjAddr, 142, 184
object, 13
object data, 194
object data model, 36, 63, 64
object database, 4
object file, 128
object header, 194
object identification, 57
object integrity, 58, 73
object-oriented programming, 9
ObjectAddress, 143, 192
ObjectExists, 134, 144, 169, 170, 171, 191, 192
one-to-many, 35
overloaded assignment operator, 115
overloaded operators, 18, 110, 187, 189
overriding member function, 23

P

parent record, 2, 32
PARODY, 57, 69
Parody, 103, 189
Parody object, 104, 119
parody.bld, 173
PARODY.BLD, 182
PARODY.LIB, 176
performance requirements, 79, 83
persistence, 41, 53
Persistent, 69, 88, 104, 106, 120, 166, 190
Persistent base class, 43
Persistent class constructor, 120
persistent object, 64, 134, 193
persistent object constructor, 105
persistent object destructor, 106
persistent object, change, 136
persistent object, create, 134
persistent object, delete, 136
persistent object, retrieve, 134
PERSONEL.DAT, 177
PERSONEL.EXE, 176
PERSONEL.NDX, 177
pointer member, 42, 47
polymorphism, 13, 22
portability, 55
PrependListEntry, 157, 188
preprocessor, 55
PreviousObject, 74, 140, 144, 190, 191, 192
PrevListEntry, 157, 189
primary key, 34, 57, 64, 93, 108, 129, 134, 139
PrimaryKey, 113, 169, 190
primitive data type, 14, 18
private member functions, 15
problem specification, 80
procedural programming, 11
Project, 111, 168

PROJECT operator, 67
proprietary data model, 31
public member function, 16, 17
PutBack, 162, 185

Q

query language, 29, 30

R

Read, 45, 104, 106, 110, 121, 125, 142, 144, 147, 148, 164, 166, 167, 189, 190
ReadObject, 106, 107, 110, 121, 122, 125, 145, 147, 166, 191
RebuildIndexes, 152, 189
RebuildingIndexes, 189
record, 28
reference counting idiom, 60
reference members, 48
refinement, 79
Relate, 113, 190
relational data model, 4, 33, 64, 66
relations, 66
relocatable library, 97
repeating groups, 67, 91
report writer, 29, 30
representation, 15
resources list, 80
retrieve a persistent object, 134
retrieving objects, 71
right, 155, 187

S

SaveObject, 106, 107, 123, 125, 126, 166, 190
schema, 32, 102
SCREENHEIGHT, 184
secondary key, 34, 57, 93, 95, 111, 129, 134, 139
SELECT operator, 67
SelectChange, 167, 170
Selection, 160, 185
SetCursor, 162, 185
single inheritance, 20
sizeof, 42, 45
Smalltalk, 20
static member, 47
StatusLine, 162, 185
String, 155, 185
Strlen, 185
structure, 42
structured programming, 65
subclass, 20
superclass, 20

T

thispers, 120
TryChange, 167
tuples, 66
type identification, 47, 102
type: Bool, 184
type: NodeNbr, 184
type: ObjAddr, 184

U

UserInput, 161, 185
UserMenu, 159, 185

V

variable-length object, 37, 65
views, 67
virtual function, 23, 49
vptr, 43, 49
vtbl, 49

W

wrapper class, 64
Write, 45, 104, 107, 108, 110, 123, 125, 129, 142, 144, 147, 148, 164, 166, 167,189, 190
WriteChar, 162, 185
WriteObject, 108, 110, 124, 125, 147, 166, 191

X

X3J16, 55, 155

Y

YesNo, 161, 170, 185

Z

Zortech C++, 173, 176

C++ DATABASE DEVELOPMENT

DISK AVAILABLE

C++ Database Development **will provide you with all the tools you need for writing C++ database programs. Learn how to use C++ to design and develop utility and database management programs. Explore the full potential of the C++ object-oriented language. Topics include database fundamentals and design, database management, building the software, and much more. Let MIS:Press put the full power of C++ database development at your fingertips.**

AL STEVENS is in his fifth year of writing the C Programming column for *Dr. Dobb's Journal*, the world's leading monthly magazine for professional programmers. He is the author of over a dozen books about programming and using computers. He lectures nationally for programmers, educators, and systems managers on technical issues related to programming and systems design. A professional programmer since 1958, and an independent programming consultant and writer since 1978, Stevens maintains a loyal following of readers who appreciate his insight and clear writing style.

In This Book...

- Coverage of PARODY–the Persistent, Almost-Relational Object Database Manager.
- An introduction to object-oriented programming.
- Relational and object-oriented database management.
- Advanced techniques for designing an object database.
- Using PARODY to build your own persistent object database management system.
- Creating, storing, and retrieving persistent objects with navigational access and associative access.
- Complete C++ source code.
- The platform-independent code compiles with Borland C++, Microsoft C++, and Zortech.

ISBN 1-55828-216-5 $24.95
90000>
9 781558 282162

MIS: PRESS
A Subsidiary of Henry Holt and Co., Inc.

Level

Advanced Programming