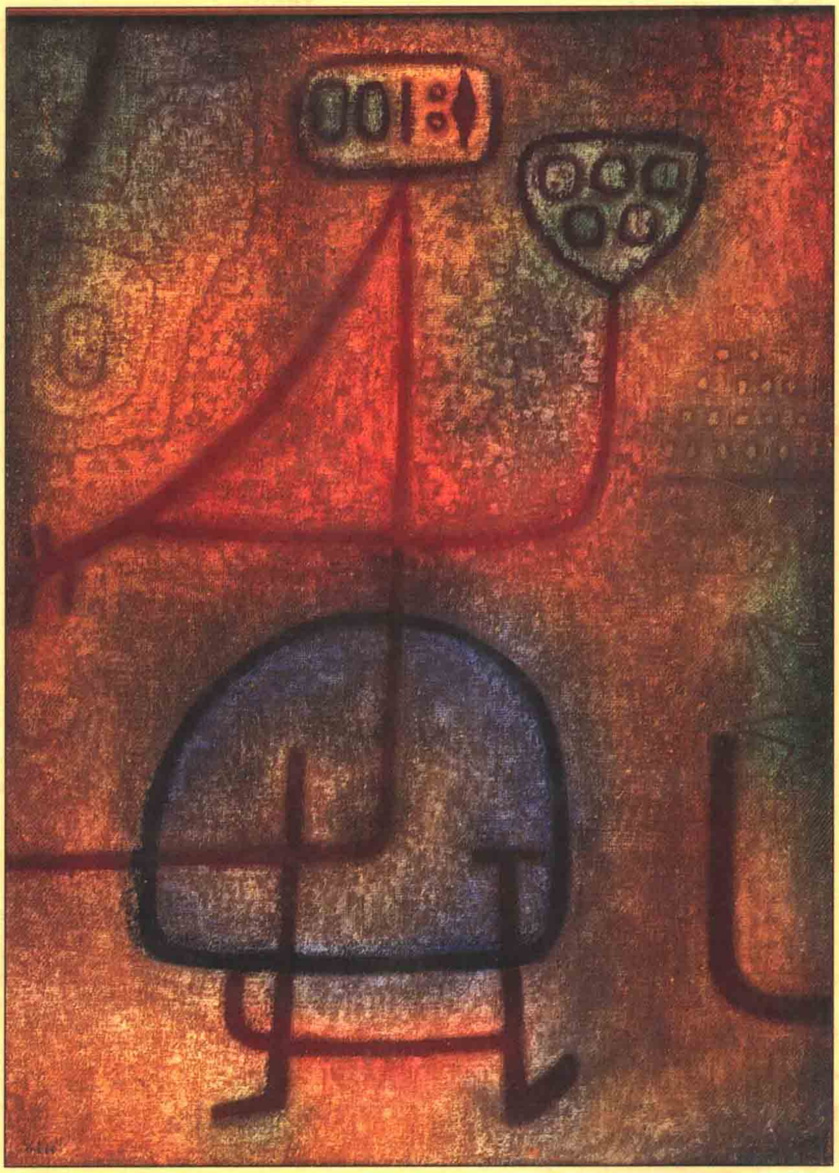
SECOND EDITION

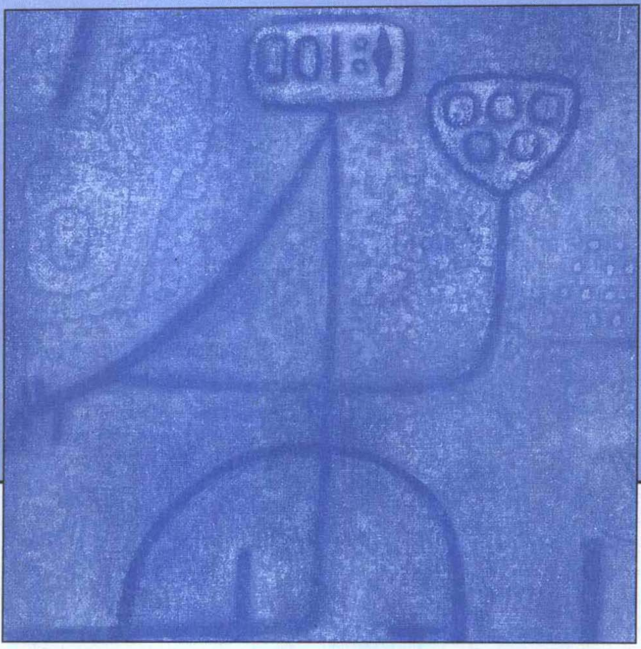
STRUCTURES AND ABSTRACTIONS

AN INTRODUCTION TO COMPUTER SCIENCE WITH TURBO PASCAL, (5.X, 6.X, 7.0)



WILLIAM I. SALMON

WILLIAM I. SALMON
University of Utah



STRUCTURES AND ABSTRACTIONS
An Introduction to Computer Science

IRWIN
Burr Ridge, Illinois
Boston, Massachusetts
Sidney, Australia

Dedicated to
the Spirit of Liberal Education,
wherever it survives.

Cover painting:
Copyright 1992 by VAGA New York
Paul Klee (1879–1940)
La belle jardiniere
(ein Biedermeiergespenst), 1939.1237
Reproduced by permission.

Senior sponsoring editor: Tom Tucker
Developmental editor: Lena Buonanno
Marketing manager: Robb Linsky
Project editor: Denise Santor
Production manager: Laurie Kersch
Compositor: Bill Salmon, *Interactive Educational Systems*
Cover designer: Amy Osborne
Printer: Von Hoffman

Library of Congress Cataloging-in-Publication Data

Salmon, W. (William I.)
Structures and abstractions : an introduction to computer science with Turbo Pascal / William I. Salmon. — 2nd ed.
p. cm.
Includes index.
ISBN 0-256-12667-4
1. Pascal (Computer program language) 2. Turbo Pascal (Computer file) 3. Data Structures (Computer science) 4. Abstract data types (Computer science) I. Title.
QA76.73.P2S253 1994
005.26'2—dc20 93-38983

Printed in the United States of America
1 2 3 4 5 6 7 8 9 0 VH 8 7 6 5 4 3 2 1

Preface

It is a pleasure to respond to the teachers and students who used this book in its first edition and to incorporate many of the improvements they suggested. As in the first edition, this book is intended for a first college-level course in computer science, emphasizing modern software engineering practice. The main issues are procedural and data abstraction, modular and hierarchic software design, program structures and data structures, and first glimpses of informal verification and complexity analysis. The goal is to provide a survey of the important concepts in our field, suitable for courses lasting from one quarter to two semesters. Pascal is the illustrative programming language, but the language is not the main mission, and there is no attempt to cover all of Pascal's syntax. The spirit here has been much influenced by the new ACM/IEEE curriculum guidelines.

What's new in this edition?

The general principles of software engineering and abstraction are introduced at the beginning of Chapter 2, as a lead-in to the explicit techniques for problem solving. The problem-solving techniques now make explicit use of procedural abstraction and encapsulation. New case studies in this chapter set up the solutions of two problems to be attacked in Pascal in later chapters.

The material on text files has been removed from Chapters 4 and 10 and placed in Chapter 17, although it still features a modular, three-level organization. The new organization allows text file material to be taught either early or late in the course, at the discretion of the instructor. To introduce text files early, simply cover Sections 17.1 and 17.2 right after Sections 4.3 and 4.4, respectively, and Sections 17.3–17.6 right after 10.3. (This sequencing is clearly marked in the text and Table of Contents.) Then the remaining sections of Chapter 17 can be covered any time after Chapter 14. Further details appear after this preface, under the heading *Teaching from This Text.*

The sections on testing and debugging, at the ends of many chapters, now contain extensive lists of actual error messages resulting from example errors, using a real-world compiler (Borland/Turbo Pascal 7.0). Other Pascal systems give similar messages in most cases.

An optional Section 5.6 has been added, explaining how to avoid global variables entirely, when this is desired.

A new Chapter 6 introduces the `IF..THEN..ELSE` and `WHILE..DO` control structures, so that students can use decisions and loops at an earlier point in the course. This is done with minimum syntax, to keep attention focused on procedural abstraction at this point in the course.

An optional Section 6.5 introduces the concept of recursion, for those instructors wishing to raise the issue this early, and for those students who wonder what happens if a procedure calls itself. The full treatment of recursion appears in Chapter 12, right after the full treatment of iteration. The chapter on recursion provides more exercises and projects than before.

The coverage of `CASE` structures has been expanded in Section 9.4.

In Sections 11.1–11.4, the explanations of loop invariants have been rewritten to make it clearer how a loop is designed to terminate correctly.

A case study on selection sorting has been added to the chapter on arrays (Chapter 14). The chapter on searching and sorting (Chapter 19) now has a section on insertion sort. This means that the text now provides complete coverage of selection, insertion, and quicksorts. Chapter 19 can now be covered right after Chapter 14 if the instructor desires.

The first example of an abstract data type is now a `Fraction` type instead of `String` (Section 16.2).

Section 19.3, on big-oh notation, now provides an informal heuristic explanation in addition to the formal mathematical one.

In the Turbo Pascal edition, Chapter 21 now introduces object-oriented programming (OOP), using lists, stacks, and queues as examples.

About a third of the questions, exercises, and projects in the book are new.

There are 14 new sidebars, averaging almost a page in length, providing a glimpse of some of the frontiers in computer science. These are substantial quotes from journals, magazines, and books, chosen for their provocative, often controversial ideas and engaging writing styles. They are intended to convey the excitement and rapid changes in computer science, while showing the pervasive effects of computing on our everyday lives and even on our philosophical outlooks.

Themes that continue from the first edition

- Students practice procedural abstraction from the beginning of the course. Algorithms are presented from a hierarchical viewpoint that encourages modular design from the very beginning, with constant emphasis. Unlike most other books, there is no need to apologize later for early monolithic programs, or to "unteach" early bad habits.
- Chapter 2 provides eight explicit problem-solving techniques before students begin Pascal coding. These techniques are used repeatedly throughout the rest of the book and become so ingrained that they provide a foundation for creative solutions to new problems.
- Students are taught to picture both the data structures and the actions that occur in a program. The book contains many "animations" (snapshot sequences) of program execution and pictures of the data structures—380 diagrams in all. (For an example, see Section 6.4.)

- Good programs don't work correctly by luck; they are engineered so that they have to work correctly. We introduce simple techniques for checking the correctness of algorithms before writing programs (Chapter 11).
- Recursion is not more mysterious or difficult than iteration—it's just less familiar. These two techniques for repetition are equally important and each illuminates the other. Therefore we present recursion and iteration side by side, with frequent comparisons, beginning in Chapter 12.

Meeting the ACM/IEEE guidelines for the 1990s

Structures and Abstractions satisfies the new curriculum recommendations of the Association for Computing Machinery (ACM) and the Institute of Electrical and Electronic Engineers (IEEE) for a first course in computer science (CS1). In addition, Parts 4 and 5 of the book overlap the latest ACM/IEEE guidelines for CS2 (data structures) courses. In particular, the book emphasizes the three processes of theory, abstraction, and design, while being accessible to first-year students. (See the publication, *Computing Curricula 1991*, ACM Order Number 201910 or IEEE Computer Society Press Order Number 2220.)

Structures and Abstractions provides enough material for a variety of introductory courses, ranging from a single quarter to two semesters in length. There is considerable optional material to allow for various approaches in different teaching situations. Some of the teaching options are discussed under the heading, *Teaching from This Text*, following the preface.

The standard edition of *Structures and Abstractions* uses (except in one appendix) only ISO/ANSI Standard Pascal. All examples have been tested in several typical Pascal environments to ensure that they run correctly. A separate edition is available for those preferring a treatment specific to Borland International's Turbo Pascal and Borland Pascal, using compilation units and objects to implement abstract data types.

Borland/Turbo Pascal 5.x/6.x/7.x

This edition of *Structures and Abstractions* uses Borland International Corporation's *Borland Pascal*® or *Turbo Pascal*®, showing the use of versions 5.x, 6.x, and 7.x of that environment as well as the new *Turbo Pascal for Windows*® (TPW). A separate version of the book is available for those preferring to use a Standard ISO treatment appropriate to other Pascal compilers.

Who can read this book?

The reader needs no previous programming experience, but should be computer-literate, with enough experience in mathematics to appreciate the need for rigorous thought and to understand algebraic proofs. *I have found that the best predictors of success in a CS1 course are skills in mathematical proofs and word problems and an ability to communicate clearly in writing.*

Specific features

Consistent emphasis on engineering and design: *Structures and Abstractions* places heavy emphasis on the fundamental techniques for proper software engineering, including procedural and data abstraction, top-down modular design, incremental testing, the use of assertions and loop invariants, and elementary running-time analysis. Modular, hierarchical design is constantly emphasized from Section 2.2 onward, even before we begin to introduce the Pascal language. Chapters 3 and 4 introduce Pascal by means of parameterless procedures, real-number data, and simple I/O, while constantly emphasizing procedural design—an emphasis that continues through Chapters 5–7. The consistent early emphasis on procedural abstraction has the advantage that hardly any monolithic programs appear in the book. Students learn good design practices from the very beginning. Similarly with data abstraction, which begins in Chapter 16: Once we begin to use abstract data types, we continue to use them in Chapters 17, 18, 20, and 21.

Classic algorithms: A first course should introduce its students to many of the classic algorithms on which later courses will build. This book includes base conversion, case mapping, counting characters and words in text, exponentiation, greatest common divisor, square roots, Towers of Hanoi, expression parsing, line drawing, string manipulations, binary search, selection sort, insertion sort, quicksort, list processing, evaluation of postfix expressions, pseudorandom number generation, and simulation. Additional case studies are provided in the lab manual.

Style: The book itself reflects the rules of good programming style. For example, subjects like modularity, procedural abstraction, data abstraction, recursion, loop invariants, and dynamic data structures are introduced by themselves in their own sections and chapters, so that the reader can focus full attention on them. Then each topic is learned "bottom up," with a sequence of stepped examples that gradually increase in complexity and abstraction. Like the programs themselves, chapters are short and modular, divided into short sections with clear goals. More advanced topics likely to be skipped in shorter courses are isolated in optional sections, often toward the ends of chapters or in later chapters. The modular design of the book results in more chapters, sections, and subsections, but these are shorter, clearer, and more flexible in use.

Explicit problem-solving techniques: Classroom testing has proven that students benefit from the presentation and repeated use of explicit methods for problem solving. Chapter 2 presents eight of the most generally useful. These techniques are illustrated with two nontrivial case studies in Chapter 2, and then are used repeatedly throughout the following chapters. By seeing the same techniques applied in various situations, readers gradually learn their use.

A spiral, not a "peek-a-boo" approach: Students need to see why the rules of practice and style are necessary. The book introduces techniques when they are needed, then uses them persistently and consistently. The reader never loses sight of a topic while seeing it unfold, because an important topic is never dropped after being introduced. The gradual deepening and continual exposure to abstraction deepens readers' understanding and appreciation of each topic.

Pedagogical aids

Chapter outlines and summaries: Each chapter begins with a few introductory paragraphs connecting the chapter with previous material and outlining the topics to be covered. The chapter ends with a summary of the main points that were made.

Important terms defined: Important technical terms are shown in boldface when they first appear. If they are likely to be unfamiliar, they are defined in the margin and listed in the glossary. Many are also listed at the end of each chapter, where they serve as a guide to concepts that should be reviewed for exams.

Visualization: Sequences of execution during procedure calls, iteration, recursion, and other complex actions are "animated" by sequences of diagrams acting as "snapshots" of the execution process. Altogether, there are more than 35 such animations, containing more than 195 diagrams. In addition, there are hundreds of diagrams of syntax and data structures—over 380 in all.

Exercises and programming projects: Questions and exercises are distributed throughout the chapters, immediately after material requiring practice and reinforcement. The questions tend to be of the simple, self-check type. Exercises range in difficulty from trivial syntax practice to typical debugging experiences and short programming assignments. Major programming projects are found at the ends of the chapters. I have tried to provide a wide range of problems to meet the needs of various introductory courses. Altogether, there are more than 650 problems, occupying over 100 pages.

Case studies: Important programming issues are illustrated by application to typical problems in computer science. These include first glimpses of many of the kinds of problems to be encountered in later courses, and are used to teach explicit techniques of problem solving as well as informal methods of verification and analysis. (Additional case studies can be found in the companion lab manual.)

Debugging aids: Debugging techniques and examples are discussed frequently. In addition to standard techniques involving modular testing and intermediate output, typical interactive debugging tools are described. Always, the reader is reminded that it is most important to prevent bugs in the first place, by good engineering practices. (The companion lab manual also provides considerable guidance with debugging, along with instructions for using the THINK and Borland/Turbo debuggers.)

Sidebars on important issues: At the ends of many chapters, you will find sidebars illustrating some of the issues that swirl through computer science. These extended quotes from a variety of professional and popular sources are intended to stir discussions in class and to encourage readers to examine their own attitudes toward important computer-related issues of our time.

Acknowledgments

Many reviewers have helped to shape this new edition. Their detailed and thoughtful suggestions along the way are much appreciated. We thank

Jay Martin Anderson, Franklin and Marshall College
Laurie Benaloh, Clarkson University
David Berque, Colgate University
Jack V. Briner, Jr., University of North Carolina–Greensboro
David T. Brown, Ithaca College
John F. Buck, Indiana University, Bloomington
Deborah L. Byrum, Texas A&M University
Adair Dingle, Lehigh University
Ron Gilster, Walla Walla Community College
Wilbur Goltermann, University of Colorado–Denver
Ramon P. Hernandez, Mesa Community College
Kip Irvine, Miami-Dade Community College
Danny Kopec, Carleton University
Cary Laxer, Rose-Hulman Institute of Technology
Lewis Lum, The University of Portland
Rebekah D. May, Ashland Community College

Jack Mostow, Rutgers, The State University of New Jersey
Debbie Noonan, College of William and Mary
Jandelyn Plane, University of Maryland–College Park
Dennis E. Ray, Old Dominion University
Charles W. Reynolds, James Madison University
Ali Salehnia, South Dakota State University
Louis Steinberg, Rutgers, The State University of New Jersey
Christopher J. Van Wyk, Drew University

In addition, the influence of the first-edition reviewers is still felt, and we thank them again: Robert B. Anderson, University of Houston; Brent Auernheimer, California State University, Fresno; Anthony Q. Baxter, University of Kentucky; Louise M. Berard, Wilkes College; David Alan Bozak, SUNY College at Oswego; Larry C. Christensen, Brigham Young University; Robert A. Christiansen, University of Iowa; Denis A. Conrady, University of North Texas; Cecilia Daly, University of Nebraska–Lincoln; Douglas Dankel II, University of Florida; Edmund I. Deaton, San Diego State University; H. E. Dunsmore, Purdue University; Suzy Gallagher, University of Texas; David Hanscom, University of Utah; Robert M. Holloway, University of Wisconsin–Madison; Ronald P. Johnson, Evangel College; George F. Luger, University of New Mexico; William E. McBride, Baylor University; Michael G. Main, University of Colorado at Boulder; Andrea Martin, Louisiana State University; Jane Wallace Mayo, University of Tennessee–Knoxville; Kenneth L. Modesitt, Western Kentucky University; David Phillips, University of Pennsylvania; George A. Novacky, Jr., University of Pittsburgh; David L. Parker, Salisbury State University; Theresa M. Phinney, Texas A&M University; V. S. Sunderam, Emory University; Stephen F. Weiss, University of North Carolina–Chapel Hill; Laurie White, Armstrong State College; Stephen G. Worth III, North Carolina State University–Raleigh; and Marvin Zelkowitz, University of Maryland.

Many teaching assistants and students have helped with the evolution of this book. I would particularly like to thank my teaching assistants, Rich Thomson, Cliff Miller, Elena Driskill, Rory Cejka, Mark Ellens, Mike Stephenson, and Lynn Eggli. I would also like to thank all the students who made suggestions and corrections, especially Alexander Kratsov, Randy Veigel, Mark Nolan, Blair Brandenberg, Ian Adams, David Swingle, and Lisa Clarkson. Over the years, I have received many helpful suggestions from John Halleck and LeRoy Eide of the University of Utah Computer Center and from my wife, Lydia Salmon. They too deserve effuse thanks.

For many years I have received encouragement and inspiration from Dave Hanscom, the undergraduate coordinator in the University of Utah's Computer Science Department. I also owe a special debt to my son, Edward Salmon, whose superb sense of design influenced the cover and several of the diagrams and projects in this book.

Many thanks to the talented people at Richard D. Irwin, Inc., who provided more support and help than I thought possible. I particularly thank Sheila Glaser, Max Effenson, and Lena Buonanno, the developmental editors, who cheered me along while trying to convince me that *but* is only a conjunction.

But I should also mention Jackson P. Slipshod, who makes frequent appearances in the questions and exercises in this book. To the best of my knowledge, he made his first appearance in a chemistry book by Joseph Nordmann, published many years ago by John Wiley & Sons, Inc. Thanks to Dr. Nordmann's fine book, Jackson has been dogging me ever since.

Questions? Requests?

I enjoy feedback from readers, and have received some of the best suggestions in this way. You can send comments, gripes, corrections, and compliments to the following e-mail addresses. (But please don't ask me to help with your homework!)

William I. Salmon
Internet address: salmon@cs.utah.edu
CompuServe address: 71565,135

TEACHING FROM THIS TEXT

This book supports a variety of college-level courses, from one quarter to two semesters in duration. The flexibility is achieved by designing the book around a core of essential chapters—Chapters 3–10 and 13–15—while also supplying a number of more independent chapters that cover discretionary topics required by some courses but not others. Few instructors will want to cover the whole book, so the following explanations are intended to help in choosing material to suit particular courses and teaching styles.

Chapter 1: Computing and Computation

The goals of the first chapter are to define and explain the concept of an algorithm, to explain what it means to say that a computer is a general-purpose symbol-manipulating machine, and to introduce the kinds of software used in translating algorithms into programs, compiling them into machine language, and executing the resulting machine code. Many but not all of the terms in this chapter should be familiar from previous computer literacy experiences, but this chapter is necessary in order to establish a common vocabulary and conceptual basis for the rest of the book. In my one-quarter classes, time does not allow me to lecture on this material, but I assign it for outside reading, warning students that they will be responsible for the contents of the chapter. Of course, I answer in class any questions that arise from the readings.

Chapter 2: Abstraction, Problem Solving, and Algorithm Design

Sections 2.1 and 2.2 introduce the complexity of modern commercial software systems and the need for abstraction to handle such complexity. In particular, Section 2.2 describes procedural abstraction, describing how actions are encapsulated in procedures. We then apply this way of thinking while introducing (in Sections 2.3–2.12) eight explicit techniques for inventing algorithms to solve problems. One of the techniques is the important principle of top-down design. Sections 2.3–2.6 begin a Fahrenheit-to-Celsius temperature conversion case study, which uses real-number data. This evolving case study is used to provide a unifying thread in Sections. 4.1, 4.5, 4.6, 4.7, 5.1, 5.2, 6.1, 6.2, and 6.7, as the topic of procedural abstraction is pursued more deeply. Section 2.7 applies further problem-solving techniques to the problem of sorting three numbers. This second case study is a setup for Section 6.3, where the algorithm is coded into Pascal. Often I do not lecture explicitly about this chapter, assigning it, along with Chapter 1, for reading during the first week of class.

Chapter 3: Program Structures

Chapter 3 is where I usually start explicit coverage in class. This chapter uses the abstraction concepts and problem-solving techniques of Chapter 2 while introducing the syntax of Pascal programs. The goal is to show how procedural abstraction is practiced in a programming language, and to show how procedural abstraction involves modularity and hierarchy. Pascal procedures are used, but without parameters. The procedures draw pictures but do not use numeric or character data. I do not give formal lectures on syntax as I cover this material; instead, I act out the solution of the problems in great detail. To cover Section 3.4, for example, I state the problem, then I describe how the output might look (one of the explicit problem-solving techniques of Chapter 2), then I describe the sequence of steps in executing an appropriate program, then I outline the algorithm, and finally, I build the syntax of a program. The last stage of this process will generate many questions about syntactic details, and I answer all of these, also discussing alternatives at every stage.

Chapter 4: Real Data and I/O

Here we introduce real-number data, explaining how it is stored, why this results in roundoff error, and how we perform real-number input and output (I/O). The goal is to introduce a data type and explain its use without distracting from the emphasis on procedural abstraction. (If we introduced integer, character, and boolean data at this point, together with their associated syntax, we would sacrifice the constant emphasis on procedural abstraction and encapsulation.) The chapter ends with the beginnings of a modularized program for Fahrenheit-to-Celsius conversion, as a motivator for the parameters to be introduced in Chapter 5. Again, I teach this material by acting out the solutions of the problems. When I get to the Pascal-coding stage, I often make "accidental" syntax or semantic errors to show what happens, how I locate the errors, and how I fix them. (I use a microcomputer with LCD overhead projection whenever possible, so that students can see me do these things in real time. When I have no projector, I simulate events on the blackboard.)

Instructors who prefer to introduce text file I/O can do so at this point. Although all file I/O material is collected in Chapter 17, it is designed for flexible use. Sections 17.1 and 17.2 are designed so that they can be covered *immediately after* Sections 4.3 and 4.4, if desired.

Chapter 5: Procedures with Parameters

Now we take the procedural syntax and top-down design principles from Chapter 3, combine them with the real-number example from Chapter 4, and add new syntactical material on using parameters with procedures. The first case study is the Fahrenheit-to-Celsius problem carried over from Chapter 4. This modularized version is now to the point where it needs protection from bad input data, and this brings up the need for control structures, which will be the topic of Chapter 6. Section 5.4 introduces the important concept of program states, explaining that the purpose of a procedure is to map a precondition state into a postcondition state. Pre- and postconditions are written for most procedures from this point on, allowing students to ease into this idea before we use it in the introduction of loop invariants in Chapter 11. Instructors who don't plan to cover invariants can downplay the emphasis on pre- and postconditions. (In my experience, students will ignore topics for which they are not held responsible!)

Chapter 6: Controlling Execution

Section 6.1 introduces a simple form of the IF..THEN..ELSE control structure and uses it to protect Chapter 5's Fahrenheit-to-Celsius program from bad input data. This improvement works, but what we really need is a controlled way to repeat execution. Section 6.2 therefore introduces a simple version of the WHILE..DO control structure and uses it to build a final version of the F-to-C program. Section 6.3 examines in Pascal the problem of sorting three numbers, which was attacked in Section 2.7 as an example of algorithm design. This case study is the vehicle for introducing the use of stubs and drivers when practicing incremental testing of programs during construction. It also provides an unusually good example of what is involved when one tries to test a program with all possible combinations of input data, and it motivates the need for the more formal verification methods to be introduced in Chapter 11. Chapter 6 also shows how nested procedure calls work, because this issue arises naturally in the three-number sorting problem. The chapter ends with an optional section on recursion because students often ask at this point what happens if a procedure calls itself.

Chapter 7: Functions

After a short introduction of the concept of a function, Section 7.2 contrasts the data-return mechanism of a Pascal function with that of a procedure. Section 7.3 explains how to decide which to use in a given case. The chapter is located here because functions are needed in Sections 8.2, 8.5, and 8.7.

Chapter 8: Ordinal Data Types

Now we have finished introducing procedural abstraction and have practiced top-down, hierarchical design, so we are free to wade through the syntax associated with all the ordinal data types. Section 8.3 presents a major case study (decimal-to-binary conversion) using integer data. The algorithm design involves careful specification of pre- and postconditions for all procedures and functions, together with specifications of the encapsulations in each and descriptions of the state transitions. (Pre- and postconditions were introduced in Section 5.4; encapsulation has been emphasized since Chapter 2.) The development of the Pascal code emphasizes incremental testing with a stubs and a driver, a concept that was introduced in Section 6.3. We also discuss how we know that a loop will necessarily terminate (a major topic of the optional Chapter 11). Section 8.5 works out a case study involving characters, functions, stubs, and an IF..THEN..ELSE structure. It also provides a set-up for the boolean expressions introduced in Sections 8.6 and 8.7. The case study on character-property functions (Section 8.7) is a further example of hierarchical design and the replacement of decision structures with boolean expressions. The chapter ends with a thorough explanation of operator precedence, involving all simple data types.

Chapter 9: Decision Structures Again

With the presentation of procedural abstraction complete, Chapters 9 and 10 cover decision structures (IF..THEN..ELSE and CASE) and repetition structures (WHILE..DO, FOR..DO, and REPEAT..UNTIL) in complete detail. (Simple forms of IF..THEN..ELSE and WHILE..DO were introduced in Chapter 6. The complete treatment takes 62 pages and would have distracted considerably from the presentation of procedural abstraction if introduced then.) Section 9.1 presents a second version of the Sort3 procedure, using the nested procedure calls from Section 6.4 and now some new IF..THEN..ELSE syntax. Boolean expressions are used to replace IF..THEN..ELSE structures in Section 9.1, as they were in Section 8.7. Nested decisions are introduced in Section 9.2 because they are needed in the menu case study of Section 9.3. This case study in turn leads into a discussion of Pascal's CASE structure in Section 9.4, and this in turn leads to a need for guard loops, as introduced in Chapter 10.

Chapter 10: Repetition by Iteration

When WHILE loops are re-examined in Section 10.1, we can discuss their use of boolean expressions, which was not possible in Section 6.2. We then consider how to count characters in input, using a sentinel to control the loop. This case study involves the concept of the Input file and file buffer (Section 4.4) and uses a stub and driver in the testing of the code. Section 10.4 discusses counter-driven pretest loops, as a motivation for the FOR structure introduced in Section 10.5. The importance of top-down design comes up again. The FOR loop example in Section 10.6 involves ordinal values and character data. Section 10.7 presents a case study involving a Monte Carlo calculation, and is optional. The case study is our first example of using a pseudorandom number generator, a topic that comes up in several later programming projects and in Chapter 21. The REPEAT..UNTIL structures of Section 10.8 are applied to the problem of processing menu selections, completing some work that began in Section 9.3. Like Chapter 9, this one will be covered completely in most courses.

Instructors who wish to use text file I/O at this point can cover Sections 17.3–17.6 following Section 10.3. Chapter 17 has been designed to allow this option.

Chapter 11: Iteration by Design (Optional)

This is an optional chapter that introduces the rudiments of formal loop design, using loop invariants and termination conditions. A variety of case studies is provided to meet the needs of different instructors. One-quarter or one-semester courses will probably not have time to cover more than Section 11.1 and one of the other case studies.

Section 11.5 is distinct from the rest of the chapter. It shows how the running times of nested loops can be expressed as functions of loop parameters, and is intended as early preparation for those intending to cover big-oh notation in Section 19.3.

Don't try to cover too much in this chapter, and allow sufficient time for the material you do cover. The chapter applies mathematics in ways that will be new to many students, and the material will take time to assimilate. Instructors preferring to skip this chapter can do so without losing continuity.

Chapter 12: Repetition by Recursion (Optional)

Recursion takes a long time to sink in, so it is introduced in Chapter 12 and revisited frequently, with heavy use of animation diagrams like those used earlier for nested procedure calls. Iteration and recursion are treated as equally important techniques for repetition, and often, both recursive and iterative versions of an algorithm are examined.

Some instructors prefer to postpone this topic, but others (including me) prefer to present recursion and iteration side by side for direct comparison. In presenting recursion, I emphasize repeatedly that there is nothing new here: A procedure is calling a procedure, as in previous chapters, but now the procedure being called happens to have the same name and to contain the same code as the caller. Working carefully through the execution snapshots (the "animations") of Section 12.1 and one of the case studies goes a long way toward dispelling the mystery that many students attach to this subject. As with the previous chapter, this one provides more material than any one instructor is likely to use, in order to allow flexibility and choice. In my one-quarter course, I usually skip Sections 12.4 and 12.6, but I make sure to mention that every loop can be rewritten recursively, and vice versa; and that some problems are more easily and clearly coded iteratively, while others are more easily and clearly coded recursively. I also always discuss comparative running times, as in Table 12.3.1 and Section 12.7.

Chapter 13: Programmer-Defined Types

Sections 13.1 and 13.2 must be covered before the next chapter, because subrange types are used for array indexes in Chapter 14. The material on enumerated types could be postponed, but why bother? Chapter 13 doesn't require much time in class, so I cover it by working out a couple of case studies.

Chapter 14: Arrays

This is a large chapter with several options. After one-dimensional arrays (Sections 14.1–14.3), many instructors like to discuss sorting, so Section 14.4 examines the selection sort algorithm. If the instructor wants to delve deeper into this subject, Chapter 19 can be covered next, providing complete discussions of sequential search, binary search, big-oh notation, insertion sort, and quicksort. Otherwise, the course can go on directly to the other topics in Chapter 14: string arrays, parallel arrays, multidimensional arrays. In my course, I usually cover most of Chapter 14 through examples worked out in class, leaving the complete treatment of searching and sorting for the next course.

Chapter 15: Records

The core issues in this chapter are presented in Sections 15.1–15.4, so shorter courses may want to skip Sections 15.5 and 15.6. If the goal is to cover some data abstraction at the end of the first quarter or semester, don't spend too much time on Chapter 15, but regard it as a preparation for the `Fraction` ADT in Section 16.2.

Chapter 16: Data Abstraction (Optional)

At this point, we enter the part of the book consisting of topics sometimes reserved for a second course or a second semester. Chapter 16 explains the central issues in the design and construction of abstract data types, and Sections 16.1 and 16.2 are prerequisite to material in Sections 18.4 (`Set` ADT), Chapter 20 (`List` ADT and `BinarySearchTree` ADT), and Chapter 21 (`Stack` and `Queue` ADTs).

Chapter 16 emphasizes the distinction between an ADT's interface and its implementation. Although Standard Pascal does not provide the tools to build a proper ADT, most commercial compilers do. Some general comments about this appear in Section 16.2, with detailed instructions for using units in Turbo and THINK Pascal provided in Appendix H. Instructors preferring to skip ADTs can skip Chapters 15, 20, and 21, and cover only the early sections of Chapter 18 and whatever portions of Chapter 17 and 19 they desire.

Chapter 17: Files (Optional)

There is considerable disagreement among instructors over when to introduce text file I/O. Some instructors want students to use text files as early as Chapter 4, while others save it until late in the CS1 course or even postpone it until the second course. In order to meet all needs, I have chosen to collect all file I/O material in Chapter 17, but to organize it in a three-level presentation. Sections 17.1 and 17.2 can be covered immediately after Sections 4.3 and 4.4, respectively, and Sections 17.3–17.6 can be covered right after Section 10.3, if desired.

Chapter 18: Sets (Optional)

Only the longer CS1 courses are likely to cover this chapter in depth, but enough material is provided here for two-semester courses. Note that Section 18.4 requires that Chapter 16 has been covered previously.

Chapter 19: Algorithms for Searching and Sorting (Optional)

This chapter is unusually complete, and provides enough material for both CS1 and CS2 courses; the instructor should choose from what is offered. The treatment of big-oh notation in Section 19.3 provides both a heuristic and a formal discussion to suit courses at different levels.

Chapter 20: Pointers and Dynamic Data Structures (Optional)

Much of this chapter will be saved for a second course, but Sections 20.1–20.3 are often covered in CS1, and some of Section 20.6. A complete coverage is provided to allow choice and to provide material for longer courses. Except for the first section, most of this chapter draws on the ADT concept introduced in Chapter 16.

Chapter 21: Object-Oriented Programming for Stacks and Queues (Optional)

This is traditionally CS2 material, but is provided here for use in longer courses. Again, it uses the ADT concept introduced in Chapter 16, but goes on to discuss inheritance and virtual methods, while showing how stack and queue objects can be derived from a generic sequence type.

Teaching Aids

Computer science is learned with hands on the machine, not while reading a book. Therefore a lab manual is available as a companion to this text. It provides detailed, hands-on instructions for using MS-DOS and Apple Macintosh operating systems and both Borland/Turbo and THINK Pascal systems. There are step-by-step guided lessons in debugging, and a number of guided case studies in problem solving and program design, including

- Dealing with roundoff error.
- Program animation.

- Experiments with integer, character, and boolean data and with parameters and control structures.
- Calendar programs.
- Testing loop invariants.
- Estimating the readability of text.
- Animating recursion.
- Making change.
- Cellular automata.
- Drawing Sierpinski curves (fractals).
- Simulating a Turing machine.
- Simulating a psychiatrist.
- Animating sorting.
- Building linked lists.

An instructor's manual is also available to interested teachers, containing

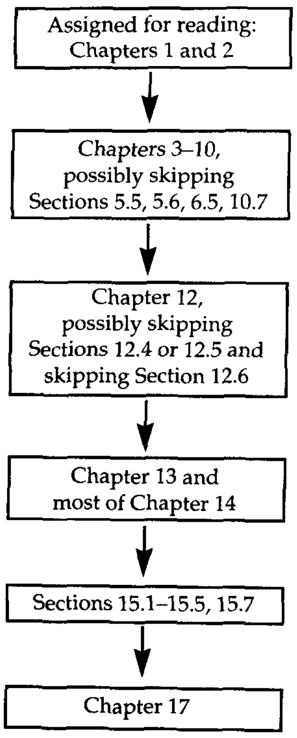
- Tips on teaching the course.
- Sample course outlines.
- Solutions to exercises and projects.
- Sample exam questions.

A bank of test questions is also available separately. The latter two items can be obtained by writing to the publisher on a school letterhead.

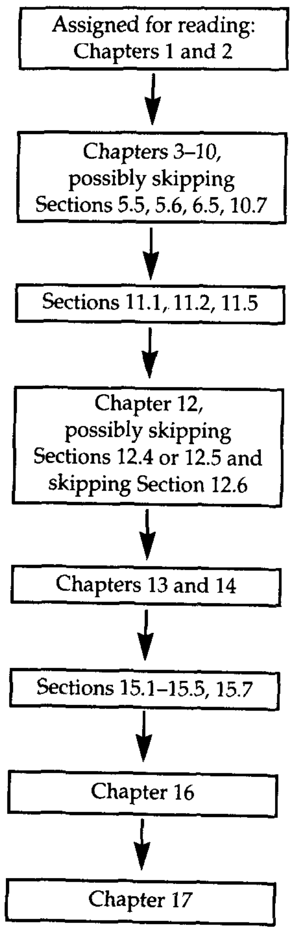
Sequencing of material

The diagrams on the next two pages show how *Structures and Abstractions* can be used in courses as short as one quarter or as long as two semesters.

Short Course


Assigned for reading: Chapters 1 and 2
Chapters 3–10, possibly skipping Sections 5.5, 5.6, 6.5, 10.7
Chapter 12, possibly skipping Sections 12.4 or 12.5 and skipping Section 12.6
Chapter 13 and most of Chapter 14
Sections 15.1–15.5, 15.7
Chapter 17

One Quarter or Semester


Assigned for reading: Chapters 1 and 2
Chapters 3–10, possibly skipping Sections 5.5, 5.6, 6.5, 10.7
Sections 11.1, 11.2, 11.5
Chapter 12, possibly skipping Sections 12.4 or 12.5 and skipping Section 12.6
Chapters 13 and 14
Sections 15.1–15.5, 15.7
Chapter 16
Chapter 17