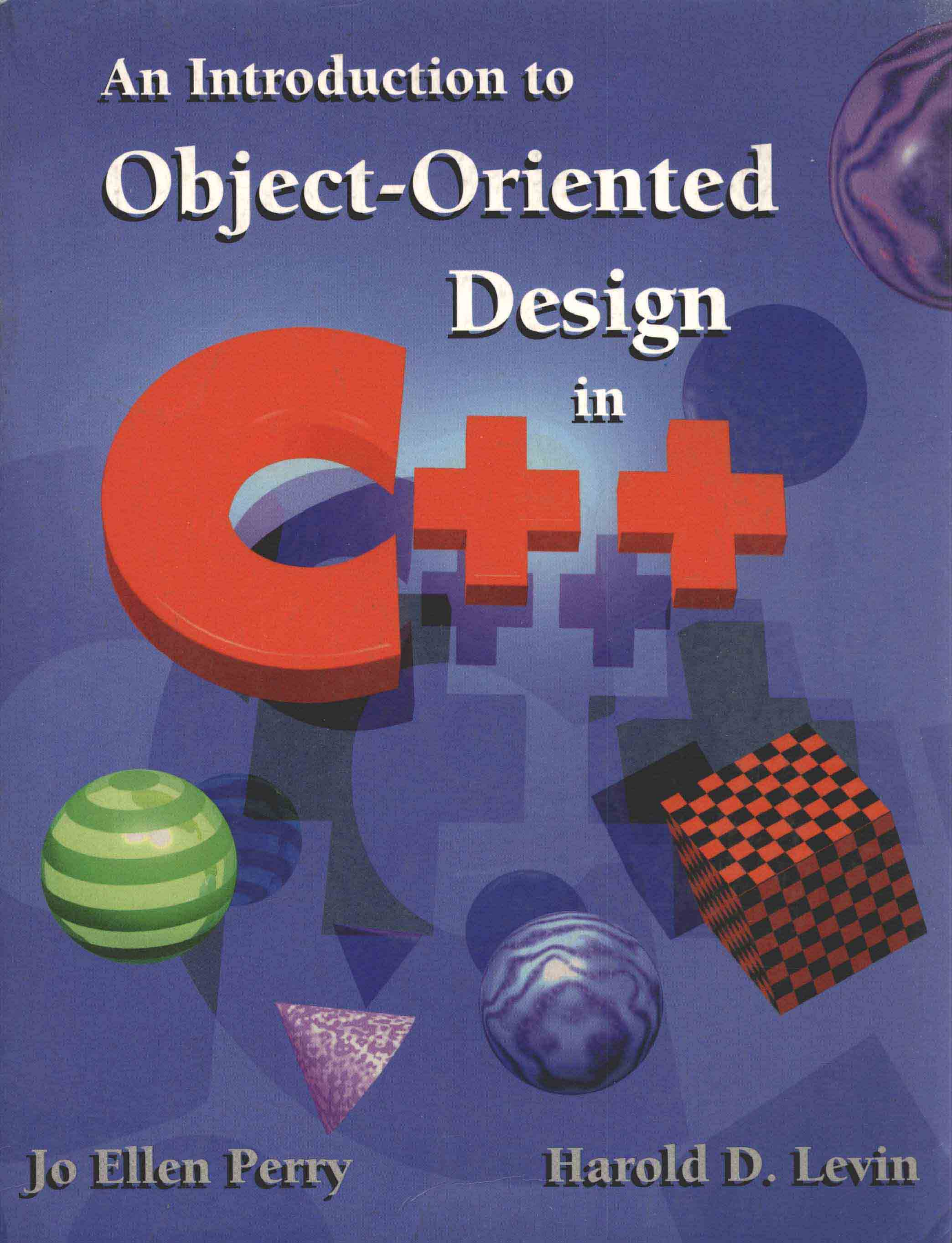
# An Introduction to Object-Oriented Design in C++



Jo Ellen Perry

Harold D. Levin

Lynne Doran Cote *Sponsoring Editor*
Helen M. Wythe *Production Supervisor*
Eileen Hoff *Cover Design Supervisor*
Roy Logan *Senior Manufacturing Manager*
Kenneth Wilson *Text Designer*
Alena Konecny *Art coordinator*
Illustrations and Composition *Publishers' Design and Production Services, Inc.*

**Library of Congress Cataloging-in-Publication Data**
Perry, Jo Ellen
An introduction to object-oriented design in C++ / Jo Ellen Perry and Harold D. Levin.
p. cm.
Includes index.
ISBN 0-201-76564-0
1. Object-oriented programming (Computer Science) 2. C++ (Computer program language) I. Levin, Harold D. II. Title. III. Title: Object-oriented design in C++.
QA76.64.P527 1996
005.13'3—dc20 95-5965
CIP

1 2 3 4 5 6 7 8 9 10—MA—98979695

# Preface

TWENTY years ago, there were no desktop PCs, most software was batch processed, and most program I/O was for unit records such as card images. Today most computers are desktop machines and we operate in an environment of networks, multimedia, and interactive graphical interfaces. This book is designed for use in introductory computer science courses that acknowledge these widespread and fundamental changes.

A couple of years ago, we heard people asking "Is object-oriented programming just a flash in the pan?" We don't hear this question anymore. Today, we are in the midst of a major paradigm shift affecting almost every aspect of computing—from commercial technology to fundamental theory. Elements of the diverse collection of topics lumped under the general heading "object-oriented" are at the center of the shift and provide the momentum that will determine the direction of the discipline into the next century.

What does all this mean with respect to teaching and learning about software development? Should students learn object-oriented techniques from the start? The answer is yes, even in the earliest portions of their training. Indeed, we think that it would be as counterproductive to teach standard techniques of structured programming early in the curriculum and then add in or switch to object-oriented methodologies in advanced courses as it would have been to first teach students to program in BASIC and later on to introduce structured programming as a portion of a specialized software engineering course. Two major ideas moved us toward the very early introduction of object-oriented concepts. One is that it is vital to take design seriously. The other is that concepts and technologies associated with software reuse are so important that they must be introduced at the earliest opportunity.

## Where did we start?

We have both been teaching introductory computer science and computer programming to students for more than a decade. Until three years ago, all of our introductory teaching amounted to the same course that has been taught in Pascal since the beginning of structured programming. In the spring semester of 1992, partly because of the political pressure of other engineering departments and partly because of student demand, we taught an introductory course using C++. That first course was strictly experimental. We did not know if we could pull it off.

To make sure that students would not be shortchanged by programming in C++ rather than Pascal, we taught the experimental course as an exact mirror of the introductory course with Pascal. Pascal and C++ lectures were kept in lockstep. The day that Pascal `CASE` statements were discussed was the same day that C++ `switch` statements were discussed. The day that procedures were introduced in Pascal lectures, functions were introduced in C++ lectures. The students in Pascal and C++ wrote exactly the same lab assignments. Only the languages were different. No hint of object-oriented development was mentioned in the C++ course.

We came out of that experiment knowing that we had both succeeded and failed. We discovered that beginning students can learn C++ syntax as easily as they can learn Pascal syntax, and they can do traditional top-down design in C++ as easily as they can do it in Pascal. But that experimental course was still a C++ copy of a traditional introductory programming course. It was out of date before it started.

This book began as a result of that experience. We knew that we did not want another Pascal text in C++ clothing. At the same time, we knew that beginning students still have to learn concepts such as control structures and functions that are common to all general-purpose languages. While that knowledge is so ingrained in experienced programmers that many have trouble remembering when they did not know it, we recognize that students are not born knowing those concepts. Our first question was "How can we teach the new paradigms and still cover the fundamental concepts that every programmer knows?" Our answer is "Teach objects early along with the fundamental concepts." It is easy to teach fundamental concepts. What about objects?

## Introducing object-oriented concepts

This book introduces objects at two levels: design and programming. At the design level, we began by *thinking* in terms of objects, comparing those objects and their interactions to animated cartoons. This is a comfortable context for most students. It is through this kind of object-think that students can apply their everyday intuition. Students know how to solve all sorts of difficult problems in ordinary life. They ought to be able to use that experience in solving programming problems.

At the programming level, objects can appear in various ways. In the beginning of this book, we use comments to bracket off the parts of programs corresponding to design objects. As we introduce new language features, we rely more on the code without the comments to describe the objects developed in our designs. With classes, we are able to capture in our code exactly what we had in our designs without relying strictly on comments to show us the way.

The focus from the beginning of the text is objects, not classes, and not C++ syntax for defining classes. Objects are concrete. Classes are the abstract mechanism for producing objects. We chose to leave classes until students could first become familiar with functions and control structures. C++ has built-in objects `cin` and `cout` that students can use from the start. Furthermore, students find it easy to use class libraries and class type objects quite early. When students can appreciate the need for producing many instances of new types of objects, they are ready to tackle the syntax and conceptual scheme for defining C++ classes.

## Following through with major object-oriented concepts

Soon after we taught the first course using C++, we overheard some students saying that they knew all about object-oriented programming because they took a one-semester course in programming using C++. We knew they were wrong, but they did not find out until they tried to sell the skills they did not have. Soon, we began hearing from students asking where could they learn the important stuff—i.e., inheritance and polymorphism.

We think students need to see the whole picture in object-oriented program development. More importantly, students think they need to see the whole picture. And that picture includes first class types, inheritance, and polymorphism. Those topics are too important to be relegated to a footnote or a short appendix. Students—and people outside academia—do not think they should be ignored. And we do not think they should be ignored either.

As we remarked earlier, there has been a significant change not only in the economics and sociology of computation but also in what is at the leading edge and what is at the center of the discipline. If you think that the situation in computing today is not essentially different from the situation in the 1970s when Pascal and structured programming were new and important, then you may well be puzzled about why students should learn about objects, classes, inheritance, and polymorphism. However, neither teachers nor students can gain a full appreciation of object-oriented program development without inheritance and polymorphism.

## A quick map of the book

Novice programmers who want to learn object-oriented program development in C++ from this book should start from the beginning and follow the sequence of chapters as we have laid them out. Experienced C programmers who are determined to read as little as possible can read Chapters 1, 3, and the beginning of Chapter 6 before going to Chapter 7, which gives the first full treatment on class definitions. Experienced Pascal programmers will also find it necessary to learn the syntax introduced in Chapters 2 through 6. Chapters 8 through 11 form the heart of the object-oriented features, syntax, and culture of C++. Chapter 12 introduces the reader to container classes and linked lists. Chapter 13 is a case study of the iterative process that we went through in developing a sophisticated class library.

This book is organized on two different themes. The first theme is design and analysis of programming problems; the second is the C++ tools that are useful in implementing solutions to these problems. Students who successfully complete a programming course should have practical programming skills when they leave the course. But to prevent creating a hacker mentality—going after an immediate solution in the quickest way possible—it is important to show students useful design methodologies from the start.

This book uses the familiar changemaker problem as a running example to illustrate iterations in design, enhancements, and implementation. The initial statement of the problem from Chapter 1 is very simple. As more programming tools become available, the changemaker is reanalyzed and redesigned to incorporate these new tools and to improve either the functionality of the program solution or to improve its organization.

## For the teacher

This book contains material for a two-semester sequence in introductory design and programming. At North Carolina State University, the first course is organized into large lecture sections (175 students) with small closed labs. We have found that it is reasonable to cover Chapter 1 through the first half of Chapter 7 plus the Stream Appendices (D, E, and F) in a traditional three-semester hour introductory course. Students in four-semester hour courses could cover more material, or simply do the same material in more depth and with more examples.

The laboratory experience is important. Ideally, we would like all low-level syntax topics relegated to the lab. The lecture is a very inefficient way to cover that material. Through laboratories, especially those that have integrated support for software development, students can experiment with the language to see the practical results of the lectures.

We realize that the second course varies tremendously among different schools. The second course at North Carolina State University has traditionally focused more on software engineering concepts than data structures. The material in Chapter 7 through the remainder of the book is appropriate for such a second course.

Comprehensive exercises, varying from straightforward syntax exercises through programming projects are included at the end of every chapter. Answers to all of the exercises except the programming projects are available for instructors upon request from Addison-Wesley. For additional information, see the supplements section following. The programming projects are open-ended and can be modified to suit individual class needs.

We hope that this textbook will be useful not only for students but also for teachers who are just now starting to teach the introductory course using C++. That pedagogical task is not trivial. And it is especially not trivial to those who know C or Pascal and come to realize very late that C++ is not just a new version of C and not just Pascal with classes. Part of what we do in this book is addressed to helping our colleagues, who may be struggling just ahead of their students. We know about that struggle from our own experience.

Problem solving is difficult no matter what the language and what the tools. But be encouraged from our discoveries. The language switch—the syntax switch—is easy. Learning the culture, how good C++ programmers do things, is harder. But the most difficult thing is learning how to think in terms of object-oriented analysis and design as opposed to structured analysis and design.

## For the student

You will begin learning by solving very simple problems and writing very short programs. Think about the programs that you use and like. If after three months, you still do not have a good idea of how to write such a program, all that indicates is that programming is a broad and deep subject. It will take considerable time and effort before you will realize the goal of having good object-oriented programming skills.

When you begin your work, you will discover ways to organize what you already know and ways to spell out things that you already know how to do. You will discover new ways of looking at problems and devising solutions to them. This book is full of advice on how to learn, what is important, when to exercise old skills, and when to learn new ones. We suggest that you read carefully. Read new material in the book before you go to class. Read it after you attend a lecture. Underline key points. And work problems. Programming is not for spectators.

Do not hesitate to experiment. If you are not sure how a piece of code works, try it out. All of us, even those of us who are professional teachers, are students too. And we learn what we know by falling into lots of holes and making lots of mistakes, just as you are almost certain to do. There is an extensive set of exercises at the end of each chapter. Answers to selected exercises are available via anonymous ftp to `ftp.aw.com`. For instructions to download these files, refer to the supplements section following.

Your compiler will not be perfect, and it may not support some features of C++ mentioned in this book. (For example, the type `bool` is not available on some compilers.) We have tried to warn you in the text where some of our compilers were deficient, and where you might run into the same problems. When possible, we have suggested simple work-arounds.

## Technical standards

The code in the first seven chapters was tested on all of the following environments. All of the code was tested in at least one UNIX-based environment.

Borland Turbo C++ version 3.0 on DOS 5.0
Microsoft Visual C++ version 1.0 (Standard Edition) for Windows 3.1

Symantec C++ version 7.0 on Macintosh, system 7.0
AT&T version 3.0 on a SUN4 using SunOS 4.1
Gnu version 2.6.1 on a DECstation using Ultrix 4.2

The technical terms that we introduce in the text are not ours. Wherever possible, we have tried to conform our use of technical terms with the practice of the C++ community, its epicenter being at Bell Laboratories. We have also tried to conform with the language standards described in Margaret Ellis and Bjarne Stroustrup's *Annotated C++ Reference Manual* (the *ARM*),[1] which serves as the base document for the ANSI standardization of C++ currently in progress. In the few places where our code uses an operating system or other system-dependent headers or procedures, we isolated the nonportable features and indicated system dependence with comments.

We have entirely avoided the use of exceptions and exception handling for two reasons. First, exceptions are not as widely implemented in C++ compilers as are the other *ARM* language features, and standards for exceptions are more in flux than other parts of the language described in the *ARM*. Second, exception handling is an important addition to the control structures of the structured programming tradition but not an integral part of our central theme—object-oriented program development.

## Supplements

The following supplements are available through the Addison Wesley ftp site:

- Source code
- Answers to selected exercises
- Instructor's solutions access information

To obtain any of the supplement files, ftp to `aw.com`. Log in with user name `anonymous` and use your internet address as the password. From there, type `cd sceng/authors/perry/oop.cs1`.

## Acknowledgments

The authors gratefully acknowledge the help of many people in writing this book. Among our colleagues, Don Martin has been especially encouraging from the start. We doubt that we could have completed our work without the assistance of Susan Jones, who has worked tirelessly to check code, do problems, and suggest

[1] Ellis, Margaret A., and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Reading, Mass.: Addison-Wesley, 1990.

ways of formulating ideas. Both Susan and Don have been with us since the inception of this project, always giving constructive criticism and refraining from asking the question "Isn't it done yet?"

We appreciate the willingness of Don Martin, Susan Jones, Carol Miller, and Steve Worth to use a draft manuscript for their classroom text. We thank our students and colleagues alike for serving as guinea pigs in this project. All have provided valuable feedback.

The editorial staff at Addison-Wesley has been very helpful in preparing the final manuscript. Our editor Lynne Cote has offered valuable words of encouragement and has also provided us with a rich reference library from among the numerous excellent Addison-Wesley publications. We thank the authors of those books for pointing out to us and to their other readers the culture and use of object-oriented techniques and C++. In addition, our production supervisor Helen Wythe has been gracious and diligent in dealing with the endless problems of putting the book together.

We have taken advantage of extensive reviewer comments, incorporating many of the suggestions into the book. We thank especially James Adcock, Microsoft Corporation; Vicki H. Allan, Utah State University; Frank Cioch, Oakland University; H. E. Dunsmore, Purdue University; Thomas Hain, University of South Alabama; Dennis Heckman, Portland Community College; Robert Kline, Westchester University; Rayno Niemi, Rochester Institute of Technology; Christopher Skelley, Insight Resources; and Phil Sweany, Michigan Technological University.

Finally, our families have lived with this project almost as much as we have. Lavon and Connie have demonstrated their love for us through their wonderful patience in our extended times away from home. They have had to put up with lost weekends and exhausted spouses for over two years, and they did it almost without complaint.

# Contents

Chapter 1 **Object-Oriented Program Development** 1

1.1 What is object-oriented program development? 2
1.2 Sample problem: The convenience store 9
1.3 Sample problem: The changemaker 16
1.4 Compilers and languages: Making the program run 23
1.5 Why is object-oriented development so important? 25
1.6 Summary 26
1.7 Exercises 28

Chapter 2 **C++ Tools: Anatomy of a Simple Program** 33

2.1 The convenience store program, line-by-line 34
2.2 Storing information 42
2.3 Expressions, statements, and numeric types 45
2.4 C++ streams 60
2.5 Programming style: Does it matter? 68
2.6 Making sure that the analysis, design, and implementation are correct 70
2.7 Summary 73
2.8 Exercises 76

Chapter 3 **Implementing Object Behaviors with Functions** 87

3.1 C++ functions that return values 88
3.2 Procedures: Functions that do not return values 101
3.3 Scopes and lifetimes 111
3.4 Programming errors and functions 118
3.5 Refining object-oriented designs 120
3.6 Using Class Libraries 133
3.7 Summary 143
3.8 Exercises 146

Chapter 4 **Decision Behavior: Selections and Alternatives** 159

4.1 The `if` and `if/else` selection statements 160
4.2 Evaluating Boolean expressions 167
4.3 Typedefs and enumerations 173

4.4 `switch` statements 177
4.5 The conditional operator: Providing shorthand for a simple `if/else` 180
4.6 Recursive functions 181
4.7 Testing: The evaluation part of the implementation process 189
4.8 Application example: Navigational running lights 192
4.9 Summary 201
4.10 Exercises 203

Chapter 5 **Iteration Behavior: Loops** 211

5.1 The first loop structure: `while` loops 212
5.2 Constructing loops: How to start, how to quit, and how to get to the next step 217
5.3 The second loop structure: `do/while;` loops 227
5.4 The third loop structure: `for` loops 230
5.5 Nested loops 238
5.6 Alternative loop constructions and the `break` statement 241
5.7 Design modification: Redesigning the calculator 243
5.8 A new problem: Text analysis 246
5.9 Summary 257
5.10 Exercises 259

Chapter 6 **List Objects and Array-Based Implementations** 269

6.1 Implementing list objects with old programming tools 271
6.2 Arrays: The fundamentals 274
6.3 List type objects 283
6.4 Array-based implementations of lists 287
6.5 Sorting and searching 296
6.6 Strings: Implementation tool for names and phrases 307
6.7 Lists of names 313
6.8 Multidimensional arrays 318
6.9 Redesigning the changemaker to fit new implementation ideas 322
6.10 Summary 327
6.11 Exercises 329

Chapter 7 **Producing Objects Through Classes** 341

7.1 Simple objects: Structs 343
7.2 Implementing objects with behaviors: Classes 348
7.3 A simple list class with an array data member 359
7.4 Constructors revisited 371
7.5 A class with string data 379
7.6 Classes with class type member data 381

7.7 An inventory table class 384
7.8 The changemaker revised—with classes 393
7.9 Organization of program source code and libraries 402
7.10 Sharing constant class data 407
7.11 Summary 411
7.12 Exercises 414

Chapter 8 **First Class Objects** 429

8.1 A numerical example: Rational numbers 430
8.2 A second example: Three-dimensional vectors 448
8.3 Designing first class types 457
8.4 Efficiency issues: Inlining 459
8.5 Summary 462
8.6 Exercises 463

Chapter 9 **Storage Management for Objects** 471

9.1 Pointers 472
9.2 Dealing with failures: `exit()` and `assert()` 487
9.3 References 489
9.4 Dynamic array creation 492
9.5 First class strings 499
9.6 Extensible arrays 522
9.7 Implementation standards for classes with pointers data 529
9.8 Summary 531
9.9 Exercises 532

Chapter 10 **Inheritance and Reuse** 543

10.1 Incremental programming: Extending an existing class 544
10.2 Single inheritance: Designing a class hierarchy 553
10.3 Multiple inheritance 569
10.4 Inheritance as an organizational tool 577
10.5 C++ streams: Built-in inheritance 579
10.6 Changing the changemaker solution to incorporate inheritance 588
10.7 Summary 604
10.8 Exercises 606

Chapter 11 **Polymorphism and Generic Classes** 615

11.1 Ad hoc polymorphism through overloading 616
11.2 Parameterized polymorphism through templates 618
11.3 Runtime polymorphism 629
11.4 Abstract base classes 648

11.5 Polymorphism and cloning for array-based lists 663
11.6 The changemaker, revisited again 670
11.7 Summary 678
11.8 Exercises 680

**Chapter 12 Container Classes and Linked Lists** 691

12.1 Examples of containers 692
12.2 Linked lists and traditional linked list algorithms 694
12.3 Implementing a linked list type `List` and `Node` classes 703
12.4 Iterator classes 715
12.5 List variations using inheritance and templates 721
12.6 Stacks and queues 733
12.7 Message passing: Letting the nodes do it recursively 743
12.8 Tables 756
12.9 Summary 770
12.10 Exercises 772

**Chapter 13 An Example Reconsidered: Some Issues in Design** 783

13.1 The `Typesetter` class 784
13.2 *Has-a* vs. *is-a* relationships 802
13.3 Memory leaks: A common flaw 804
13.4 ATNs for specification and implementation correctness 806
13.5 Separating public interface from implementation details 819
13.6 Summary 821
13.7 Exercises 822

**Appendix A: ASCII Character Codes and Escape Sequences** 825

**Appendix B: C++ Keywords** 827

**Appendix C: Operator Precedence** 829

**Appendix D: Stream Input** 831

**Appendix E: Stream Output** 839

**Appendix F: File Streams** 847

**Bibliography** 857

**Index** 859

Chapter 1

# *Object-Oriented Program Development*

**Chapter Contents**

1.1 What is object-oriented program development?
What is an object?
Connections between computer-world objects and real-world objects
Object-oriented analysis and object-oriented design
Lessons for students

1.2 Sample problem: The convenience store
Analyzing and designing a solution
Implementing the solution

1.3 Sample problem: The changemaker
Analysis and design
Algorithms
Implementation of the changemaker solution

1.4 Compilers and languages: Making the program run

1.5 Why is object-oriented development so important?

1.6 Summary
Glossary

1.7 Exercises

COMPUTERS are not smart; they just know how to follow instructions. Unlike dogs or cats or people, computers follow instructions very well, step-by-step. A computer **program** is a set of instructions to the computer. The purpose of this book is to teach you to write computer programs that will tell computers how to solve particular problems or do particular things.

This chapter begins with a look at how the entire process of creating a computer program originates. We will follow a special approach called **object-oriented**

**program development**, an approach that is somewhat new in the computing world but very old in the world of human experience. It involves creating models of real-world situations and building computer programs based on those models.

The best way to learn about object-oriented development of solutions to programming problems is through examples. There are no generic recipes that are going to work with all problems. Computer problem solving is a creative activity. It can be very simple or very complex, depending on the nature of the problem. And that's what makes it interesting. If solving problems were just a matter of following a simple list of instructions, then we wouldn't need programmers at all. We'd just write one program that would be able to write all other programs!

We'll examine two different problems to illustrate object-oriented development concepts, solving them step-by-step, from the beginning. First, we analyze the problems, then we design and draw out solutions on paper. We finish by writing the computer programs that are the actual problem solutions. The focus for this chapter is on analysis and design, because that constitutes the hard part of problem solving.

In this book we'll show you how to "object-think" when you are at the analysis or design levels of solving programming problems. We'll introduce specific details of how to code your solutions for the computer using the programming language C++. And we'll approach C++ programming tools gradually. Each program that you will encounter while studying this text uses the concepts of C++ that you already know or will soon learn. As you progress with your mastery of the features available in C++, you will see how some of the initial designs for problems can be changed to take advantage of increasingly sophisticated features of the language.

## 1.1 What is object-oriented program development?

A program is somewhat like the script of a play. A script describes the cast of characters, the environment in which they exist, and the things the characters do. The sets and the props of a play represent physical things, and the characters represent people that seem real (usually). The interactions among the characters represent situations and events. In many cases, computer programs contain computer-world representations of the things or objects that constitute the solutions of real-world problems. Executing a program such as this is like performing a play.

Object-oriented program development means constructing programs as models of real-world events. The entire programming process begins with construction of a model of the event. The end result of the process is a computer program that contains features representing some of the real-world objects that are part of the event. Execution of the program simulates the event.

Most of the things that you are asked to do in a beginning programming course are things that you know how to do without a computer. You know how to alphabetize a list of names, convert temperatures from Fahrenheit to Celsius, and determine the cost of a shopping cart full of groceries. But unless you've programmed before, you probably don't have the foggiest idea of how to make a computer solve these fairly simple problems. You probably don't even know where to begin.

As our first step in program construction, we'll look at an intuitive way to describe what is to be done. This description has nothing necessarily to do with computers. At this point, we will pay very careful attention to the objects involved in our intuitive solution and their behaviors. We will rely on our own real-world experiences and intuition for guidance.

Once we figure out what a program should do, we will be ready to create a model. Problem solving through model construction is not a new kind of intellectual activity. It's been going on since the beginning of civilization. Today we all use models constantly. Think about traveling by automobile from one city to another. Highway maps are models we use to solve the problem of getting from one geographical region to another. The objects that make up a highway map include representations of such things as cities, roads, and rivers. The relationships among these objects determine the routes that we can travel to go from one place to another.

What kinds of models can we create for computer programs? There are countless answers to this question. The basic requirement of any model is that it be useful. There can be many different models that will be useful in a given programming problem or task. Some models may even be radically different and still be useful. The modeling process itself is not rigid. For any one problem there is no unique answer to what model should be constructed and what objects must be involved.

Let's look at a list of problems that might be solved by computer programs. Think about each in terms of what situations the corresponding computer program can model. Imagine a cartoon for each situation. What are the objects in the cartoons? How do they interact? Some models are better than others. The best thing you can do is to pick out what you think is most useful. It takes experience and creativity to think up useful models.

- A video game software manufacturer wants to create a new video game that simulates a jet fighter.
- A physician needs software tailored to patients at her medical clinic. The patients have many varieties of illnesses and funding sources for their treatments.
- The Environmental Protection Agency needs a program that simulates acid rain deposition in the Appalachian Mountains under certain power plant location scenarios.
- An inventor is building a wheelchair that can climb stairs. The wheelchair must be simulated in software before the inventor is willing to pay for expensive tooling of the wheels, gears, and microprocessor that constitute the actual wheelchair.

You can't solve these problems now, but you can begin to think about the scenarios that they model. The video game will simulate interactions among fighter jets and is especially easy to imagine as a cartoon. The physician's software will take the place of the tedious bookkeeping and record keeping that used to be relegated strictly to hand calculations and notations. Acid rain deposition, which will take place as a story from the computer rather than as real rain in the Appalachian Mountains, will involve computer-world power plants and computer-world trees. The wheelchair simulator will be able to show how well a particular wheelchair design fits the specifications of the inventor.

The corresponding programs to solve these problems will differ immensely in length and complexity. Even the effort required to test the solutions to make sure they are correct will differ greatly among the programs. A program is **correct** if it does what it is supposed to do. The correctness of software may have an enormous impact on its users or it may be of relatively trivial significance. If the video game fails to keep correct scores for its players, the results may be of little consequence. But the failure of the wheelchair to climb a set of stairs properly could result in serious injury for the person in the wheelchair.

When you first start solving programming problems, your efforts will be very modest. You will not be able to solve the problems we just listed, and you will probably spend a considerable amount of time just figuring out how to write your solutions in C++.

C++ is a computer language specifically designed to support object-oriented program development. The new language tools that you will learn won't tell you how to solve the problem. But you will work within the confines of these new tools in order to create programs that will implement your solutions, programs that will actually solve the problems and perform the desired tasks.

## What is an object?

In our discussions of problems and models, we will use the term **object** over and over again. The term object has the same meaning as a noun or noun phrase: It is a person, place, or thing. Examples of real-world objects are endless: person, table, vending machine, airline flight schedule, computer, dictionary, city, and the ozone layer to name a few.

All of the example problems in this chapter can be described in terms of real-world objects—from jets to ledgers to people to wheelchairs. When we look at a problem or situation to figure out what a program solution is supposed to do, we will do so in terms of the objects that make up the problem. We will identify those objects in terms of what they are, how they behave, and how they interact with each other.

Some objects—airplanes, people, and major cities—are very complicated, others are very simple. An object may include other objects. For example, a personal computer is an object. One of the objects that makes up the computer is the central processing unit (CPU). Another object is its main memory and another is the