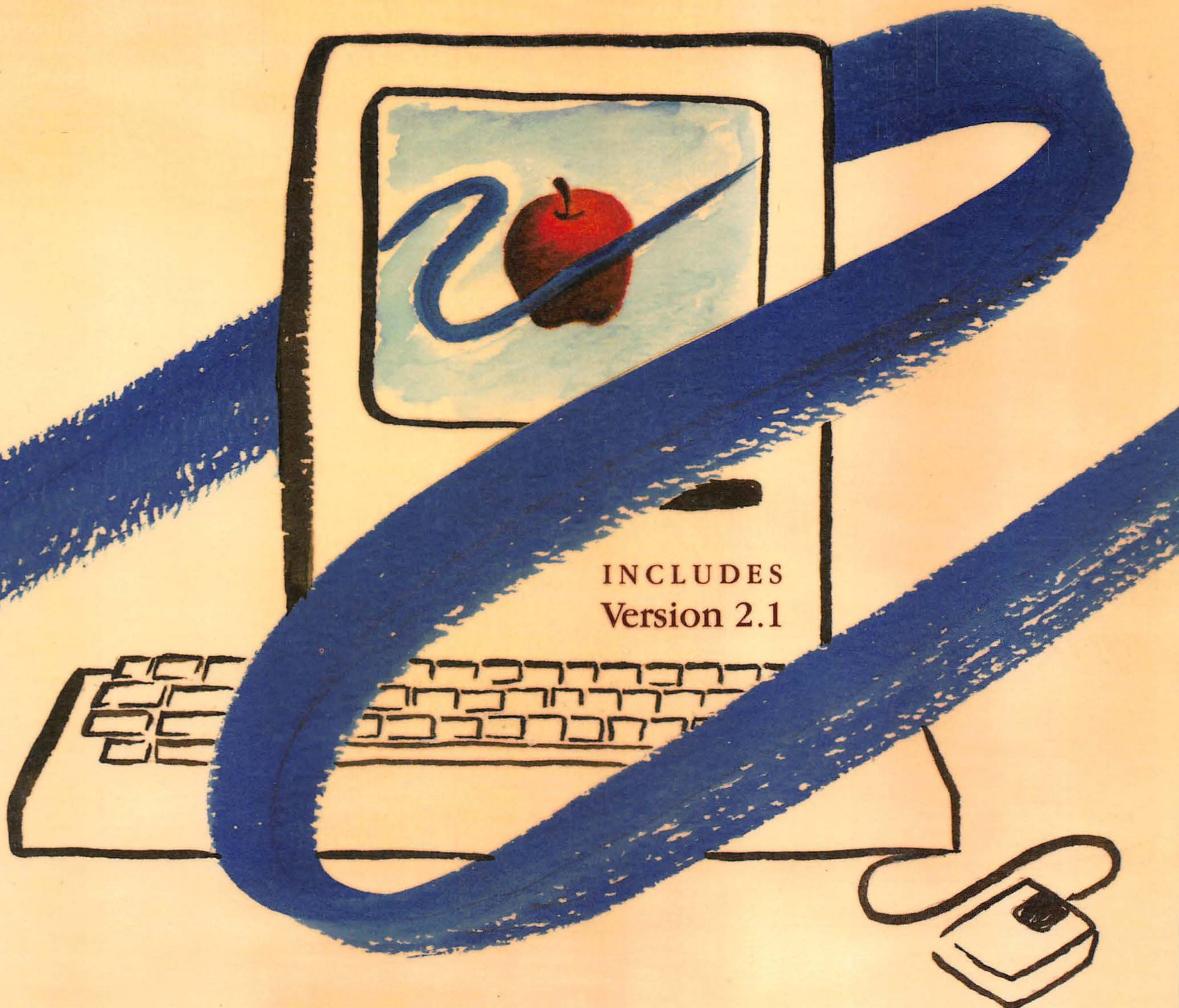
PROGRAMMING MACINTOSH PASCAL



John J. DiElsi Elaine S. Grossman

PROGRAMMING
MACINTOSH PASCAL

PROGRAMMING MACINTOSH PASCAL

John J. DiElsi
Elaine S. Grossman

Macmillan Publishing Company
New York

Collier Macmillan Publishers
London

To our parents
LEAH and ERNEST PARIS
TINA and JOHN DIELSI
for their love, encouragement, and self-sacrifice

Copyright © 1987, Macmillan Publishing Company, a division of Macmillan, Inc.

Printed in the United States of America

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher.

Macmillan Publishing Company
866 Third Avenue, New York, New York 10022

Collier Macmillan Canada, Inc.

Library of Congress Cataloging-in-Publication Data

DiElsi, John J.
Programming Macintosh Pascal.

Includes index.
1. Macintosh (Computer)--Programming. 2. PASCAL (Computer program language) I. Grossman, Elaine S. II. Title.
QA76.8.M3D543 1987 005.265 86-28495
ISBN 0-02-329521-X

Printing: 1 2 3 4 5 6 7 8 Year: 7 8 9 0 1 2 3 4

ISBN 0-02-329521-X

Preface

When you pick up a book, it's always nice to have some notion of what it is about before you open the cover.

The title of this book clearly states the purposes of this text: (1) to present a structured approach to the programming language Pascal, and (2) to show how to implement Pascal on the Macintosh microcomputer.

The Macintosh Pascal language environment is ideal for learning programming. The pull-down menus allow beginning programmers to concentrate on Pascal without being encumbered by sometimes difficult and confusing editing and operating instructions found in many systems. Also included in this unique system is the ability to see a program listing, text output and graphic output on the monitor screen simultaneously. Debugging of programs is simplified by displaying the contents of selected memory locations at specific points in a program.

A new update of Macintosh Pascal called Version 2.1 is now available. This text can be used by those employing either the new Version 2.1 or its predecessor. The differences between these two versions, as they apply to the text, are identified in Appendix H.

Improvements offered in the newer version include the ability to use Macintosh Pascal with a hard disk, the removal of copy protection, the ability to write larger programs, the use of tabs to organize program documentation, the ability to pass procedures and functions as parameters, and the addition of new functions and procedures. Some of these embellishments are not discussed in this text since we felt it best to concentrate on the fundamentals.

We have used our teaching experience to create a text that offers a logical, organized presentation of the rudiments of structured programming.

PART ZERO: THE MACINTOSH ENVIRONMENT

The display capabilities available on the Macintosh distinguish it from most other microcomputers by emphasizing a menu-driven, visual approach to the use of an operating system. Part Zero presents a summary of the instructions needed to create and execute Pascal programs on the Macintosh.

Those who are already familiar with this operating system may choose to skim over or entirely skip Chapter 1. If you have not worked with this system, this chapter provides the commands essential to the use of the operating system.

The second chapter in this part of the book introduces the Macintosh Pascal system. Although the menus provide many options, only those necessary to create, edit and execute a Pascal program are covered. The chapter also contains several Hands-On Exercises that can lead you through processes helpful in using Macintosh Pascal.

Two appendices to Part Zero supplement the material presented in these two chapters. Appendix A gives a listing of the key equivalents that can be used to select menu choices from the keyboard instead of with the mouse. Appendix B contains a step-by-step process that enables you to initialize a disk if you need additional space in which to store programs. Although the Pascal disk can store programs, the available space on it is very limited, and therefore the creation and use of a disk solely for the storage of programs is desirable. The instructions in Appendix B do not produce a "startup disk"—one that can be used to start the system.

We feel it is very important that you become familiar with the system before going on to Pascal, as you can then focus your attention fully on the fundamentals of creating a program.

PART ONE: FIRST STEPS IN PASCAL PROGRAMMING

This part is designed to introduce both a methodology for problem-solving as well as a selection of fundamental Pascal tools with which to work. The Pascal statements presented in this part of the book provide the means for the simple input, output, and processing of data.

In addition to learning the vocabulary, syntax, and grammar of Pascal, the art of constructing well-structured programs must also be developed. Most programmers find this art difficult to learn. For this reason, the text provides a number of features to guide you. Chapter 3 is entirely devoted to describing a six-step methodology that you can use for problem solving. The steps are further enhanced by the use of the input-processing-output (IPO) charts and Nassi-Shneiderman (N-S) charts. Since the process is essentially independent of a programming language, no knowledge of Pascal is required at this stage. Although this methodology may seem tedious, it will help you to design and create the solutions to complex problems. Uppercase letters are used to represent input, processing, and output quantities.

In addition to introducing the basic components of any Pascal program, Chapter 4 discusses generic data types. If you only know BASIC, the concept of a data type and its declaration is relatively new and must be clearly understood at an early stage. Although many data types are presented, the problems in the text primarily use real, integer, string, and Boolean. Chapter 12 ("Modular Programming") could be inserted here, however, we feel that you will get a better understanding of the concepts presented in Chapter 12 after you've had some experience in the creation of complex programs.

Chapter 5 discusses how data can be sent from main memory to the outside world, and how data can be entered from the outside world into main memory. Once data resides in main memory, they can be output to the **Text** window of the Pascal screen, the **Drawing** window of the screen, a data file stored on a disk, or a printer. These are four different ways to display the same results. This chapter also shows how data can be entered into main memory from the keyboard or an existing data file. You can choose some of these methods, or all of them. A more extensive discussion of files is given in Chapter 13 ("Advanced Data File Techniques").

The last chapter of Part One shows how calculations on data are performed in order to achieve desired results. To emphasize the concept of a labeled memory location, a Hands-On Exercise on the use of the **Observe** window is included. The techniques demonstrated by this exercise should prove useful for finding errors in larger programs.

PART TWO: PROGRAMMING STRUCTURES

Part Two provides the next steps in developing your programming skills by employing the theory of structured programming and introducing sequential, selection, and repetition programming constructs. It presents material that allows you to solve problems that involve the computer making decisions and repeating processes. This part discusses built-in precoded routines and shows how large quantities of data can be handled with an economy of program code.

Many useful built-in functions and procedures are presented in Chapter 7. You may be selective in your reading by choosing those which are of interest or benefit to you. You may find the graphic capabilities of the Macintosh particularly appealing.

Chapter 8 shows you how to create and develop your own functions and procedures. The ability to break up a large program into smaller components, functions and procedures, is the basis of modular design. The functions and procedures permit the passing of values between parts of a program and may cause some difficulty, but the understanding of this process and the ability to do it is important.

Chapter 9 presents the second of the three basic programming structures: the decision structure. Until this point, all program statements were executed in sequential order. The if. . .then. . .else and case statements in this chapter permit programs to select one of many alternative paths of execution.

The last of the three basic structures (looping) is presented in Chapter 10. The **for**, **while**, and **repeat** statements in the chapter allow a computer to repeatedly perform the same operations on different sets of data. The techniques of summing and counting using loop structures are also demonstrated.

Chapter 11 introduces arrays and records, which allow your program to handle and store large amounts of data with a minimum of coding. An understanding of these fundamental data structures provides a good basis for implementing files (Chapter 13) and more sophisticated data structures. Elementary algorithms for common operations performed on these structures (searching, sorting, and merging) are covered in this chapter.

PART THREE: ADVANCED PROGRAMMING CONCEPTS

The programming skills and techniques presented in the previous parts of this book are now applied to the solution of substantial problems. This part presents the theory behind the construction of solutions, their implementation using files, and the introduction of additional data types and structures.

Chapter 12 ("Modular Programming") presents the theory behind principles of structured programming. Although these principles were followed throughout the text, their formal development at this point can be appreciated since you now have a greater familiarity with programming. The contruction of user-friendly programs that display programming menus offers practical real-world applications of these theoretical ideas.

Chapter 13 ("Advanced Data File Techniques") carries the organization of array and record data structures into files. One application from Chapter 12 that was solved with an array of records is also solved in this chapter with a file of records. The creation, searching, and updating of files are detailed.

The more sophisticated Pascal structures of enumerated data types, pointer data types, and recursive subprograms are introduced in Chapter 14 ("Advanced Pascal Structures"). Simple examples are presented and discussed. These topics provide a good introduction to an elementary course in data structures.

It is beyond the scope of this book to present exhaustive coverage of each feature of Macintosh Pascal. We selected those topics which we deemed of greatest importance and direct you, for further information, to the manual that accompanies your version of Pascal.

GENERAL FEATURES

Where applicable, each chapter concludes with a section on typical programming errors, including a description of the errors and examples. These sections highlight the errors commonly made by novice programmers.

Each chapter ends with Nonprogramming and/or Programming Exercises. The former are designed to test a mastery of programming concepts; and the latter, a mastery of programming techniques. Answers to selected Nonprogram-

ming and Programming Exercises are in Appendix G. Answers to the remaining Nonprogramming Exercises and additional Programming Exercises can be found in the *Instructor's Resource Manual.* Each programming exercise is preceded by the code (B) or (G) to indicate that the problem is of a business or general nature.

In addition to the appendix of selected answers, six other appendices provide useful summaries for quick reference of important facts.

Both Examples and Problems are contained in the text. Examples consist of program segments that appear in a two-column format. The Pascal code is in the left column and its corresponding explanation in the right column, making it easier to "see" the code and its explanation. Problems differ from Examples in that they consist of problem statements followed by the complete process required to produce the Pascal solution.

For further clarity, keywords or phrases appear in darker print. This helps you to distinguish between Pascal code and the text used to interpret this code.

To avoid duplicating portions of programs to show output on the printer as well as the screen, program solutions throughout the text are coded to produce output to the screen only. It is not difficult to make minor changes in these programs to send output to the printer.

COURSE OF STUDY

This text was written for use in an introductory college computer course that uses a structured high-level language. It is also appropriate for a high school course, especially as part of an advanced placement program in computer science, or it can serve as a self-instruction guide for a computer enthusiast who wishes to learn Pascal in a Macintosh environment.

In any case, Part Zero may be omitted if you are already familiar with the programming environment. Chapters 3 through 6 should be covered in the order presented. The theory presented in Chapter 12 may be introduced in Chapter 4 if so desired. The procedures and functions presented in Chapter 7 can be selectively covered. However, you should be aware that the examples in the remainder of the text may use some of the functions and procedures of Chapter 7.

To emphasize the structure of Pascal, programmer-defined functions and procedures (Chapter 8) are introduced before selection (Chapter 9) and iteration (Chapter 10). This order also allows programmer-defined subprograms to be included in examples and problems in the chapters that follow.

Files (Chapter 13) should be included in an introductory course, while selected topics in Chapter 14 may be covered if time permits.

Acknowledgments

We wish to express our thanks to the students we have taught over the years because teaching is a two-way learning experience, and much that is contained in the pages of this text is the product of what we have learned from those students. We would also like to thank Paul Hoffman, Corey Schou, and David Marotta for their time and helpful suggestions. Special thanks are due to our colleagues at Mercy College, Westchester County, New York, for their encouragement, and to Philip Cecchettini, Barbara Pickard, Larry Lazopoulos, Linda Purrington, and Bernie Scheier for their guidance and support. A great measure of our success is due to the support and patience of those closest to us: our family and friends. It is they who have had to put up with the late hours and our uneven temperaments. They deserve our sincere gratitude and thanks.

John J. DiElsi
Elaine S. Grossman
December 1986

Contents

PART ZERO:
THE MACINTOSH ENVIRONMENT 1

CHAPTER 1:
The Macintosh Operating System 3

1.1 Introduction 3
1.2 The Macintosh System Screen 6
1.3 The Operating System **File** Menu 7
Open 8
Close 9
Eject 9
1.4 The **View** Menu 9
1.5 The **Special** Menu 11
1.6 The **Trash** Can 11
Exercises 13

CHAPTER 2:
The Macintosh Pascal System 14

2.1 Introduction 14
2.2 The Pascal **File** Menu 15
New 15
Open... 15
Close 16
Save 17
Save As... 17
Print... 18
Quit 18

2.3 **Edit** Options: HANDS-ON EXERCISE 19
Cut 20
Paste 21
Copy 21
Clear 22
Select All 22

2.4 Editing Alternatives: HANDS-ON EXERCISE 23
2.5 **Search** Options: HANDS-ON EXERCISE 25
What to find . . . 26
Everywhere 26
Find 27
Replace 28

2.6 The **Run** and **Pause** Menus 28
Go 28
Pause 28
Halt 29
Reset 29

2.7 The **Windows** Menu 29
Untitled 30
Text 30
Drawing 31
Clipboard 31
Type Size . . . 31

2.8 A Programming Session: HANDS-ON EXERCISE 32
Exercises 34

APPENDIX A:
MENU KEY EQUIVALENTS 35

APPENDIX B:
USING SUPPLEMENTARY DISKS 36

PART ONE:
FIRST STEPS IN PASCAL PROGRAMMING 41

CHAPTER 3:
Design and Construction of Problem Solutions 43

3.1 Introduction 43
3.2 Problem-Solving Methodology 44
Step 1: Analyzing a Problem 44

Step 2: Creating an Input-Processing-Output (IPO) Chart 44
Step 3: Looking for Clues 46
Step 4: Planning a Solution Using a Nassi-Shneiderman (N-S) Chart 46
Step 5: Coding the Solution 46
Step 6: Testing and Debugging the Solution 47
3.3 Practical Applications 47
Exercises 54

CHAPTER 4:
The Components of a Pascal Program **56**
4.1 Introduction 56
4.2 Generic Data Types 56
Integer 57
Long Integer 57
Real 57
Double 57
Extended 58
Character 58
String 58
Boolean 58
4.3 Declaration Statements 58
Constant Declarations 59
Variable Declarations 60
Type Declarations 61
4.4 Essential Program Components 63
Program Names 63
Begin and End 63
Comments 64
Putting the Parts Together 64
4.5 Typical Programming Errors 65
Exercises 66

CHAPTER 5:
A First Step: Input and Output **68**
5.1 Introduction 68
5.2 Program Output 68
Text Screen Output 69
Drawing Screen Output 74
Data File Output 76
Printer Output 78
5.3 Program Input 78
Keyboard Input 79
Data File Input 81

5.4 Sample Programs 82
5.5 Typical Programming Errors 95
Nonprogramming Exercises 95
Programming Exercises 97

CHAPTER 6:
Assignment Statements and Programming Aids **99**

6.1 Introduction 99
6.2 Performing Calculations 99
Numeric Operators 99
Numeric Expressions 100
Assignment Statements 102
6.3 Sample Programs 104
6.4 Programming Utilities 113
*The **Observe** Window 113*
Walking Through a Program: HANDS-ON EXERCISE 114
Step 115
Step-Step 116
Stop 116
Go-Go 118
6.5 Typical Programming Errors 119
Nonprogramming Exercises 119
Programming Exercises 121

PART TWO:
PROGRAMMING STRUCTURES **125**

CHAPTER 7:
System-Defined Functions and Procedures **127**

7.1 Introduction 127
7.2 Numeric Functions 128
*The **abs** Function 129*
*The **sqr** Function 129*
*The **sqrt** Function 130*
*The **odd** Function 130*
*The **round** Function 131*
*The **trunc** Function 131*
*The **random** Function 131*
7.3 String Functions and Procedures 137
*The **length** Function 137*
*The **pos** Function 138*
*The **concat** Function 138*

The copy Function 139
*The **delete** Procedure* 140
*The **insert** Procedure* 140
7.4 Graphic Procedures 145
*The **moveto** Procedure* 146
*The **lineto** Procedure* 147
*The **drawline** Procedure* 147
*The **framerect** Procedure* 148
*The **paintrect** Procedure* 149
*The **invertrect** Procedure* 150
*The **paintcircle** Procedure* 151
*The **invertcircle** Procedure* 151
7.5 Typical Programming Errors 161
Nonprogramming Exercises 161
Programming Exercises 162

CHAPTER 8:
Programmer-Defined Functions and Procedures **165**
8.1 Introduction 165
8.2 Functions 166
8.3 Procedures 168
8.4 The Structure of a Program Containing Subprograms 172
The Placement of Functions and Procedures 172
The Calling of Functions and Procedures 173
The Scope of Identifiers 174
8.5 Problem Solutions Using Functions and Procedures 178
8.6 Typical Programming Errors 196
Nonprogramming Exercises 197
Programming Exercises 199

CHAPTER 9:
Choices **202**
9.1 Introduction 202
9.2 Altering the Flow of Control 202
9.3 Conditional Expressions 203
Relational Operators 203
Logical Operators 204
Order of Operations 205
9.4 The **if** . . . **then** . . . **else** Structure 206
The Single-Option Structure 206
The Double-Option Structure 208
Nested Selection Structures 213
9.5 The **case** Structure 215

9.6 Typical Programming Errors 225
Nonprogramming Exercises 225
Programming Exercises 228

CHAPTER 10:
Loops **232**

10.1 Introduction 232
10.2 Looping Structures 232
*The **for** . . . **do** Structure 233*
*The **while** . . . **do** Structure 236*
*The **repeat** . . . **until** Structure 238*
10.3 Summing and Counting Statements 245
10.4 Nested Looping Structures 252
10.5 Typical Programming Errors 263
Nonprogramming Exercises 263
Programming Exercises 266

CHAPTER 11:
Fundamental Data Structures **268**

11.1 Introduction 268
11.2 The Array Data Type 268
11.3 The Record Data Type 282
11.4 Operations on Data Structures 291
Searching 291
Sorting 291
Merging 293
11.5 Typical Programming Errors 300
Nonprogramming Exercises 301
Programming Exercises 303

PART THREE:
ADVANCED PROGRAMMING CONCEPTS **307**

CHAPTER 12:
Modular Programming **309**

12.1 Introduction 309
12.2 The Theory of Modular Program Design 310
Top-Down Design 311
Structured Programming 312
Documentation 313

12.3 User-Friendly Programs 313
Creation of a Program Menu 314
Subprogram Dialogue 315
12.4 Practical Applications 315
Nonprogramming Exercises 360
Programming Exercises 360

CHAPTER 13:
Advanced Data File Techniques 363

13.1 Introduction 363
13.2 Creating a File of Records 364
*The **open** File Statement 365*
13.3 Writing to a File of Records 365
*The **write** File Statement 366*
*The **filepos** Function 366*
*The **close** File Statement 367*
13.4 Reading from a File of Records 367
*The **read** File Statement 367*
*The **eof** Function 368*
13.5 Modifying a File of Records 369
Adding a Record 369
*The **reset** File Statement 369*
Changing a Record 369
*The **seek** File Statement 370*
13.6 A Practical Application 371
13.7 Typical Programming Errors 377
Nonprogramming Exercises 377
Programming Exercises 379

CHAPTER 14:
Advanced Pascal Structures 382

14.1 Introduction 382
14.2 Enumerated Data Types 382
14.3 Pointer Data Types 387
*The **new** Procedure 388*
*The **dispose** Procedure 390*
Linked Lists 390
14.4 Recursion 395
The Structure of a Recursive Definition 395
The Implementation of a Recursive Definition 396
14.5 Typical Programming Errors 399
Nonprogramming Exercises 399
Programming Exercises 402

APPENDIX C:
ASCII CHARACTER CODES 406

APPENDIX D:
RESERVED WORDS 408

APPENDIX E:
MACINTOSH PASCAL INSTRUCTION SUMMARY 409

APPENDIX F:
MACINTOSH PASCAL FUNCTION AND PROCEDURE SUMMARY 411

APPENDIX G:
ANSWERS TO SELECTED EXERCISES 415

APPENDIX H:
MACINTOSH PASCAL VERSION 2.1 464

INDEX 473

PART ZERO

THE MACINTOSH ENVIRONMENT

C H A P T E R 1

The Macintosh Operating System

1.1 INTRODUCTION

Every computer system is composed of hardware and software. The term *hardware* refers to the physical components of a system, while *software* refers to the instructions written to direct the hardware to perform specific tasks. These instructions are "soft" in the sense that they are a collection of ideas, something created in the mind. Although the Macintosh system's hardware is similar to the hardware found in other computer systems, its software is quite unique. The software that accompanies the Macintosh microcomputer has been designed to make the hardware as easy as possible to use.

The fundamental hardware components of any computer system include the central processing unit (CPU), main memory, and input/output (I/O) devices. See Figure 1.1.

The CPU consists of a control unit and an arithmetic/logic unit (ALU). The control unit acts as a coordinator of computer operations. Instructions and data are routed through this unit to the proper destination. The ALU is the portion of the CPU that actually performs calculations and makes decisions.

Main memory stores the data and instructions that are to be processed by the computer. The computer can only interpret instructions and data that are contained in main memory.

Input devices are used to enter data and instructions into main memory. Output devices accept information from main memory for the purpose of displaying or storing that information in another part of the computer system.

Figure 1.2 depicts a typical Macintosh system. The CPU and main memory are both housed in the unit containing the display screen. Although some computers have color monitors, the Mac's is black and white. The monitor is an output device, because it receives information from main memory and displays it.

One novel feature of the Macintosh system is the mouse. The mouse is a small device attached to the Mac by a cord. It can be maneuvered on a flat surface next to the computer. As you move this mouse around the tabletop, a corre-

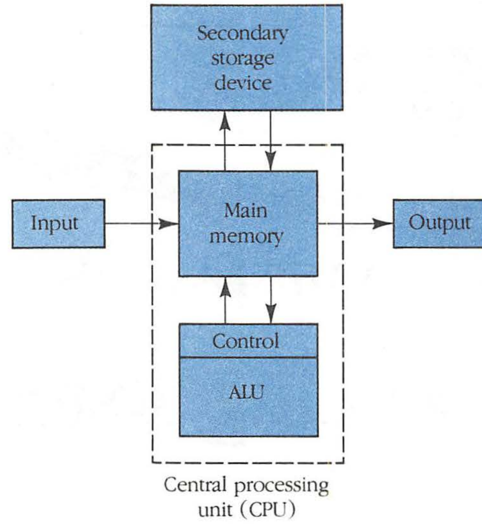


FIGURE 1.1

FIGURE 1.2

sponding arrow (cursor) moves on a monitor or display screen. When the arrow points to an activity, you simply click or depress the button on the mouse. Because the mouse can be used to select activity options for the Macintosh system, it is regarded as an input device.

Another input device is the keyboard, through which you can enter information into main memory. Like the mouse, the keyboard is connected to the monitor unit by a cord that allows for flexible positioning.

While the Macintosh has a main memory, this memory is not large enough to store all the information that you may wish to keep. Therefore, computer systems include auxiliary or secondary storage devices. These devices store information in a form that is easily accessible to the computer's main memory. One such device is a disk drive. Disks used to store information are inserted into the disk drive located on the monitor unit. Data elements stored on these disks can then be transferred into main memory so that the computer can interpret and process them. Since the computer can transfer data to the disk from main memory and read data from the disk into main memory, the disk drive serves as both an input and an output device.

Finally, a printer is connected to the monitor unit. This output device is similar to a typewriter and produces printed copies of information stored in main memory. One printer commonly used with the Macintosh system is called the Imagewriter.

Throughout this chapter, we assume that you have access to a system composed of the Macintosh computer, a mouse, a keyboard, an Imagewriter printer, and a Macintosh Pascal disk. Modifications in this setup may require slightly different operating instructions.

Recall that software is created by the human mind. It is organized as sets of instructions that direct the computer to perform particular functions. An operating system is the most important software in a computer system. It serves as an intermediary between a user and a computer system. After receiving an instruction, it activates and controls the hardware necessary to carry out that instruction. The Macintosh has an operating system that makes it very easy for an inexperienced user to communicate with the computer.

A set of instructions to perform a particular task is called a *program*. Some programs are written for users who do not know how, or who cannot spend the time, to create their own programs. Examples of such programs or packages include MacWrite, a word processing package, and MacPaint, a graphics package. Although commercial software is available for some applications on some systems, software packages do not exist to solve every problem for every system. The purpose of this text is to provide an understanding of the fundamentals of programming using Macintosh Pascal so that you can create your own original programs or packages.

Many computers offer Pascal as a programming language, but Macintosh Pascal is unique and, in many ways, easier to use. The mouse innovation makes the selection of various activities fairly easy. To take full advantage of the system, you first must become familiar with the elements of the Macintosh operating system. An *operating system* is a software package that is used to control all the

activities of a computer. Although quite a few activities are available, only those necessary for the production of elementary programs are introduced initially. You might want to consult the reference manuals for a complete discussion of the Macintosh or Macintosh Pascal features. Don't be afraid to experiment with the mouse and the various options on the system as the sections of this chapter are presented. After a brief exposure, you should become comfortable using the mouse, and before long the letters MAC should come to mean *Move And Click.*

1.2 THE MACINTOSH SYSTEM SCREEN

When the Macintosh is turned on and the Macintosh Pascal disk is inserted in the disk drive, a *Welcome to Macintosh* greeting appears on the screen. This is followed by a list of words across the top of the screen. These words form what is known as a *menu header.* A menu header displays the options available for operating the Macintosh system (see Figure 1.3).

In addition to the menu header, two symbols appear on the upper right and lower right portions of the system screen. These symbols are called *icons* and are an integral part of the Macintosh system. An icon, in general, is a picture that represents an object. The icon in the upper right corner of the screen is a drawing of a disk labeled **Pascal** and represents the portion of the software system involved in the creation and use of programs written in the programming

FIGURE 1.3

language Pascal. The icon in the lower right corner of the screen is a drawing of a garbage can labeled **Trash**. Its function is to remove information from a disk.

A brief description of some of the more important menu choices follows. It is by no means complete but serves as an introduction for the novice to this system.

1.3 THE OPERATING SYSTEM **FILE** MENU

One of the entries in the menu header is **File**. The **File** option has its own menu, which provides commands for handling the information that is stored on disks. To see the **File** menu, move the mouse until the arrow on the screen points to the word **File** and hold the mouse button down. If the button is released, the menu disappears. **File** commands appear under the word **File**, which is now displayed in inverse mode, that is, with white letters on a black background (see Figure 1.4).

Some options may appear in dark print, and others appear dimmed. The dark print indicates the ones available at this particular time, while the dimmed print indicates options not currently available. As the system is used, the options that are available at any one time change.

To select a particular option using the mouse, move the mouse so that the arrow on the screen moves down the list of displayed choices. As the arrow

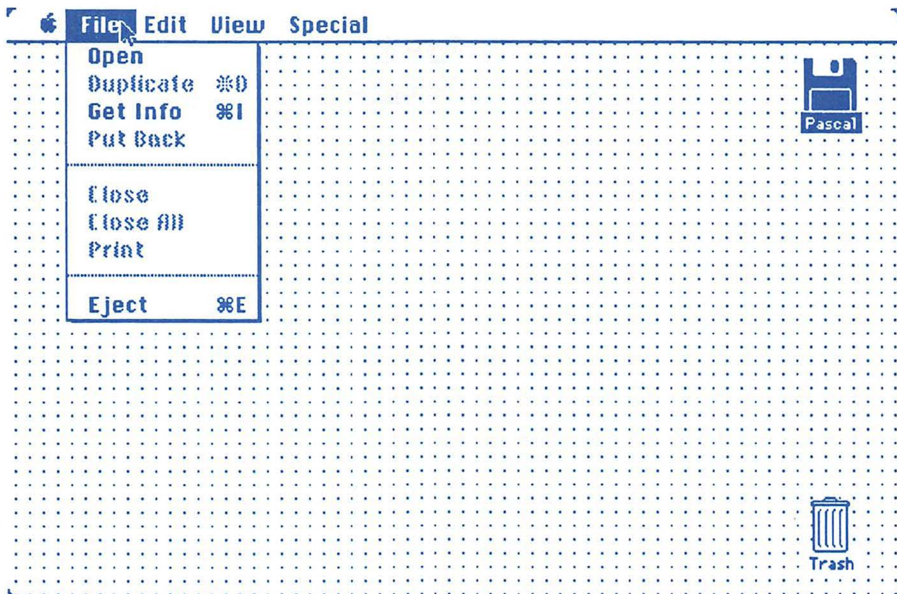


FIGURE 1.4

moves over each choice, the choice inverses. If you would like to select the inversed choice, release the button on the mouse.

The mouse may be the easiest way to select an option, but some commands may be chosen using alternate methods. For example, the **Eject** option can be selected by either using the mouse or holding down the command (⌘) key and then depressing the E key on the keyboard. Menu key equivalents are listed in Appendix A, on p. 35.

OPEN

The **Open** option in the **File** menu is used to activate files. A file is a collection of related information. A program is an example of a file. If this option is selected, a rectangular area called a *window* entitled **Pascal** appears on the screen. This **Pascal** window displays the contents of the Macintosh Pascal disk used to start the system. See Figure 1.5.

The Macintosh operating system allows you to move, erase, and change the size of windows. This capability permits you to view multiple windows at the same time. To move a window, use the mouse to place the screen arrow on the window's name, hold the mouse button down, and drag the window to any location on the screen. As this is done, the outline of the window moves until the button on the mouse is released. At that point, the entire window appears in a new location on the screen.

A window can be made larger or smaller by placing the screen arrow on the overlapping boxes in the lower right corner of the window. While holding

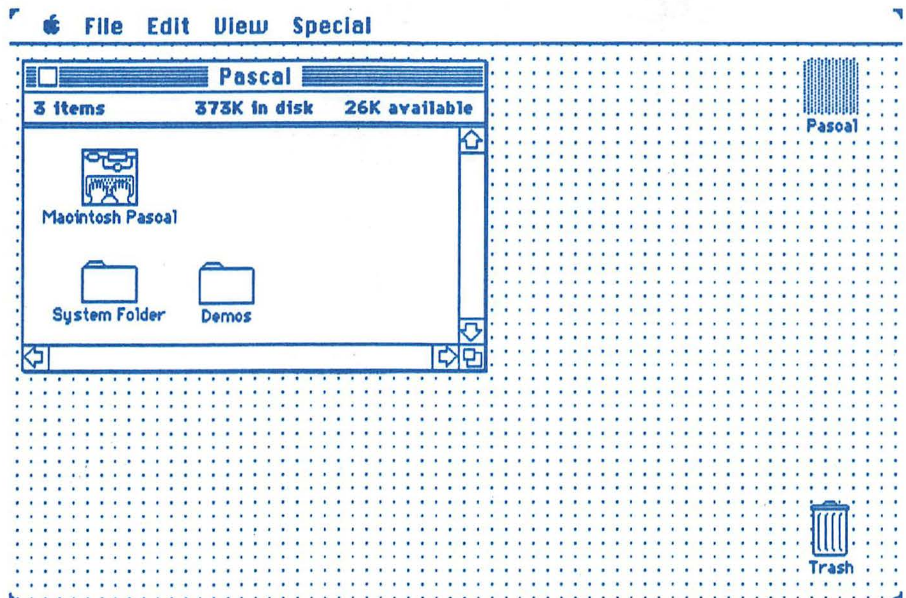


FIGURE 1.5

down the button, move the mouse to shrink or expand the window until the outline is the size you want, and release the mouse button.

Outlines of arrows appear along the right side and bottom of each window. These arrows are used to move or scroll the contents of the window when the contents are too large to be shown in the window. By clicking the mouse button on these arrows, you can move the contents in a window up or down. You can now view all the icons in that window, even though they may not be visible at the same time.

The icons appearing in the **Pascal** window represent files. Each file can be activated by clicking the mouse while it is positioned on the icon signifying that file and then choosing **Open** from the **File** menu.

Instead of using the **Open** option, you can double-click the mouse (click the mouse button twice in quick succession) while it is positioned on any icon. A new window appears on the screen as a result of activating one of these icons.

CLOSE

The **Close** option in the **File** menu deactivates the active opened file. Choosing this option removes the window representing that file from the screen. A file may also be closed by clicking the mouse in the square box in the upper left portion of a window.

EJECT

The **Eject** option in the **File** menu allows you to remove a disk from the disk drive. You should not try to force the removal of any disk from the drive, and always select this option. Once the disk is ejected, you can easily remove it from the drive and insert a new disk or turn the system off.

1.4 THE VIEW MENU

The **View** menu permits you to see, in different ways, information about the files stored on the disk. Figure 1.6 shows the **View** menu options and the corresponding **Pascal** window.

The **View** option in effect is **by Icon.** If you drag the arrow down the menu listing to the **by Name** option, and release the button on that choice, the file listing then appears in a different format. See Figure 1.7.

Now the files are displayed in alphabetical order with size, kind of file, and date of last modification of the file indicated.

The size of the file is measured in units of K. One K represents 1,024 keyboard characters. The Demos file in Figure 1.7 contains approximately 37,000 characters. This view of the files on a disk is useful when trying to determine how much room is left on the disk. A disk has approximately 400 K available for storage.

The remaining options in the **View** menu merely list the files according to different criteria.

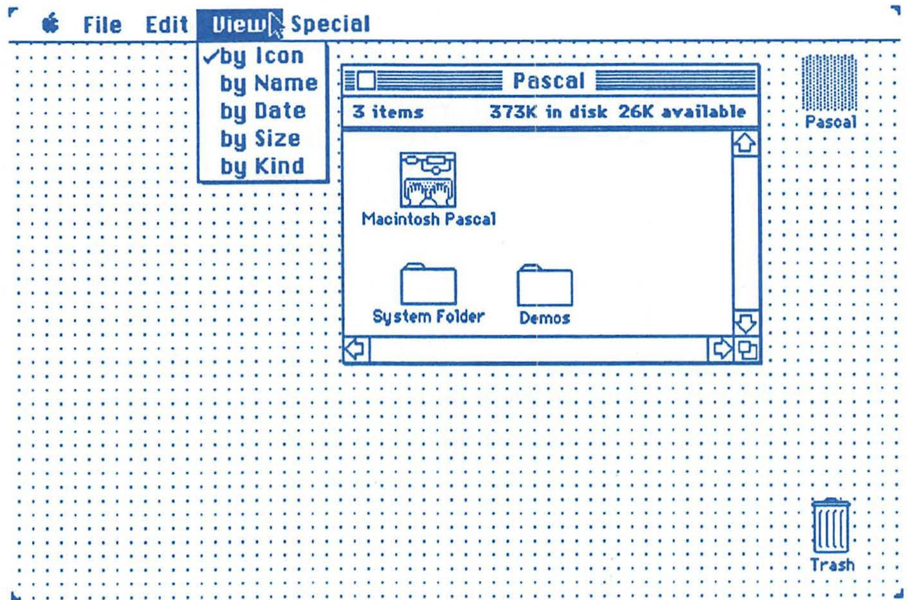


FIGURE 1.6

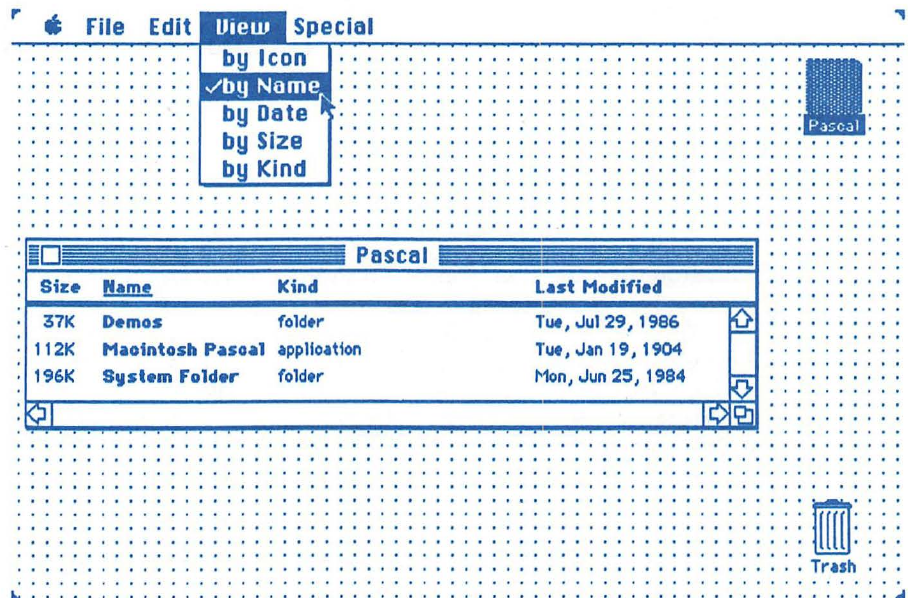


FIGURE 1.7

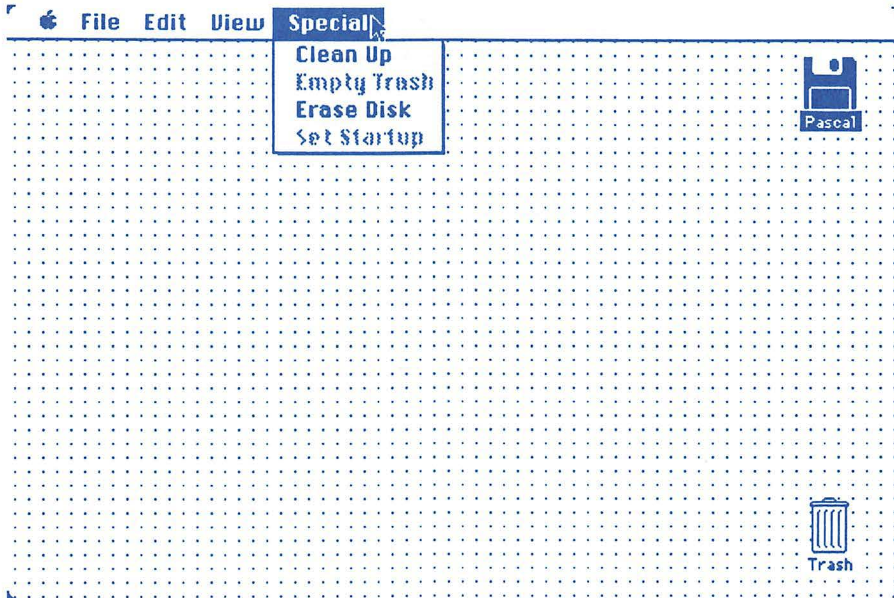


FIGURE 1.8

1.5 THE **SPECIAL** MENU

The last entry in the menu header is **Special**. Figure 1.8 displays the **Special** menu options.

If **Clean Up** is selected, the icons in the active window are arranged in orderly rows and columns. This capability proves useful if the window contains overlapping icons.

1.6 THE **TRASH** CAN

The **Trash** icon in the lower right portion of the screen is used for removing files from the screen. Suppose the disk currently contains a file called **Sample** as indicated by Figure 1.9.

To delete the file **Sample** from the disk, drag the **Sample** icon to the **Trash** icon and release the button on the mouse when the arrow is on the **Trash** icon. The **Sample** icon disappears from the screen. However, all is not lost at this point. **Trash** is also a file. That is, the **Trash** file may be displayed by double-clicking the mouse button while on the **Trash** icon. A **Trash** window now appears on the screen showing its contents. See Figure 1.10.

In the event of an error, the file **Sample** can still be saved from destruction by simply dragging it back into the Pascal window. If the file is to be deleted, select the **Empty Trash** option from the **Special** menu. Activating this option completely erases any files that are shown in the **Trash** window.

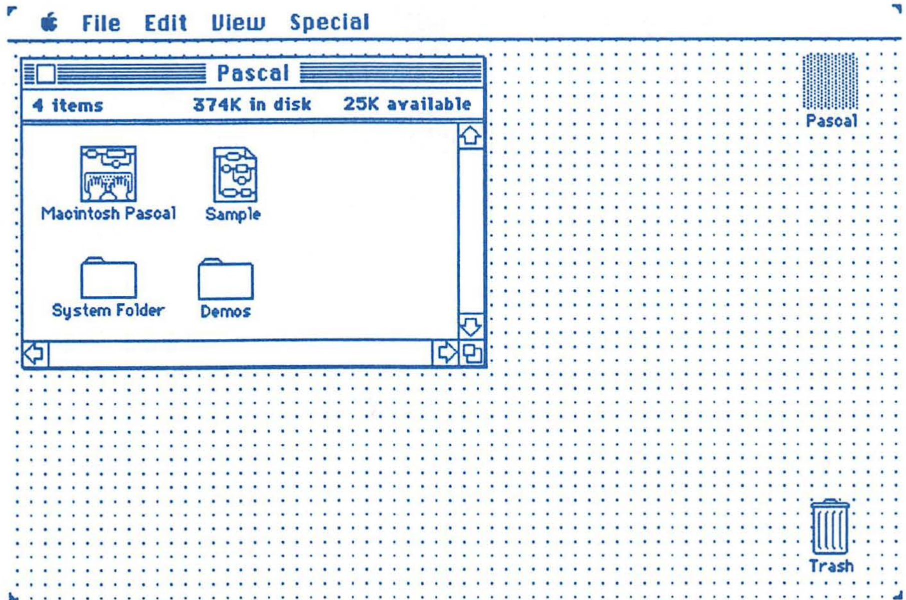


FIGURE 1.9

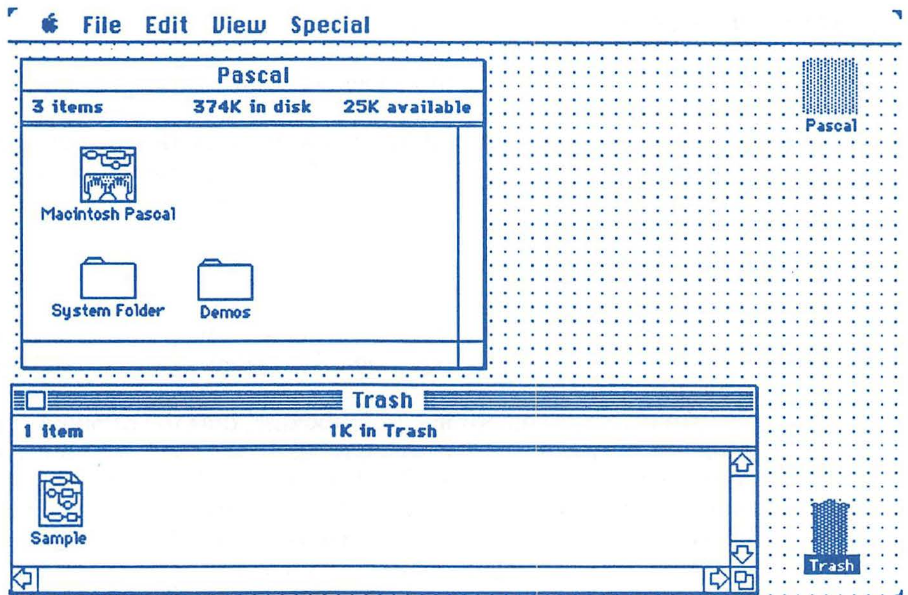


FIGURE 1.10

EXERCISES

Indicate the step or steps a user should take to accomplish each of the following tasks on the Macintosh computer:

1. View the files stored on a disk **by Size.**
2. Display the **Trash** file.
3. View the options contained in the **File** menu.
4. Remove a disk from the disk drive.
5. View the files stored on the disk **by Icon.**
6. Move a window.
7. Change the size of a window.
8. Delete a file from a disk.
9. Select an option from the **File** menu.
10. Close a window.

CHAPTER 2

The Macintosh Pascal System

2.1 INTRODUCTION

The Macintosh system screen gives you the means through which you can enter the Pascal programming environment. To gain access, open the file called **Macintosh Pascal** by double-clicking the mouse while the screen arrow is on that icon in the **Pascal** window, or by clicking once on that icon and using the **Open** option from the **File** menu. The screen that appears, illustrated in Figure 2.1,

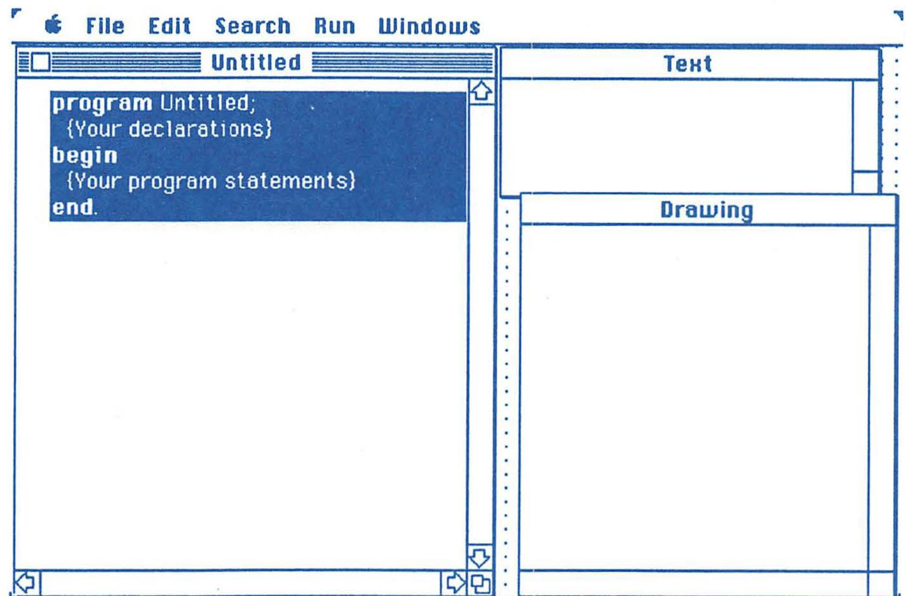


FIGURE 2.1

provides the environment necessary to create, modify, and execute Pascal programs.

This screen contains a menu header and three windows. The **Untitled** window on the left side of the screen is for displaying a list of instructions for the computer to follow and is therefore called the *program window*. The general format for a Pascal program appears in inverse print in this window. The **Text** window in the upper right corner of the screen displays the results of a program in text form, and the **Drawing** window, in the lower right corner, displays program results that require graphic form. Program execution may produce output in either or both windows.

The menu header at the top of this screen differs from the one at the top of the Macintosh system screen. The key words in this menu include **File**, **Edit**, **Search**, **Run**, and **Windows**.

A brief description of some of the more important choices follows.

2.2 THE PASCAL FILE MENU

The Macintosh Pascal **File** menu lets you create, open, close, print, and manipulate Macintosh Pascal files. The options in the **File** menu are displayed in Figure 2.2.

NEW

The **New** option creates a program window called **Untitled** each time that option is selected. A previously opened file must be closed before **New** can create a window for a new program.

A cursor, which in the program window is represented by a straight vertical line flashing on and off, indicates the place where entered data appear. This cursor may be moved to any location in the program window by clicking the mouse when the symbol I is at the desired location. The screen arrow changes to a I symbol when the arrow enters the program window.

OPEN...

The next available option in the **File** menu is **Open...**. This selection causes a dialog box similar to the one in Figure 2.3 to be displayed on the screen.

FIGURE 2.2

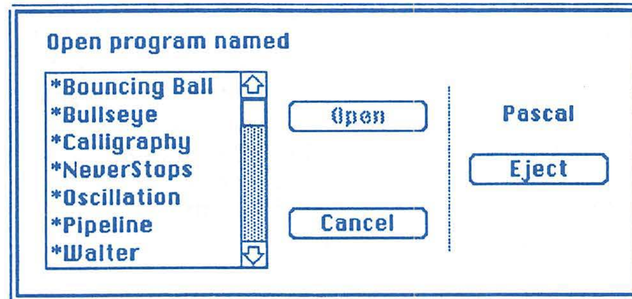


FIGURE 2.3

Figure 2.3 shows an alphabetical listing of the Pascal programs stored on a disk. The mouse can be employed to scroll the listing upward or downward. At this time, **Open** is dimmed and cannot be selected until a choice of a program file has been made.

The **Open** selection allows you access to the contents of a program file. Each opening of a program file causes the instructions in that file to be loaded into main memory and displayed in a new program window. The name of the opened program file appears at the top of the window displaying the contents of that file.

In order to choose a particular file, move the arrow so that it points to the appropriate name and click the button on the mouse. The selected file inverses, and **Open** becomes dark. Move the arrow to **Open** and click once. The dialog box entitled **Open program named** disappears, and a program window is created. An alternate method for opening a program file is to click the mouse button twice in quick succession while the arrow is on the program name. In either case, the program listing is displayed in a window headed by the name of the file. This is the same procedure followed for opening a file from the operating system screen.

If you want to return to the **Pascal** screen without opening or loading a file, click **Cancel** instead of **Open**. If, however, the program to be loaded is stored on another disk, click the **Eject** oval, insert that disk, and open the desired file displayed in the directory of that disk. Specific details on using a secondary disk can be found in Appendix B, which appears on page 36.

CLOSE

The next option available on the **File** menu is the **Close** option. This option puts an active or opened file away and removes that file's program window from the screen. Once this option is selected, no additional work can be done on the file unless the file is reopened.

SAVE

When the option **Save** is chosen, a copy of the program in the program window is saved on a disk, using the file name appearing at the top of the window. If the text is untitled at this time, **Save** is not available and **Save As...** must be chosen.

SAVE AS...

The **Save As...** option saves an untitled program or a titled program under a different name. The latter can be used for making a second copy of an important program. Activating this option displays a dialog box allowing space for the program name. This box may be blank or contain the name of a titled program. See Figure 2.4.

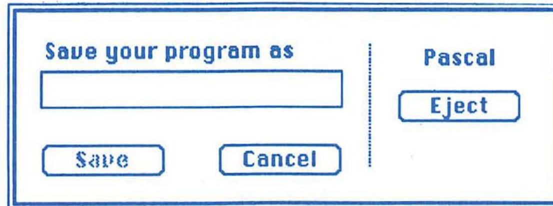


FIGURE 2.4

To clear the box containing the name of a titled program displayed in inverse mode, press the Backspace key on the upper right of the keyboard and enter the new name. Once the name has been entered, you may choose to **Save**, **Cancel**, or **Eject** by moving the arrow to the appropriate choice and clicking the mouse. Again, **Eject** is used when secondary disk is employed (see Appendix B). This process does not change the name in the original program window. If the new name chosen has already been used, another dialog box appears, indicating that an existing file has the same name. See Figure 2.5.

If you click **Yes**, the file is saved and the file previously saved under that name is deleted. If you click **No**, control returns to the naming box. You can simply erase the name in that box by using the Backspace key and choose another name.

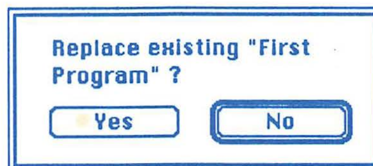


FIGURE 2.5

PRINT...

The **Print...** option in the File menu displays a dialog box requesting information concerning the quality, page range, number of copies, and type of paper feed employed in printing the contents of the program window. See Figure 2.6.

The darkened circles indicate the default options. You may choose to alter these selections to suit a particular application. When **OK** is chosen, a copy of the listing in the program window is sent to the printer.

Quality: ○ High ◉ Standard ○ Draft [OK]
Page Range: ◉ All ○ From: [ ] To: [ ]
Copies: [1]
Paper Feed: ◉ Continuous ○ Cut Sheet [Cancel]

FIGURE 2.6

QUIT

The last option in the **File** menu is the **Quit** option. When this option is selected, control leaves Macintosh Pascal and returns to the Macintosh system screen. If changes have been made in a window and have not been saved when the **Quit** option is selected, the system asks whether the changes should be stored or not. See Figure 2.7.

At this point you can select to **Save** or **Discard** the currently opened program file or cancel the **Quit** option. If **Cancel** or **Discard** is selected, the system returns to the Macintosh system screen.

Do you want to save or discard the changes to your program before closing?
[Save] [Discard] [Cancel]

FIGURE 2.7

2.3 EDIT OPTIONS

The next option in the menu header at the top of the Pascal screen is **Edit**. The **Edit** choices can be used to make changes in the program displayed in the program window and to provide additional commands that can be of valuable assistance when altering larger, more complex programs.

Figure 2.8 illustrates the options in the **Edit** menu.

FIGURE 2.8

The best way to become familiar with certain procedures is by stepping through an exercise with the Macintosh. Several of these Hands-On Exercises appear in this chapter.

HANDS-ON EXERCISE

After entering Pascal from the system window, copy the program `EditExample` into the program window exactly as follows. The inversed type that initially appears in the program window (see Figure 2.1) disappears when you start typing.

```
program EditExample;
begin
 writeln('first');
 writeln('second');
 writeln('third')
end.
```

Select the second **writeln** line of that program for editing by pressing the mouse button and dragging it across that line. The selected portion of the program appears in inverse print, as shown in Figure 2.9.

When part of a program has been selected, additional options in the **Edit** menu become available for use.

Continued

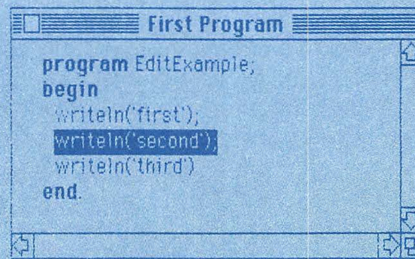


FIGURE 2.9

CUT

The **Cut** option allows you to remove text from the program window. Select this option by pressing the mouse button, dragging the arrow down the menu listing, and releasing it while the arrow is on **Cut**. The selected line from the program window no longer appears. It has been transferred to a window called the **Clipboard**. The **Clipboard** temporarily stores data removed from the program window.

To display the contents of the **Clipboard**, choose **Clipboard** from the **Windows** menu. See Figure 2.10.

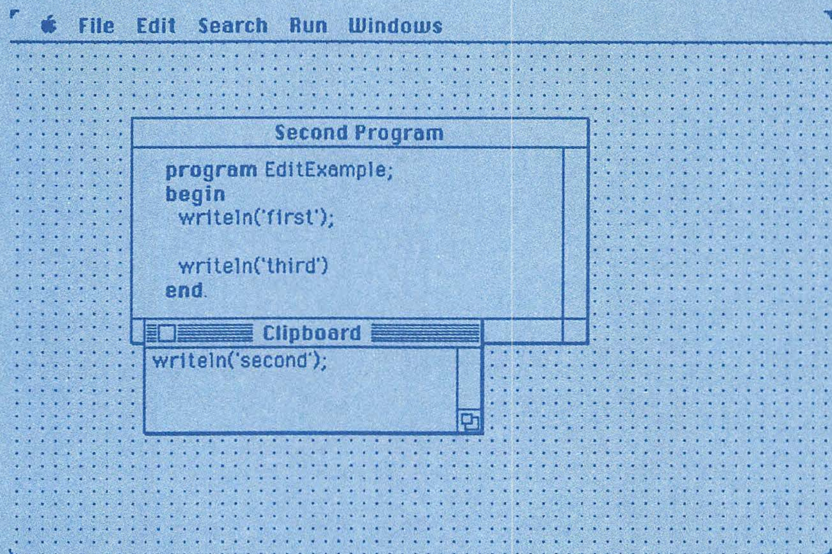


FIGURE 2.10

Continued

The **Clipboard** window shows the portion that was removed from the program window. As with all other windows, the **Clipboard** window can be moved around the monitor screen and its size changed.

The **Clipboard** must be removed from the screen before any other **Edit** options can be chosen. To remove the **Clipboard**, click the mouse while the arrow is in the square in the upper left corner of the **Clipboard** window.

PASTE

The **Paste** option allows you to move text from the **Clipboard** window to the program window. Move the cursor to a position immediately preceding the **end**. Now select **Paste** from the **Edit** menu. **Paste** takes the current contents of the **Clipboard**, `writeln('second');`, and moves it to the cursor position in the program window. Figure 2.11 shows the results of this **Paste** operation.

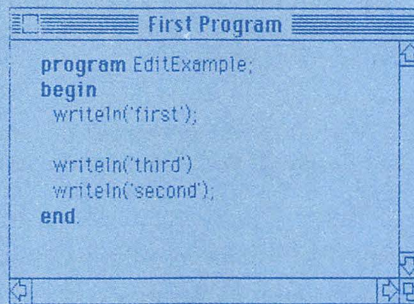


FIGURE 2.11

COPY

Copy leaves the program window unaltered but makes a copy of the selected portion of the program window in the **Clipboard**. Select `writeln('first');` by dragging the mouse across the instruction until the entire line is in inverse print. Select **Copy** from the **Edit** menu. The previous contents of the **Clipboard** are erased. Choose **Clipboard** from the **Windows** menu to confirm this last operation.

Remove the **Clipboard** and move the cursor in the program window to the line following `writeln('second');`. Select **Paste** from the **Edit** menu and a second copy of `writeln('first');` appears in the program window. See Figure 2.12.

Continued

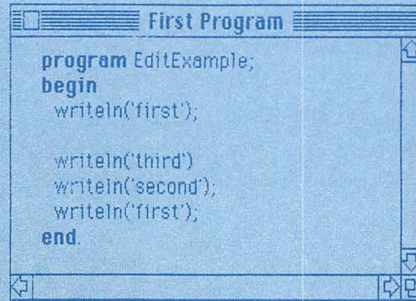


FIGURE 2.12

CLEAR

Clear erases a portion of a program with no chance for retrieval. Select `writeln('second');`. Select **Clear** from the **Edit** menu. Now `writeln('second');` is removed from the program window, but it is not saved in the **Clipboard.**

If the **Clipboard** is now shown, its previous contents, `writeln ('first');`, appear. Remember that the **Clipboard** stores only the last portion of the program "clipped" from the program window.

SELECT ALL

The **Select All** option inverses the entire program in the program window.

Choose **Select All** from the **Edit** menu. Figure 2.13 shows the contents of the program window after this option has been chosen.

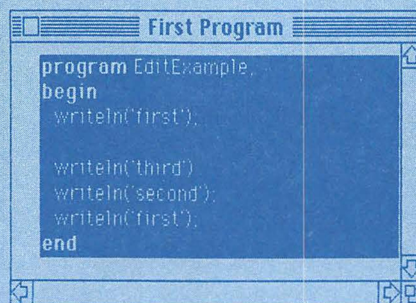


FIGURE 2.13

Choose **Cut.** The entire program is transferred to the **Clipboard,** and the program window is empty. The entire program can be returned to the program window by selecting **Paste** from the **Edit** menu.

2.4 EDITING ALTERNATIVES

When you are using the keyboard to enter a program in the program window and you make an error, there are several ways to correct it, including using the options in the **Edit** menu. If you notice the error immediately, you can employ the Backspace key on the keyboard to erase the incorrect character and then type the correct one. A disadvantage of using this method is that all the characters that lie between the cursor and the incorrect character are also erased and must be retyped.

To correct an error in the text that is not adjacent to the current position of the cursor, reposition the cursor at the beginning of the incorrect portion of the text in the window by moving I, the symbol for the arrow on the program window, and clicking the mouse. The vertical line cursor symbol now precedes the text to be corrected. Depress the mouse button and drag the mouse across the incorrect text. All the characters covered change to inverse mode. Release the button on the mouse and the inversed characters disappear as you type the correct version of the text. When this procedure is complete, you can move the cursor to any other position by using the mouse, as described earlier.

To illustrate this procedure, let's step through an exercise.

Consider the program `SearchExample` shown in Figure 2.14. After entering Pascal from the system window, copy `SearchExample` into the program window.

HANDS-ON EXERCISE

Second Program

```
program SearchExample;
begin
  writeln('Good morning!');
  writeln('The whether is clear and mild.');
  writeln('This is jogging whether.')
end.
```

FIGURE 2.14

Use the **Save As...** option from the **File** menu and save this program under the file name `Second Program`.

In Figure 2.14, the word *weather* is misspelled. Move the cursor between the *w* and the *h* and drag the cursor across the letters *he*. See Figure 2.15.

Continued

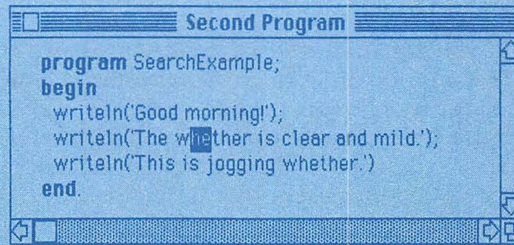


FIGURE 2.15

Now type the letters *ea.* See Figure 2.16.

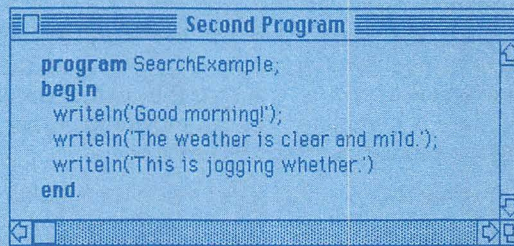


FIGURE 2.16

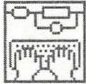
The error in this line has now been corrected. This process is repeated to correct the error in the next line. The following section provides a method for correcting identical errors at the same time.

Save this program on your disk under the name `Third Program` using the **Save** or **Save As...** option from the **File** menu.

2.5 SEARCH OPTIONS

The **Search** option in the Pascal window can be used to look for and/or replace text in the program window. This process can prove very useful in the editing of a program. Figure 2.17 gives a list of the options available in the **Search** menu.

FIGURE 2.17



HANDS-ON EXERCISE

For the purposes of illustration, recall the file `Second Program` (Figure 2.14) from your disk by activating the **Open** option from the **File** menu or again typing the code into the program window. You may first have to use **Close** from the **File** menu to remove any previous program from the program window. This is the original version of the previous program with both incorrect spellings of *weather*.

Elongate and move this window to the lower half of the monitor screen as shown in Figure 2.18.

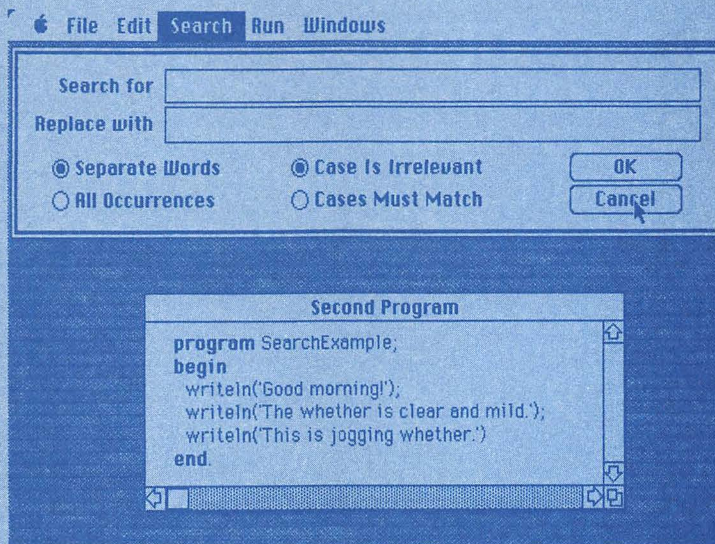


FIGURE 2.18

Continued

WHAT TO FIND...

Select the **What to find...** option from the **Search** menu. A dialog box similar to the top half of Figure 2.18 is produced.

In the **What to find...** dialog box, the flashing vertical line cursor symbol should appear to the right of **Search for.** If it does not, move it there by using the mouse. If any other text appears in the window and is entirely inversed, press the Backspace key. Now select a unique set of characters that contains the characters to be replaced. Note that *he* cannot be selected because if it is, the word *the* is also changed. The letters *whe* can be typed in this area since *whe* is unique to both incorrect words. Move the cursor to the writing area to the right of **Replace with**. Erase any previous text in that area by using the method described above. Now type the letters *wea.* The letters *wea* can now be used to replace the letters *whe.*

Two additional selections can be made in this box. If **Separate Words** is selected, the replacement takes place only if the text in the **Search for** and **Replace with** areas includes complete words. If this option is chosen in the example just considered, the text of `Second Program` would not change since the letters *whe* and *wea* are not individual words, but are embedded in or parts of other words, namely *whether* and *weather.* In order to select the other option, **All Occurrences**, move the arrow to the circle before that phrase and click the mouse once. The little dot should move from the circle before **Separate Words** to this circle.

The second choice is between **Case Is Irrelevant** and **Cases Must Match**. When the option **Case Is Irrelevant** is selected, the distinction between uppercase and lowercase letters is ignored. If uppercase and lowercase differences are important, select the option **Cases Must Match**. Figure 2.19 shows the dialog box completely filled in for this application.

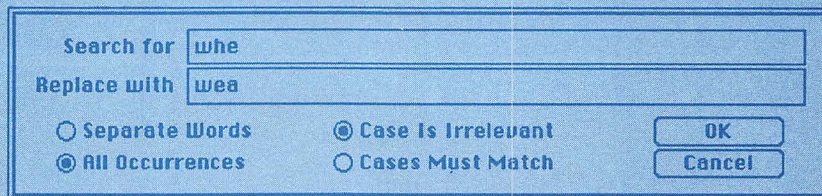


FIGURE 2.19

EVERYWHERE

The **Everywhere** option allows you to replace text for every occurrence in a program. Use the mouse to select the option **OK** in the **What to find...**

Continued

box. The box disappears. Go back to the **Search** menu and choose **Everywhere.** Another dialog box is displayed to verify your intent. See Figure 2.20.

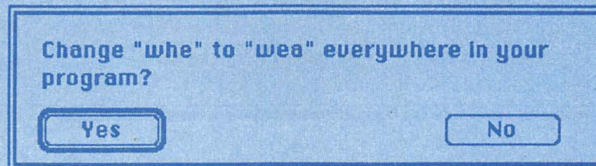


FIGURE 2.20

Choose **Yes.** This selection changes all the embedded letters *whe* to *wea,* regardless of case. The finished product appears in Figure 2.21.

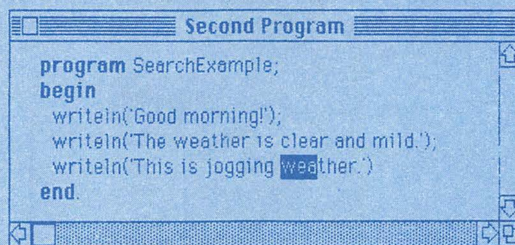


FIGURE 2.21

FIND

The option **Find** from the **Search** menu points out the first occurrence of the incorrect text in the program window as identified in the **Search for** line in the **What to find...** box. To find something else, return to the **What to find...** option and change the text in the **Search for** area by entering the word **Good.** Now after selecting **Find** the program in the program window looks like the one in Figure 2.22. You can now enter the program window directly and make the desired changes.

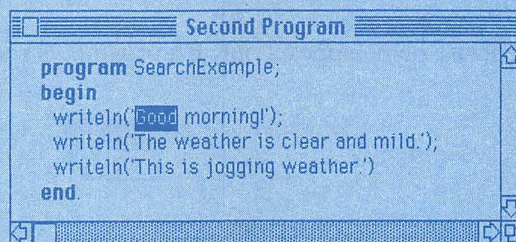


FIGURE 2.22

Continued

REPLACE

The **Replace** option uses the information in the **What to find...** box to change the first occurrence of the text following the cursor position in the **Search for** area with the text in the **Replace with** area. For example, if this option had been used rather than **Everywhere** in Figure 2.21, only the first *whether* would have been changed.

2.6 THE RUN AND PAUSE MENUS

The commands that direct the computer to process the instructions in a program are included in the **Run** menu. The options contained in this menu are listed in Figure 2.23.

Although the **Run** menu contains many options, only the **Go** and **Reset** options are presented here.

FIGURE 2.23

GO

Choosing the **Go** option causes the program to be executed from the beginning of the program.

PAUSE

The **Pause** menu appears only if a program is running (see Figure 2.24).

FIGURE 2.24

Depressing the mouse button while the arrow is on the menu header causes the program to stop execution. A pointing finger appears at the place where the program execution pauses. If the button is released, the program continues execution from the point where it was halted.

HALT

If the **Halt** option is selected when the **Pause** menu is on the screen, the execution of the program ceases and can be restarted using **Go** from the **Run** menu. As with **Pause**, the program continues from the point at which it was halted unless the **Reset** option is chosen before **Go**.

RESET

The **Reset** option initializes program execution so that the **Go** option executes a program from the beginning; that is, as though it were never stopped at all.

2.7 THE WINDOWS MENU

The choices in the **Windows** menu permit you to activate windows and control the size of type used in these windows. Figure 2.25 displays these options.

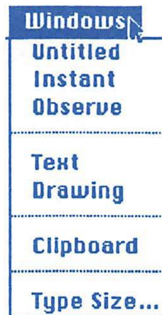


FIGURE 2.25

Only one window may be active at one time. An active window is characterized by a series of parallel lines to the left and right of the window name. To activate an inactive window, either choose that window option from the **Windows** menu, or click the mouse while the screen arrow appears in that inactive window. An active window always appears in front of any inactive window.

If any of the windows listed under the **Windows** menu appear on the screen, they can be removed by clicking the mouse while the screen arrow is in the box in the upper left corner of these windows. Any window listed in the **Windows** menu that does not appear on the screen can be activated by clicking on its

name in the menu. These windows can also be enlarged and moved as described in Chapter 1.

Recall that when initially booted the system automatically displays the program window (**Untitled**), the **Text** window, and the **Drawing** window.

UNTITLED

The **Untitled** option activates the program window. The name of the file at the top of the program window appears as the first entry in the **Windows** menu. If you have not given a program file a name, the name **Untitled** is used. In Figure 2.26, **SAMPLE** appears at the top of the program window and also appears as the initial choice in the **Windows** menu.

When the program window is closed, this option is no longer available. No file name replaces the name of the previous program, and it appears dimmed in the **Windows** menu. To reactivate this option, a program file must be opened using **New** or **Open...** from the **File** menu.

TEXT

An activated **Text** window displays the text output (letters, numbers, and special characters) that results from the execution of a program. If this window does not appear on the screen when a program is executed, text output is not visible. If the **Text** window is then activated, output from the program is displayed. If

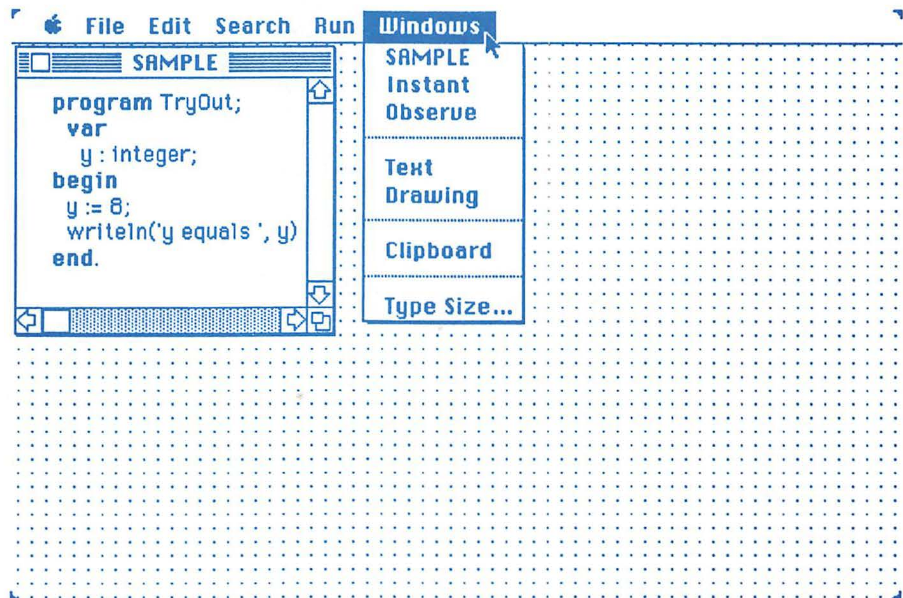


FIGURE 2.26

no text output is required, this window should be removed from the screen to allow additional room for other windows.

DRAWING

An activated **Drawing** window displays graphic output produced by a program that uses the graphic abilities of Macintosh Pascal. If no graphic output is required, this window should be removed from the screen to allow additional room for other windows.

CLIPBOARD

The **Clipboard** window displays any text cut or copied while editing a program. Details of its use were explained in the Hands-on Exercise on the **Edit** menu.

TYPE SIZE...

Choosing **Type Size...** from the **Windows** menu displays a dialog box on the screen. This box allows you to select the size of the characters displayed in the windows (see Figure 2.27).

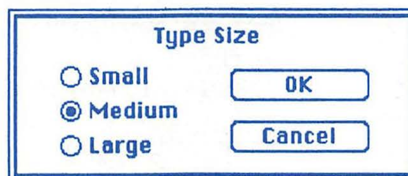
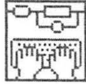


FIGURE 2.27

To change the type size from **Medium** (the default type) to **Small** or **Large**, use the mouse to move the screen arrow to the circle preceding the desired choice and click the mouse button. Then click the mouse button while the screen arrow is on **OK** to complete the selection.

2.8 A PROGRAMMING SESSION



HANDS-ON EXERCISE

The following step-by-step exercise illustrates and emphasizes the fundamentals of entering and executing a program using Macintosh Pascal. Although it may be a little premature for you to write and execute a Pascal program, the exercise nevertheless helps summarize elementary operating system concepts. You should try to follow the directions and become familiar with the Macintosh operating system.

Turn on the Mac and insert the Macintosh Pascal disk.

Turn on the printer.

Double-click on the **Macintosh Pascal** icon to enter the Macintosh Pascal screen.

Hit the Backspace key to clear the program window.

Enter the following program by typing the instructions at the keyboard. The results appear in the **Untitled** program window. Enlarge the window by increasing its width to equal that of the entire screen. The indentation and emphasis of keywords is automatically done by the system.

```
program Total;
(* Find the sum of the numbers from 1 to 100 *)
 var
  sum, number : integer;
begin
 sum := 0;
 for number := 1 to 100 do
  sum := sum + number;
 writeln ('The sum is ',sum)
end.
```

Reduce the size of the program window so that the **Text** window becomes visible.

Get a listing of these program instructions on paper by selecting **Print...** from the **File** menu.

Click **OK** in the response box that appears in that box.

Move and change the size of the **Text** window so that it appears below the program window and extends to the width of the screen.

Activate the program window by clicking the mouse button while the arrow is in the program window.

Continued

Save this file under the name `Example 1` by using **Save As...** from the **File** menu. Notice the title change in the program window.

Execute the program by choosing **Go** from the **Run** menu.

Close this program file by choosing **Close** from the **File** menu.

Check to see if this save has been accomplished by selecting **Open...** from the **File** menu. The program named `Example 1` should appear. The list may have to be scrolled until it appears.

Open `Example 1` by double-clicking the mouse button while the screen arrow appears on the name of the program.

Choose **What to find...** from the **Search** menu and search for *100* and replace it with *25*. Use **Separate Words** and **Case Is Irrelevant.**

Go back to the **Search** menu and select **Everywhere.** Each *100* in the listing window should have been changed to *25*. Remember that the change does not take place until this command is issued.

Add the underscored program instructions to make the program appear as follows:

```
program total;
(* Find the sum of the numbers from 1 to 25 *)
 var
  sum,number : integer;
  f : text;
begin
 rewrite (f, 'printer:');
 sum := 0;
 for number := 1 to 25 do
  sum := sum + number;
 writeln ('The sum is ',sum);
 writeln (f, 'The sum is ',sum)
end.
```

Get a listing of the changed program on the printer.

Make sure the program window is active.

Use **Save** to automatically save the new version of the program under the name `Example 1.`

Execute the changed program. A new **Text** window is generated, and output is produced on the printer.

Make sure the program window is active and select the **Save As...** option to save the same program under the new name `Example 2.`

Continued

Quit Macintosh Pascal by choosing the **Quit** option from the **File** menu.

Trash `Example 1`.

Verify that `Example 1` has been "trashed" by double-clicking the mouse button while the arrow is on the **Trash** icon.

Empty the **Trash** by choosing that option from the **Special** menu. `Example 1` should disappear from the **Trash** window.

Choose **Eject** from the **File** menu and remove the disk.

Turn off the Macintosh and the printer.

In general, it is a good idea to save your program before you attempt to execute it. Then, in case your program contains a fatal error that shuts down the system, you will not lose your program. You can then reboot MacPascal, load the saved copy of your program, correct the errors, resave the program, and try to execute it.

EXERCISES

Indicate the step or steps a user should take to accomplish each of the following tasks on the Macintosh computer:

1. Execute a program.
2. Save a program or file under the name *Max.*
3. View the options contained in the **File** menu.
4. Change the size of output text characters to large type.
5. Remove a disk from the disk drive.
6. Scroll a program upward or downward.
7. Determine if a program has been saved as a file in storage.
8. Erase the text window.
9. Save a file under another name.
10. Transfer a stored program from the disk into main memory.
11. Move any window.
12. Change the size of any window.
13. Save a file after changes to it have been made.
14. Select an option from the **File** menu.
15. Correct the spelling of a word that appears four times in a program.
16. Obtain a listing of program instructions on paper.

APPENDIX A

Menu Key Equivalents

Many menu commands on the Pascal screen can be chosen by holding down the command (⌘) key and pressing the appropriate key (either upper- or lowercase), instead of by using the mouse. The following table lists these key equivalent options.

MENU ITEM	*KEY*	*MEANING*
Copy	C	Copy selected text onto **Clipboard.**
Cut	X	Remove selected portion of program; place on **Clipboard.**
Everywhere	E	Replace all occurrences of previously specified string.
Find	F	Find next occurrence of previously specified text.
Go	G	Run program in active window.
Paste	V	Put **Clipboard** contents at the cursor position.
Replace	R	Replace next occurrence of previously specified string.
Select All	A	Select entire document in active window.
Step	S	Show execution of program one statement at a time.
What to find	W	Specify search and replace options in **Find** menu.

A P P E N D I X B

Using Supplementary Disks

The Macintosh Pascal disk has limited space to store user programs. If blank disks are properly prepared for the system, they can be used to store programs. The process of preparing disks for this use is termed *initialization.*

Although initialized disks can provide additional storage for programs, they cannot be used to start or boot the system. A startup disk is required for that purpose. The Macintosh manual that accompanies the computer gives specific details on creating a startup disk. Since a startup disk contains additional system information, it has considerably less space available for the storage of programs than a disk that is merely initialized.

MacPascal provides for initialization when a blank, unformatted disk is put in the drive. The following step-by-step process illustrates the initialization of a blank disk, the storage of a program on that disk, and the opening of a program file from a disk other than the Pascal disk.

After creating a Pascal program, select **Save As...** from the **File** menu. Figure B.1 shows a dialog box that appears on the screen.

Pascal appears over the **Eject** option, naming the disk currently in the drive. Now, give the new program file a name by entering that name in the rectangular area in the box. Figure B.1 indicates that the name **SAMPLE** was chosen.

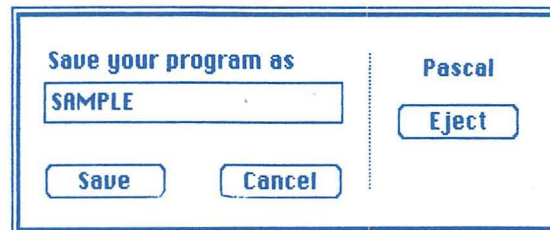


FIGURE B.1

Select **Eject** by clicking the mouse button while the screen arrow is in the **Eject** oval. After the Pascal disk has been ejected, insert a blank, unformatted disk. The system does not recognize the disk and displays a message to that effect, as shown in Figure B.2.

FIGURE B.2

Select **Initialize** by clicking the mouse button on the oval containing **Initialize. Initializing disk...** appears in the dialog box, and after a brief period of time, the system displays **Untitled** in inverse print in a dialog box—in effect, asking for a name for the new disk so it can be identified. Enter the name **Programs** (see Figure B.3).

FIGURE B.3

Select **OK** with the mouse button. The system then displays the name for the program file previously entered (see Figure B.4).

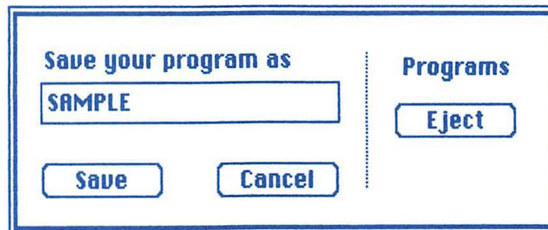


FIGURE B.4

The name given to the new disk, **Programs**, appears over the **Eject** oval.

Select **Save** by using the mouse button, and a copy of the program file **SAMPLE** is stored on the Programs disk.

After the Programs disk has been ejected, put in the Pascal disk. Deactivate the program file **SAMPLE** by choosing **Close** from the **File** menu. **SAMPLE** is no longer in main memory.

To recall a program from another disk, choose **Open...** from the **File** menu. A catalog of the Pascal programs stored on the Pascal disk currently in the drive is displayed as shown in Figure B.5.

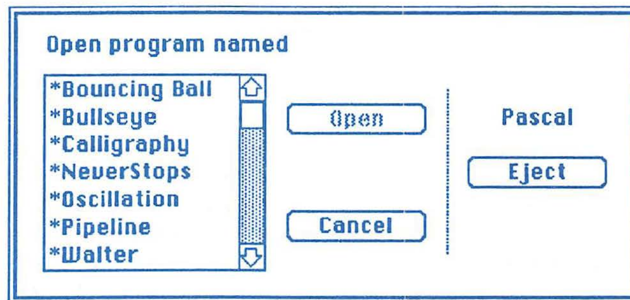


FIGURE B.5

Since the program **SAMPLE** is on the Programs disk, it is necessary to eject the Pascal disk using the Eject oval and insert the Programs disk. After this is done, the screen appears as in Figure B.6.

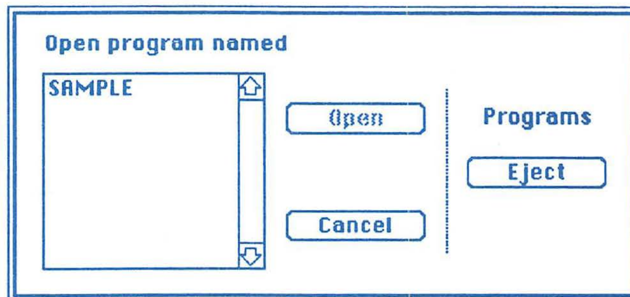
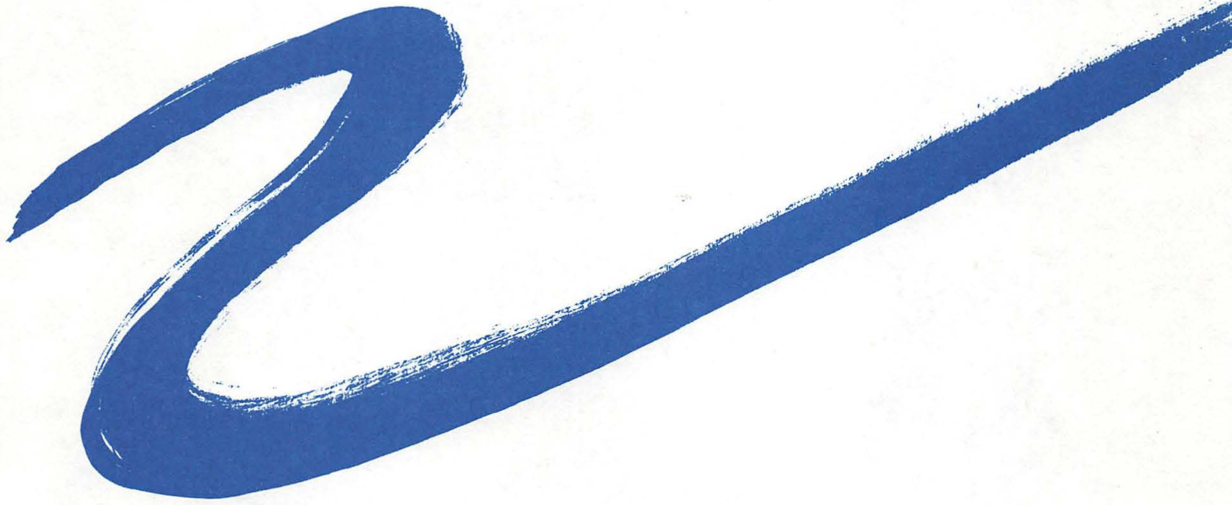


FIGURE B.6

Select **SAMPLE** by double-clicking the mouse button while the screen arrow is on that name or by single-clicking on that name and choosing **Open**. After the Programs disk is ejected, insert the Pascal disk. The **SAMPLE** program is now in main memory and displayed in the program window entitled **SAMPLE**.

Any selection from the header menus may require the reinsertion of the Pascal disk. In that case, the system automatically ejects the Programs disk and prompts the user to put the Pascal disk back into the drive.

PART ONE



FIRST STEPS IN PASCAL PROGRAMMING

C H A P T E R 3

Design and Construction of Problem Solutions

3.1 INTRODUCTION

Problems are a fact of life. In order to solve a problem, it must be analyzed first. For example, suppose you want to start a new business. An initial investigation might be made to determine what is needed to run the business. These needs might include fixtures and equipment, amount and type of inventory, number and type of personnel, days and times of operation, and so on. Constructing a list of requirements is the first step in the solution process. A second step may be to study a similar business that is already operating. Finally, a detailed plan is constructed and implemented. Then equipment and inventory are purchased, personnel hired, and the new business started. If difficulties developed, you would have to be ready to modify the original plan to handle whatever had not been anticipated.

Although a considerable amount of time may be spent in the planning stage, in the long run a well-planned solution produces effective results. A hastily drawn plan frequently ends in utter confusion and wastes time and money. Careful organization at the planning stage is the key to good results.

This text features a general six-step methodology for solving problems with specific applications to the Macintosh computer. Using an organized planning scheme in the early stages of problem solving should enable you to solve complex problems with greater ease.

The problem-solving methodology in this chapter provides a good basis for the creation of well-written programs. A more detailed study of the design and construction of programs that offer solutions to large, complex problems is presented in Chapter 12. At that stage you should have developed an appreciation and an understanding of just what it takes to write a program.

3.2 PROBLEM SOLVING METHODOLOGY

The specific methodology for the solution of problems on a computer is as follows:

Step 1: Analyzing the problem
Step 2: Creating an input-processing-output (IPO) chart
Step 3: Looking for clues
Step 4: Planning a solution using a Nassi-Shneiderman (N-S) chart
Step 5: Coding the solution
Step 6: Testing and debugging the solution

Five of the six steps listed here are language-independent: the steps in the planning process apply to the solution of a particular computer problem using any programming language. Step 5 is language-dependent and concerns writing the program solution in a specific programming language—in this text, Macintosh Pascal.

Step 1: Analyzing the Problem A problem must be understood before it can be solved. You must carefully read the statement of the problem and identify the data given and the results required. The data presented in the statement generally constitute input data in the program. The results required generally constitute the output to be displayed. Essentially, then, analyzing a problem becomes a matter of determining input and output (I/O).

Step 2: Creating an Input-Processing-Output (IPO) Chart Making a record of a process is called *documentation* and is essential at every stage of the programming process. One way to provide documentation is to list and describe all the quantities involved in a problem and to identify the names that are used to represent them in the solution. The names are called *identifiers.* An IPO (input-processing-output) chart satisfies this need; see the sample in Figure 3.1.

The chart has three sections: INPUT, PROCESSING, and OUTPUT. The quantities that must be supplied to the computer are listed in the INPUT section. Appropriate names should be selected for each, and the names (identifiers) should be listed in the first column.

The PROCESSING section includes the identifiers that must be calculated for solving the problem. These are intermediate quantities used to calculate output results or are the output results themselves. It may be necessary to return to this section of the IPO chart after Step 4 of the process, because the intermediate quantities needed in the solution may become known at that time. Quantities listed in the PROCESSING section should be accompanied by brief descriptions.

As the name implies, the OUTPUT section should contain only (1) those identifiers whose values are to be displayed and/or (2) descriptions of pictures to be drawn. No descriptions for identifiers are necessary, because the names

IPO CHART

Class	Identifier	Description
INPUT		
PROCESSING		

	Identifiers		Drawing
OUTPUT			

FIGURE 3.1

of these quantities should have already appeared in either the INPUT or PROCESSING section of the IPO chart.

In this chapter, identifiers are highlighted by displaying the entire name in uppercase letters. This characteristic makes it easier for you to recognize them in discussions and explanations.

Step 3: Looking for Clues Techniques needed to solve a particular problem are similar to techniques used in previously solved problems. The general character of a problem suggests the clues to look for. For example, one problem may involve the calculation of discounts and another payroll calculations, even though both involve a deduction of quantities from a given amount. You should use this step as part of the general methodology, although searching for clues does not always prove successful.

Each problem that you can solve adds to your background and experience. Practice develops good programming skills.

Step 4: Planning a Solution Using a Nassi-Shneiderman (N-S) Chart This step in the process involves reviewing all the information gathered in the previous steps and designing an outline of the procedures to be used in solving the problem, that is, bridging the gap between the input and the output. Many of the techniques employed by professional programmers involve the use of diagrams to illustrate the solution process. In addition to permitting easy visual comprehension, diagrams provide additional documentation to aid in correcting errors and later modifying programs. Diagrams show solution plans in ways that are intelligible to programmers and nonprogrammers alike.

This text makes use of a special type of schematic diagram called a Nassi-Shneiderman (N-S) chart. The chart is simple to produce and understand. It allows you to design a "computer" solution to a problem without any knowledge of a specific programming language. In addition, its structure permits easy transformation from problem solution to Macintosh Pascal code.

As with any problem, the more complex it is the greater the benefit to be derived from the plan. You may find that schematic charts provide little assistance in solving elementary problems. However, the experience gained through their use is of greater benefit when more complex problems are encountered. When properly used, schematic diagrams provide a blueprint for the coding operation.

Step 5: Coding the Solution Once the problem has been analyzed, the IPO chart constructed, and a plan of solution designed, use Step 5 of the methodology. This step involves the translation of the logic design of the solution into the actual program instructions that produce the desired results. The N-S chart serves as the main guide for this step. The translation into program code—or *coding,* as it is usually called—is initially written on paper. The program is not ready for entry into the computer at this stage because a testing stage must still take place.

Although the coding in this text is done in Macintosh Pascal, another programming language may be substituted. The first four steps in the methodology

serve the same purpose. That is precisely the aim of the methodology, namely, to provide a universal procedure for analyzing and designing the computer solution to a problem.

Step 6: Testing and Debugging the Solution When this step is reached in the process, you should have a prototype program written on paper. Before you enter that program into the computer for a trial run, you should select representative sets of data and test the program with those data to ensure that correct results are obtained. This hand-testing of the program is referred to as a *walkthrough.* In the event that a walkthrough produces incorrect results or no results, you should return to the beginning of the problem-solving procedure to try to locate the error. The process of locating an error in a program is called *debugging,* because an error in a program is referred to as a *bug.*

Two types of errors may be present in a program: (1) a syntax error and (2) a logic error. Syntax errors include misspelled keywords, incorrect use of punctuation, and the like, for example, the use of **writelm** instead of **writeln.** Macintosh Pascal provides automatic syntax error checking as each instruction is entered into the program window. When an error is detected, it is indicated by a change in the typeface of a portion of the code displayed in the program window. At that point, you may correct the error.

Logic errors, on the other hand, are more difficult to detect. They arise when instructions are intelligible to the computer (syntactically correct), but the results the program produces are not those you intended. For example, the program may include an instruction to add two quantities, when the operation should have been multiplication. Careful walkthroughs may be necessary to uncover these errors.

3.3 PRACTICAL APPLICATIONS

The following examples illustrate the problem solving methodology presented in this chapter. Elementary problems are used because you are at the beginning stages of programming. The coding stage, Step 5, is omitted and presented in Chapter 5 when complete programs are written.

Example 1: The DiskCount Record Shop offers customers a 12 percent discount on the total amount of purchases. Write a program to accept the total amount of a purchase, then determine and display the discounted price.

STEP 1: Analyzing the Problem
In this problem, the amount of the purchase is entered. The discount is 12 percent, which can be expressed as 0.12. The decimal form for this value is used to calculate the amount of the discount. The amount of the discount is then subtracted from the amount of the purchase to produce the final discounted price.

STEP 2: Creating an IPO Chart

The INPUT section of the IPO chart contains the total amount of the purchase. A suitable name for this quantity is TOTAMT. Because the final discounted price is a result of the processing in the program, that value should be included in the PROCESSING section of the IPO chart. FINALPRICE is a good choice for a name. The OUTPUT section of the IPO chart includes both TOTAMT and FINALPRICE. See Figure 3.2 for the completed IPO chart.

IPO CHART

Class	Identifier	Description
INPUT	TOTAMT	total amount of purchases
PROCESSING	FINALPRICE	discounted price
	DISAMT	amount of discount

	Identifiers		Drawing
OUTPUT	TOTAMT		
	FINALPRICE		

FIGURE 3.2

When you are devising a plan for the solution of a problem, new quantities may have to be calculated in intermediate steps to achieve the final solution of the problem. These should always be placed in the PROCESSING section.

STEP 3: Looking for Clues

Since this is the first example to be considered, there is no past experience to review for clues. Accordingly, the plan must be implemented without any familiar models to use for reference.

STEP 4: Planning a Solution

The solution plan must describe how to proceed from the input value to the desired output result. The problem states the discount as a percentage. Therefore, before the final discount price can be determined, the total

dollar value of the discount must be calculated. This calls for an intermediate step, namely, the calculation of the discount amount. A name for the discount amount is DISAMT. At this stage, you should return to the IPO chart and enter DISAMT, with its appropriate description, in the PROCESSING section.

The more intricate features of N-S charts are developed as the problem solutions become more complex.

Basically, however, a program is represented by a box, and this box is subdivided in a particular way to illustrate the thought process behind the solution to a problem. For the moment, the program box is divided by drawing horizontal lines to separate operations that are to be performed in a sequential order.

Recall the function of the N-S chart, and consider Figure 3.3.

FIGURE 3.3

This diagram describes the steps that lead to the solution of the problem. The order in which the steps are arranged in the N-S chart is the order in which they are to be executed. Then, when they are translated into instruction code, their order in the N-S chart dictates the order that the corresponding instructions must take in the correct solution.

Refer to the descriptions contained in each box in the N-S chart in Figure 3.3. These are not Pascal instructions. Diagrams should contain descriptions that are language-independent. If the descriptions are English-like, the same chart may be used for programming in different programming languages. Of course, you may find it helpful to use the same names selected in the IPO chart when describing instructions in the N-S chart.

STEP 5: Coding the Solution

The N-S chart is used as a guide for coding the solution. The Pascal coding representing the solution would be written at this stage of the process. This step in the process is detailed in Chapter 5.

STEP 6: Testing and Debugging the Solution

You should walk through the solution with a set of sample data. Let's select $79.00 for TOTAMT. A sample walkthrough appears as follows:

```
DISAMT = .12 x TOTAMT = .12 x 79.00 = 9.48
FINALPRICE = TOTAMT - DISAMT = 79.00 - 9.48 = 69.52
```

The program is executed to see if the answer it produces matches the answer found in the walkthrough. If the program results do not match the correct results, you have to go to the beginning of the problem solving process, locate the error, make corrections, and retest before re-executing the program.

Example 2: In a three-day period during the summer, the city of Lincoln, Nebraska, recorded high temperatures of 87, 79, and 95 degrees Fahrenheit. Compute and display the average high temperature for that period.

STEP 1: Analyzing the Problem
In this problem, the three daily high temperatures are the given data and accordingly serve as the input. The desired result is the average high temperature for the three days, and is the output.

STEP 2: Creating an IPO Chart
Figure 3.4 lists a sample IPO chart. The three input names (TEMP1, TEMP2, and TEMP3) represent three distinct daily temperature values. AVGTEMP represents the average of the three daily temperatures.

STEP 3: Looking for Clues
Again in this example, as in the first, you have virtually no problem solving experience to call on. Looking for clues does not prove beneficial because this is the first program of this type that is being considered.

IPO CHART

Class	Identifier	Description
INPUT	TEMP1	high temperature for day 1
	TEMP2	high temperature for day 2
	TEMP3	high temperature for day 3
PROCESSING	AVGTEMP	average high temperature for 3-day period

Class	Identifiers	Drawing
OUTPUT	TEMP1	
	TEMP2	
	TEMP3	
	AVGTEMP	

FIGURE 3.4

STEP 4: Planning a Solution
The N-S chart is easy to construct. The first box calls for the entry of the three daily temperatures. Although three separate boxes in the N-S chart can be used for this purpose, combining them in one box does not cause confusion. Recall that the purpose of the N-S chart is to outline the plan for the solution. It is not intended that each box in the N-S chart correspond to a single Macintosh Pascal instruction. Once the three daily temperatures have been entered, the average is calculated and then displayed. Of course, the steps must be executed in the same order in which they appear in the N-S chart (see Figure 3.5).

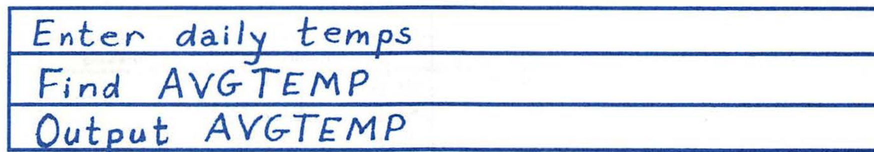


FIGURE 3.5

STEP 5: Coding the Solution
This step is added to the methodology in Chapter 5.

STEP 6: Testing and Debugging the Solution
In this problem, the actual data indicated in the statement of the problem are used for the walkthrough:

```
AVGTEMP = (TEMP1 + TEMP2 + TEMP3)/3
= (87 + 79 + 95)/3 = 261/3 = 87
```

The program should now be executed and the results compared. The run also produces 87, and that is the correct result.

Example 3: The Christopher Construction Company's employees work on an irregular daily basis; that is, the number of hours worked each day of the week depends on the particular job to be done. Accept a worker's name, hourly pay rate, and the number of hours that person worked on each of the days from Monday through Friday. Find and display the employee's gross pay, the average number of hours worked per day, and the employee's net pay, if 14 percent is deducted for taxes.

STEP 1: Analyzing the Problem
In this example, input data consist of the employee name, the hourly pay rate, and the number of hours worked on each of the five weekdays. The output should include the gross pay, the average number of hours worked per day, and the net pay for each employee.

STEP 2: Creating an IPO Chart
Figure 3.6 provides a sample IPO chart. The INPUT section contains the

IPO CHART

Class	Identifier	Description
INPUT	NAME	name of employee
	RATE	hourly pay rate
	HOURS1	no. of hours worked in day 1
	HOURS2	no. of hours worked in day 2
	HOURS3	no. of hours worked in day 3
	HOURS4	no. of hours worked in day 4
	HOURS5	no. of hours worked in day 5
PROCESSING	GROSSPAY	weekly gross pay
	AVGHOURS	average hours worked per week
	NETPAY	weekly net pay
	TOTHOURS	total hours worked per week
	DEDUCT	weekly deduction

Class	Identifiers		Drawing
OUTPUT	NAME	HOURS4	
	RATE	HOURS5	
	HOURS 1	GROSSPAY	
	HOURS2	AVGHOURS	
	HOURS3	NETPAY	

FIGURE 3.6

employee's name (NAME), the pay rate (RATE), and five distinct names representing the number of hours worked on each of the five days (HOURS1, HOURS2, HOURS3, HOURS4, and HOURS5). The PROCESSING quantities are represented by the names GROSSPAY, AVGHOURS, and NETPAY.

STEP 3: Looking for Clues

At last the search for hints pays off. In this problem, the average number of hours worked daily must be found. In Example 2, the average of three daily temperatures was calculated. Accordingly, a similar type of statement should be used in this problem. In addition, the net pay must be determined from the gross pay. This process involves subtracting a percentage of the gross pay from the initial gross pay and resembles the discount calculation from Example 1, so an analogous procedure can be used in this problem.

STEP 4: Planning a Solution

Several intermediate steps are now required to bridge the gap between the input quantities and the output results. A blueprint is provided in Figure 3.7.

The first box in the N-S chart describes the quantities to be input. Only one box is used in the chart for this purpose, although several separate quantities are to be entered. In order to calculate AVGHOURS, the total number of hours worked for the five days must be calculated first. Let TOTHOURS represent this quantity; return to the IPO chart and add that name in the PROCESSING section. The next box in the N-S chart describes the TOTHOURS calculation. The following box asks for the calculation of AVGHOURS, which can be found from TOTHOURS by a simple division.

Enter NAME ,RATE, 5 daily hours
Find TOTHOURS
Find AVGHOURS
Find GROSSPAY
Find DEDUCT
Find NETPAY
Output GROSSPAY, NETPAY, AVGHOURS

FIGURE 3.7

The next box calls for the calculation of GROSSPAY, which can now be found by multiplying TOTHOURS by RATE. Before calculating NETPAY, the deduction must be determined. Let DEDUCT represent this total, and enter it in the IPO chart. The DEDUCT value may be calculated by using the percentage operation. NETPAY, the value called for in the next box of the N-S chart, can be calculated by subtracting DEDUCT from GROSSPAY. Once all the quantities have been calculated, the final box in the N-S chart directs that they be displayed.

STEP 5: Coding the Solution
This step is detailed in Chapter 5.

STEP 6: Testing and Debugging the Solution
In this problem, there is no need to enter sample data for NAME, since it is not involved in the production of results but simply serves as a form of documentation. For the remaining variables, select the following values:

```
RATE = 9.00
HOURS1 = 5.5
HOURS2 = 8
HOURS3 = 3.5
HOURS4 = 6.5
HOURS5 = 4
```

The following calculations result:

```
TOTHOURS = HOURS1 + HOURS2 + HOURS3 + HOURS4 + HOURS5
         = 5.5 + 8 + 3.5 + 6.5 + 4 = 27.5
AVGHOURS = TOTHOURS/5 = 27.5/5 = 5.5
GROSSPAY = TOTHOURS x RATE = 27.5 x 9.00 = 247.50
DEDUCT = GROSSPAY x .14 = 247.50 x .14 = 34.65
NETPAY = GROSSPAY - DEDUCT = 247.50 - 34.65 = 212.85
```

When the program is executed, check the results produced by the program with these results and make appropriate changes in the program, if necessary.

These examples provide a brief introduction to the problem-solving methodology described in this chapter. The procedure takes on greater significance as problems become more sophisticated. It is important to learn and employ the steps of this procedure from the very beginning, so that they are familiar to you when problems become more complex. A structured way of thinking is developed, and good programming skills result from the constant application of specific principles. As you attempt and complete more problems, you should become more confident. The development of skill in programming becomes a matter of diligence and perseverance. We encourage you to be patient and thorough.

EXERCISES

Each programming exercise is preceded by a (B) or a (G) to indicate whether the problem is of a business or general nature.

For each of the problems in this section, analyze the statement of the problem and employ Steps 1, 2, 3, 4, and 6 to find a solution.

(G) 1. A student takes four exams in a course. The average of the first three is to count for 60 percent of the final grade, and the fourth exam is to count for 40 percent of the final grade. Calculate and output the final grade for the course.

(G) 2. Jack Handy wishes to paint the walls of his living room. The dimensions of the room are length, 14 feet; width, 12 feet; and height, 8 feet. He estimates that he can reduce the total wall area by 80 square feet to allow for windows and door openings. If one quart of paint covers 120 square feet, find the number of quarts Jack Handy needs.

(B) 3. An employee at a business office earns $7.60 per hour for the first 35 hours worked in one week, and time-and-a-half ($11.40) for all hours worked in excess of 35. If the employee worked a total of 39 hours during a week, find the gross weekly pay for the employee.

(B) 4. A shopper makes three purchases at the Friendly Discount Center. The list prices are $29.95, $32.50, and $19.75. If all merchandise at the center qualifies for a discount of 15 percent, find and output the total amount of discount and the total net selling price for the three items purchased.

(G) 5. A college student takes four courses during one semester. Her grades in these courses are as follows: *A* in English, *B* in speech, *C* in history, and *B* in mathematics. The English and history courses are three-credit courses, the speech course is a two-credit course, and the mathematics course is a four-credit course. The college uses a grade-point system to calculate semester averages: *A* counts 4 quality points; *B*, 3 points; *C*, 2 points; *D*, 1 point; and *F*, 0 points. Calculate and print the semester grade-point average for this student.

(B) 6. Mary Smith earns a weekly gross salary of $265. When her take-home pay is calculated, deductions are subtracted as follows: 12 percent for federal taxes, 7 percent for Social Security, 4 percent for state taxes, 3 percent for pension contribution, and

a fixed amount of $4.50 is deducted for hospitalization. Calculate and output Mary Smith's net take-home pay.

(B) 7. A salesperson earns a base weekly salary of $150. In addition, commissions are earned at a rate of 3 percent on the total weekly sales of $3,000 or more. Fred Miller sells a total of $3,400 during one week. Calculate and output his total weekly pay, including base salary and commission earned.

(B) 8. Ann Saver has three bank accounts. She has $4,000 on deposit at County Savings, which pays an annual simple interest at the rate of 5 percent; $3,500 on deposit at State Trust, which pays 6 percent; and $2,800 on deposit at Community Savings, which pays 5.5 percent. Find the total interest Ann Saver earns in one year and the updated amounts on deposit in each bank at the end of one year.

(G) 9. The area of a square is equal to the length of one side squared. The area of a circle is approximately equal to 3.14 times the radius squared. Find and output how much larger the area of a square is than the area of a circle if the square has a side of length 10 and the circle has a radius of 5.

(B) 10. If money is deposited in a bank that compounds interest, the amount on deposit *(A)* when a principal *(P)* is invested for N interest periods at a rate of interest per interest period *(I)* is given by the formula $A = P(1 + I)^N$. If $1,000 *(P)* is invested for 12 interest periods *(N)* at a rate of interest per period *(I)* of 1.5 percent, find the total amount *(A)* on deposit at the conclusion of the 12 interest periods.

CHAPTER 4

The Components of a Pascal Program

4.1 INTRODUCTION

Language is a means of communication between people; for example, we can have conversation, sign language, body language, and so on. Language is also the means of communication between a person and a computer. When a language is used for this purpose, it is called a *programming language.* Many programming languages have been developed since the 1950s. One of these languages, Pascal, was named after the French mathematician Blaise Pascal (1623–1662). The Pascal language was designed to provide you with the tools to write instructions for a computer. These instructions are like English so as to be easily understood by English-speaking programmers, reliable in use, and easily changed. Since its inception, Pascal has become a popular language for use with microcomputers. This text provides a comprehensive guide to Macintosh Pascal, a unique and powerful version of Pascal formulated specifically for the Macintosh microcomputer.

The Pascal programming language, like any other language, has its own rules for grammar and syntax. Each statement in Pascal is called an *instruction.* Instructions are designed to convey messages that direct the computer to perform certain operations. These instructions must be formulated according to those rules and must be executed in a particular order to produce correct results. The sequential listing of the instructions is called a *program.*

4.2 GENERIC DATA TYPES

Computers receive data, process those data, and output the results of the processing. The processing may be done on numbers or words. Pascal requires that you specify the type of data to be used so that it can properly allocate memory and use appropriate operations. Pascal data specifications follow.

INTEGER

An integer is a whole number and contains no fractional or decimal part. An integer value in a Pascal program can range from -32768 to 32767. When entering integer data types in a program, only the character symbols + and − and the digits 0 through 9 are permitted. Decimal points, commas, and slashes are not allowed.

LONG INTEGER

When integer results fall outside the range just specified, the long integer (**longint**) type should be used. The range of values for numbers of this type is from -2147483648 to 2147483647.

REAL

Numeric values that contain fractional or decimal parts are considered to be real numbers and can be expressed in standard decimal form or in e-notation, a form of scientific notation. For example, 1234.56 is a real number. It can also be expressed in scientific notation as a real number multiplied by a power of 10, as in 1.23456×10^3. The exponent of 10 indicates the number of places the decimal point was moved to the left in the original number so that only one integer remains to the left of the decimal point in the scientific representation. Macintosh Pascal shortens this representation to 1.2e+3, by rounding the decimal part. Here, e represents multiplication by a power of 10. Additional examples follow:

5678912321 is equivalent to 5.7e+9
−783571 is equivalent to −7.8e+5

In a similar fashion, .00008812 can be represented by 8.812×10^{-5} in scientific notation, and 8.8e−5 in Macintosh Pascal shorthand. The negative power of 10 indicates the number of places the decimal point was moved to the right in the original number so that only one integer appears to the left of the decimal point in the scientific or e-notation representation. Additional examples follow:

.000000000439372 is equivalent to 4.4e−10
−.000000813471 is equivalent to −8.1e−7

The system uses e-notation to express real values if no other format is specified. Real values can range from ±1.5e−45 to ±3.4e+38. Commas are not allowed in the Pascal representation of real numbers.

DOUBLE

Real numeric quantities that must have larger or more precise values can be expressed by a double-precision real. The double-precision real values can range from ±5.0e−324 to ±1.7e+308.

TABLE 4.1

TYPE	*DESCRIPTION*
Integer	Whole numbers between −32768 and 32767
Long integer	Whole numbers between −2147483648 and 2147483647
Real	Decimal numbers with precision range ±1.5e−45 to ±3.4e+38
Double-precision real	Decimal numbers with precision range ±5.0e−324 to ±1.7e+308
Extended-precision real	Decimal numbers with precision range ±1.9e−4951 to ±1.1e+4932
Character	Single keyboard character
String	0- to 255-character sequence
Boolean	True or false

EXTENDED

A greater degree of precision can be obtained by using extended-precision real values. These real values range from ±1.9e−4951 to ±1.1e+4932.

CHARACTER

Single letters or special characters can also serve as data in a program, and Macintosh Pascal defines a single keyboard character to be of type character. *A*, *%*, *i*, and + are all examples of data type character.

STRING

A value containing more than one character is classified as a string data type. This value type can contain no characters (the null string) or a sequence of up to 255 letters, numbers, or special characters. *Mr. Smith*, *ID# 1234*, and *!@#$%&** are all examples of the data type string.

BOOLEAN

A quantity that can only have a value of true or false is said to be of type Boolean. Boolean types are primarily used for decision making in a program.

Table 4.1 summarizes the types of data values and their properties.

4.3 DECLARATION STATEMENTS

If the value of a particular quantity used in a program does not change while the program is being executed, it is termed a *constant*. For example, a discount rate may be 12 percent, for all items purchased. Alternately, if a particular quantity may have many values during the execution of a program, it is termed a *variable*.

For example, although the discount in the previous example is always 12 percent, the price of a purchased item may be $13.00 or $7.99. Therefore, the discount rate is considered to be a constant, and the price of the item is considered a variable.

In order to reference constant or variable quantities in a program, you must assign them names or identifiers. The identifiers chosen should indicate what those identifiers represent; for example, `discount` could represent the constant value of a discount of 12 percent, and `cost` could represent the variable cost of a purchased item.

In Macintosh Pascal, identifiers must begin with a letter and may be followed by up to 254 characters that include other letters, numbers, or underscore symbols (_), all of which are significant in distinguishing one identifier from another. The letters may be entered as uppercase or lowercase. The system treats all letters identically, regardless of whether they are uppercase or lowercase letters. For example, Macintosh Pascal considers `cost`, `COST`, and `Cost` as the same identifier. The only other restriction in forming an identifier is that it cannot have the same spelling as any member in a set of reserved words that have specific meanings in Pascal. A list of these reserved words appears in Appendix D, at the end of this book.

Values for the constant and variable data types previously discussed require different-sized storage locations. A Boolean value requires much less storage than an extended-precision real. For the efficient management of storage in the Macintosh system, the amount of storage required, and therefore the data type, for each identifier must be known before that identifier can be used in a program.

Declaration statements are used to define constant and variable identifiers by declaring what value or which data type each is to represent in a program. These statements permit the proper amount of memory to be allocated for each identifier.

CONSTANT DECLARATIONS

Identifiers that are used to represent constant quantities are defined in a constant declaration. The format for a constant declaration follows:

FORMAT: **const**

```
<identifier> = <constant>;
<identifier> = <constant>;
        . . .
```

The reserved word **const** is followed by an identifier, the equal sign (=), the constant value assigned to that identifier, and a semicolon (;) to end the statement. If additional constants are to be declared, they can follow the first one using the same format.

The angle brackets, < and >, are not to be included in the identifier name or the constant value. They are used to help describe the quantities that make up each Pascal statement. Thus, <identifier> represents the position where a legal identifier is placed. This notation is continued throughout the text.

You need not press the Return key after each statement is entered or worry about making the program appear in an orderly format. One novel feature of Macintosh Pascal is that it automatically lines up statements as portions of the program are being entered.

The constant value assigned to an identifier determines the type of the identifier.

Example:

```
const
 wn = -34;
 word = 'test';
 tf = True;
 num = 3.4e-6;
```

The identifier wn is assigned the value −34 and is therefore considered an integer. Likewise, `word` is a string having the value `test`; `tf`, a Boolean with a value `True`; and `num`, a real having a value `3.4e-6`.

Once an identifier has been declared as a constant, its value cannot be altered. Any attempt to do so causes an error message to appear on the screen (see Figure 4.1).

FIGURE 4.1

The semicolon serves a very important function in Pascal. It is used to separate individual Pascal statements in a program list. When a character is used to separate items in a list, it is called a *delimiter.* The semicolon is the delimiter for Pascal statements and is used to clarify the meaning of a Pascal program. In the case of this declaration section, the semicolon is used to separate individual constant declarations, and the semicolon after the last declaration separates the constant declaration portion of the program from the portion of the program that immediately follows it.

VARIABLE DECLARATIONS

Identifiers that are used to represent variable quantities are defined in a variable declaration section. The format for this Pascal section follows:

```
FORMAT: var
        <identifier list> : <type>;
        <identifier list> : <type>;
                    . . .
```

The reserved word **var** is followed by an identifier or list of identifiers separated by commas, the colon (:), the type assigned to the identifier(s), and a semicolon

TABLE 4.2

TYPE	*ABBREVIATION*
Integer	**integer**
Long integer	**longint**
Real	**real**
Double-precision real	**double**
Extended-precision real	**extended**
Character	**char**
String	**string**
Boolean	**boolean**

(;) to end the statement. If additional identifiers are to be declared, they can follow the first one, using the same format.

The types available for each identifier were discussed in Section 4.2. The reserved word abbreviations indicative of each type are listed in Table 4.2.

Example:

```
var
 money : real;
 count, year : integer;
 test : boolean;
 letter : char;
 phrase, word : string;
```

The data type for `money` is real and has a value that must fall within the legal range for real numbers. If a value outside this range is used, an error message results. In a similar fashion, `count` and `year` are to be used for integer values; `test`, for Boolean values; `letter`, for single characters; and `phrase` and `word`, for strings of characters.

One statement can be used to declare one or several identifiers as a specific type. In this example (page 65), both `count` and `year` are declared to be of type integer in the same Pascal statement, and `phrase` and `word` are declared to be of type string in a single Pascal statement. For purposes of clarity, you may wish to use one declaration statement for each identifier, although this is not necessary.

The semicolon again serves to separate the declaration statement of one identifier from the next. The semicolon following the last statement separates the variable declaration portion of the program from the portion that succeeds it.

TYPE DECLARATIONS

The variable declaration statement is used to associate an identifier with one of the generic data types presented in Section 4.2. MacPascal also allows you to create your own data types. Creating types may include renaming an existing data type or creating a type that is a subrange of an existing type.

Unlike constant and variable declarations, no memory is allocated when a type is defined. Memory is allocated only when that defined type is used in the declaration of a variable.

FORMAT: **type**

```
        <type identifier> = <definition>;
        <type identifier> = <definition>;
                     . . .
```

The reserved word **type** is followed by a type identifier, the equal sign (=), the definition of that type, and a semicolon (;) to end the statement. If additional types are to be declared, they can follow the first one using the same format.

Example:

```
type
 float = real;
 range = 0..100;
```

The identifier `cost` is declared to be of type `float`. The **type** declaration indicates that `float` is equivalent to **real**, and `cost` can have any values in the legal range for real numbers.

```
var
 cost : float;
 number : range;
```

The identifier `number` is declared to be of type `range`. As indicated in the type declaration for `range`, `number` can have any integer value between 0 and 100, inclusive. The integer value is implied by the fact that the limits of the range (0 and 100) are both integers. The two dots separating 0 and 100 indicate that all integers between 0 and 100, including 0 and 100, are defined for `number`. A value for `number` outside these boundaries results in an error message that says **Assignment will cause identifier to be outside of subrange boundary.** The use of subranges in this manner is useful for detecting the entry of improper data.

It may be useful for you to think of a type as an outline for an identifier, and any identifier declared to be of a specific type must follow the outline defined for that type.

4.4 ESSENTIAL PROGRAM COMPONENTS

Every Pascal program must contain the following three parts: a program name statement, a word indicating the beginning of the program, and a word indicating the end of the program. The construction of a program is similar to the creation of a term paper. The paper must have a title, and there must be some indication of where the paper begins and where it ends.

PROGRAM NAMES

The first statement in a program identifies the program by name and is called the *program header line.* It has the following format:

FORMAT: **program** `<program identifier>;`

The program identifier following the reserved word **program** should differ from other identifiers in the program. Although no restrictions control the use of upper- and lowercase letters in a program, all program identifiers in this text begin with an uppercase letter. A semicolon ends this statement and separates it from the rest of the program.

Example:

`program Test;`	This statement gives the name `Test` to the program that follows.

BEGIN AND END

The body of the main program follows the program header line and is initiated by the reserved word **begin** and concludes with the reserved word **end** followed by a period (.). In addition to indicating the beginning and end of the main program, these words can be used to indicate the beginnings and ends of smaller portions of the main program, as paragraphs are used to separate sections in a term paper.

Figure 4.2 illustrates a minimal Pascal program. Although this program does not serve any function, it gives you an idea of the basic structure of a program.

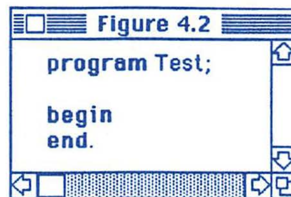


FIGURE 4.2

COMMENTS

Pascal allows you to make notes within a program. These notes or comments can be used to clarify the meaning of various parts of a program or to describe in English what is going on in Pascal. This method of annotation is similar to parenthetical remarks made in a term paper. These remarks do not change the contents of the paper, and may even be deleted from the paper without altering its general intent. They mainly serve to clarify the ideas already expressed. In a similar fashion, comments clarify, without changing, the intent of a Pascal program.

Comments are separated from the functional parts of a program by enclosing them within braces, { and }. Comments can also be written between (* and *) —parentheses paired with asterisks.

Examples:

`{ This is a comment }`	This comment is enclosed in braces.
`(* Also a comment *)`	This notation also serves to separate comments from the rest of the program.

When using parentheses and asterisks, make sure you do *not* leave a space between the (and * or the * and). If you do, an error will occur, because the system does not consider this to be a comment.

Comments may be placed anywhere in a program. They should, however, not interfere with the statements that actually do something. They are commonly used to describe constant and variable identifiers and to explain the purpose of a program or portions of a program.

PUTTING THE PARTS TOGETHER

Figure 4.3 shows the basic structure of a Pascal program with comments.

A program can be thought of as being divided into three sections. The first is the *program header section,* which includes the program header statement and any comments describing the purpose of the program.

The second section is the *declaration section,* in which all constants, types, and variables are defined. It is not necessary that all three of these declarations appear in a program, but the **const** declaration must precede the **type** declaration, and the **type** declaration must precede the **var** declaration.

The last section, the *processing section,* follows the declaration section of the program. It contains the Pascal statements that use the declarations to form the solution of the problem.

A discussion of the program header section and introductory material on the declaration section was presented earlier in this chapter. In Chapter 6, "Assignment Statements and Programming Aids," you will begin to find out how to fill in the processing section of a Pascal program.

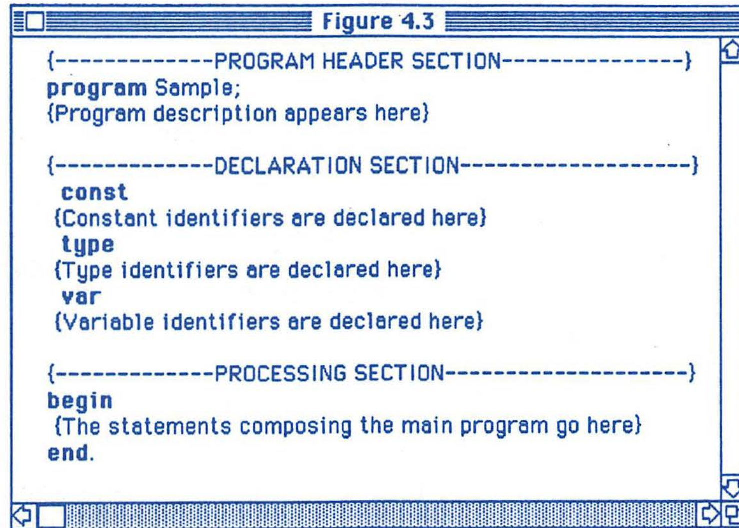


FIGURE 4.3

4.5 TYPICAL PROGRAMMING ERRORS

The following errors are typical of Pascal programs.

1. Insertion of a space between a parenthesis and an asterisk when used to enclose a comment statement.

   Example:
   ```
   ( * Comment *)
   ```

2. Omission of a period following **end** in the main program.

   Example:
   ```
   program Alpha;
   begin
   end
   ```

3. Omission of a semicolon ending statements.

   Example:
   ```
   program Beta
   ```

4. The placement of **const**, **type**, and **var** declarations in an order other than the one specified.

   Example:
   ```
   program Gamma;
   var ...
   const ...
   type ...
   ```

5. Omission of a comma separating identifiers in a **var** declaration list.

   Example:
   ```
   var
     x y : integer;
   ```
   where x and y are separate identifiers

6. The use of the same identifier in more than one declaration.

Example:
```
const
 max = 100;
var
 max : integer;
```

7. The use of a reserved word as an identifier.

Example:
```
const
 begin = 34.5;
```

8. The omission of **begin** or **end** to enclose a processing section.

Example:
```
program Delta;
end.
```

EXERCISES

1. Specify the Pascal data type for each of the following values. Strings and character data are enclosed within apostrophes. If no Pascal data type is available, say so.
   a. `true`
   b. `12345678`
   c. `1.76e5`
   d. `1.76e-400`
   e. `'words'`
   f. `'m'`
   g. `761`
   h. `-1.76e+300`
   i. `false`
   j. `'maybe'`
   k. `'false'`
   l. `7896.345`
   m. `'1234.56'`
   n. `1,234`
2. Write a declaration statement for each of the following identifiers and descriptions:
   a. `test` to have Boolean values
   b. `big` to have extended-precision real values
   c. `small` to have real values
   d. `pi` to have a value of 3.14159
   e. `color` to be of type string
   f. `x` and `y` to have integer values
   g. `w` to have string values
   h. `s` to have values of a single keyboard character
   i. `large` to have integer values, each 6 to 8 digits long
   j. `huge` to have double-precision values
   k. `dec` to have a type identical to real
   l. `answ` to have a value of 'y'
   m. `choice` to be a type with values yes and no

n. `tax` to have a value of 0.065
o. `logic` to have values of true and false

3. Classify each of the following Pascal statements as valid, if the statement is syntactically correct, or invalid. If the statement is invalid, explain why.
   a. `program First`
   b. `begin;`
   c. `theend.`
   d. `var x = integer;`
   e. `const c = 'string';`
   f. `type salary = (10000,...,30000);`
   g. `{ anything goes }`
   h. `(* anything goes * )`
   i. `begin (*'This is OK.'*) end.`
   j. `var r,s,t : real;`
   k. `type ditto = longint;`
4. Fill in the keywords or symbols in the outline of the Pascal program given in Figure 4.4.

Sample;

k = 0.65;

digit = 0..9;

i, j : digit;

body of main program

.

FIGURE 4.4

C H A P T E R 5

A First Step: Input and Output

5.1 INTRODUCTION

Constructing a program to solve a problem is like baking a cake from the directions in a cookbook. You know what kind of cake to make (the output), so you read the book to find out the ingredients needed to make the cake (the input). After the ingredients are assembled, you follow the directions in the cookbook to produce your culinary masterpiece.

The construction of programs follows a similar pattern. The results produced by the program can be displayed on the monitor screen, printed on paper, or sent to disk storage, creating a file of data. This type of data is called *output*. Data that are needed by the program for solving a problem can be entered into the computer's main memory through the keyboard or from a data file that has been previously stored on a disk. This type of data is called *input*.

In the problem-solving methodology discussed in Chapter 3, the IPO chart is divided into input and output sections. When the solution is translated into MacPascal code, the manner in which the solution is implemented determines (1) whether the output should be sent to the screen, the printer, or a data file and (2) whether the input is from the keyboard or a data file. When you become aware of all options for displaying and entering data, you then decide how to acquire data and display results. You are by no means restricted, because different methods of input and output can be used in the same program.

5.2 PROGRAM OUTPUT

We will begin by considering how program results can be output, because programs can be written to produce output without requiring input. Also, if a program doesn't produce some form of output, is there any reason for writing that program?

When entering Pascal, the screen displays three windows. One of these windows is the program window where instructions are entered and listed. The other two windows, the **Text** window and the **Drawing** window, are used for screen display of program output. The **Text** window is used to display textual output produced during a program run and the **Drawing** window is used to display textual or graphic output. If the **Text** or **Drawing** window is not present on the screen when a program is run, any screen output produced by that program is invisible. If screen output is expected, make sure the appropriate output windows are opened before a program is run.

TEXT SCREEN OUTPUT

Output to the screen is initiated when a specific instruction in the program window is executed. Program statements usually begin with a keyword that indicates a specified computer activity. The keyword is followed by the information required for the proper execution of the instruction. Although MacPascal permits keywords to be typed in any form, all uppercase, upper- and lowercase, or all lowercase, this book presents all keywords in lowercase, and in darker print when they appear in the text.

Many of the keywords in MacPascal can be used only as keywords and not for any other purpose. Therefore, they are considered *reserved words.* A list of these reserved words may be found in Appendix D, at the end of this book.

The keywords in the instructions that produce output in the **Text** window are **write** and **writeln.** The difference in the appearance of the output displayed by using these two keywords lies in the positioning of the printing cursor after the output instruction is executed. With the keyword **write**, the next printing position in the **Text** window is next to the last character output; with the keyword **writeln**, the next printing position is at the beginning of the next line in the **Text** window.

After we introduce a MacPascal keyword in this text, we present it in a format statement. This statement shows the general appearance for the instruction that is used with the keyword. We also discuss and demonstrate variations in this format. In general, all instructions in MacPascal end with a semicolon. An exception to this rule occurs when the instruction immediately precedes an **end**; in that case, the instruction need not have a semicolon.

Messages can be printed in the **Text** window using either keyword. The correct instruction formats for the use of the **write** and **writeln** keywords for string messages are as follows:

FORMAT: **write('<message>');**

FORMAT: **writeln('<message>');**

The format statements show that the instruction must begin with the keyword **write** or **writeln** followed by parentheses. Within the parentheses, a string message enclosed in apostrophes may appear. The angle brackets that appear

in format statements (< and >) are used to enclose a description of what is to be placed there and are not used when the code is written.

Examples:

Statement	Result
`write('Success! ');`	`Success!` is displayed in the **Text** window, and the next printing position is adjacent to the exclamation point.
`writeln('Success! ');`	`Success!` is also displayed in the **Text** window, but the next printing position is in the first position on the line below `Success!`

The following examples demonstrate the difference in appearance of the output produced in the **Text** window by the **write** and the **writeln** keyword instructions in a simple program.

Examples:

Statements	Result

`write('one');`
`write('two');`
`write('three');`
`write('four');`
	The output produced by these four statements appears on a single line, because the messages are included in **write** statements.
	Output:
	`onetwothreefour`

`writeln('one');`
`writeln('two');`
`writeln('three');`
`writeln('four');`
	The output produced by these four statements appears on separate lines, because the messages are included in **writeln** statements.
	Output:
	`one`
	`two`
	`three`
	`four`

The **write** and **writeln** instructions can be used in the same program to organize output in different ways.

Example:

`write('one');`
`writeln('two');`
`write('three');`
`writeln('four');`
	Output following a **write** statement appears on the same line as the output from the **write** statement. Output following a **writeln** statement appears on the following line.
	Output:
	`onetwo`
	`threefour`

If an apostrophe is to be contained in an output message, two consecutive apostrophes must be entered in the message of the **write** or **writeln** statement.

Example:

`writeln('It''s a boy!');`	This statement produces the output `It's a boy!` The two apostrophes appear as one.

Output to screen instructions can be used to produce more than a single literal message. Several literal messages can be combined in the same instruction by enclosing each within apostrophes and separating them by commas. The commas form output fields in the instruction.

Example:

`writeln('one','two','three');`	The messages appear on the same line with no spaces separating them. Output: `onetwothree`

It is usually desirable to separate literal messages that are contained in the same output instruction using field width descriptors. This positive integer describes the number of spaces allocated for displaying the output item it follows.

FORMAT: **write('<message>' : <field width>);**

FORMAT: **writeln('<message>' : <field width>);**

The field width number is separated from the message by a colon. The output is then right-justified in the described field.

Example:

```
writeln('1234567890123456');
writeln('one':10, 'two':6);
```

The first message appears in a field of 10 spaces and the second one in a field of 6 spaces. A line of digits is printed above the two messages so that you can readily see the field spacing.

Output:

```
1234567890123456
       one   two
```

If no field width descriptors are given, output from each field abuts the output produced in the previous field.

In addition to messages, output to screen instructions can display numeric data values.

FORMAT: **write(<data value> : <field width>);**

FORMAT: **writeln(<data value> : <field width>);**

Field widths can also be specified for data values.

Example:

```
writeln('12345678901234567890');
writeln(-7777:10, 1111:6);
```

Two integers are displayed with an extra space in each integer field reserved for a sign.

Output:

```
12345678901234567890
     -7777  1111
```

If the numeric values to be output are real (have a decimal portion) and no field width descriptors are given, the values are automatically displayed in e-notation, left-justified in a default field width of 10 spaces, and rounded to the nearest tenth.

Example:

```
writeln('12345678901234567890');
writeln(-77.77, 1.111);
```

Output:

```
12345678901234567890
-7.8e+1    1.1e+0
```

Using two descriptors enables you to output real numeric data without e-notation in a field of a certain width and to specify the number of decimal places the result is to contain.

FORMAT: **write(<data value> : <field width> : <decimal places>);**

FORMAT: **writeln(<data value> : <field width> : <decimal places>);**

The first descriptor still determines the width of the space in which the output is displayed, and the second descriptor specifies the number of decimal places. Neither of these descriptors is required. They should be used to create orderly output.

Example:

```
writeln('12345678901234567890');
writeln(-77.77:10:1, 1.111:10:2);
```

The real value of −77.77 is output in 10 spaces and rounded to one decimal place, and the real value of 1.111 is output in 10 spaces but rounded to two decimal places.

Output:

```
12345678901234567890
-77.8          1.11
```

To copy data from memory to the screen, the output instruction must refer to the data by using the identifiers defined in the program declaration section. When the instruction contains identifiers rather than values, the computer displays the values that are stored in memory locations represented by those identifiers.

FORMAT:
write(<identifier> : <field width> : <decimal places>);

FORMAT:
writeln(<identifier> : <field width> : <decimal places>);

The next example shows the output from instructions that contain literal messages as well as identifiers.

Example:

```
const
 max = 99;
 min = 0;
begin
 writeln('Max:  ',max:3);
 writeln('Min:  ',min:3);
```

The values for `max` and `min` are right-justified in fields.

Output:

```
Max:  99
Min:   0
```

If you want blank lines between lines of output, the **writeln** keyword can be used as a complete instruction.

Example:

`writeln('line one');`
`writeln;`
`writeln('line two');`
	These statements display line one, skip a line and display line two as shown below:
	`line one`
	
	`line two`

Before sending output to the **Text** window, you may want to clear it of any previous contents. The keyword that accomplishes this clearing is **page**.

FORMAT: **page;**

When this statement is executed, the **Text** window is erased so that any output to the window following this statement appears on a clean window.

DRAWING SCREEN OUTPUT

Items sent to the **Text** window are displayed line by line, in much the same manner as output on a typewriter. Items in this window cannot be made to appear on a previous line once the position of the printing cursor has passed that line.

MacPascal offers you complete control of any output to the screen. This control is necessary when documenting graphic output in the **Drawing** window, because descriptions must be placed in positions that do not interfere with the drawing being displayed.

Each position in the **Drawing** window is identified by two integers. The first one indicates how far a specific position is from the left side of the window, and the second number indicates how far that position is from the top of the window. Each position on the **Drawing** window thus has a unique pair of values that identifies the position's location. The two values are separated by a comma and are always interpreted in the manner just described.

The "origin" or 0,0 position is located in the upper left corner of the window. When the MacPascal system is booted, horizontal position values range from 0, on the left, to approximately 200, on the right. Vertical position values range from 0, at the top of the window, to approximately 200 at the bottom. This is not the orientation commonly used for graphing mathematical functions. Figure 5.1 shows this initial grid and the several locations in the window.

Textual output can be assigned to a point on this grid by specifying the position where the output is to appear in the window. The keyword that accomplishes this is **moveto**.

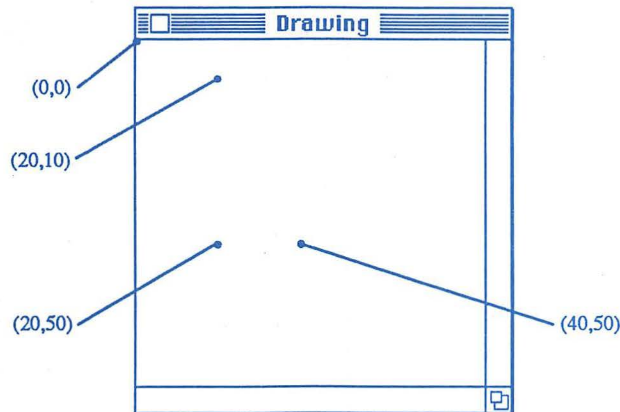


FIGURE 5.1

FORMAT:
moveto(<horizontal position> ,<vertical position>);

Remember that the first value represents the horizontal distance and the second the vertical distance, and both values must be within the limits of the grid in order to be visible.

Example:

`moveto(80, 100);`	This statement moves the printing position 80 units from the left side of the **Drawing** window and 100 units from the top of the window.

As with all windows, the **Drawing** window can be moved anywhere on the screen and enlarged to accompany any size output. When it is enlarged, the maximum values that produce visible output in the window are increased.

To direct output to the **Drawing** window instead of the **Text** window, the keyword **writedraw** is used. As with the **write** and **writeln** instructions, the **writedraw** instruction outputs literal messages as well as actual numeric data and data from memory. The format for this instruction follows.

FORMAT: writedraw(<identifier or data list>);

Example:

`moveto(80, 80);` `writedraw('middle');`	The display position is determined by the **moveto** statement, and `middle` is shown at that position in the **Drawing** window. See Figure 5.2.

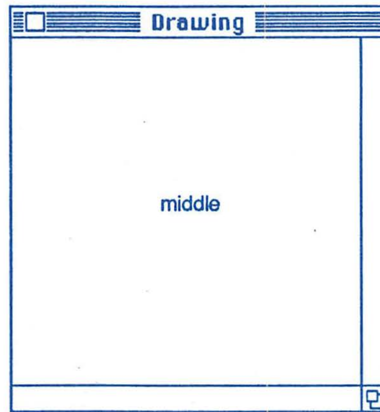


FIGURE 5.2

The MacPascal system has many graphic capabilities. This is only an introduction to the **Drawing** window. You can and will do a lot more than just print messages in that window. Soon you will be drawing lines, painting circles, rectangles, and other shapes, as well as combining literal and graphic output.

DATA FILE OUTPUT

If the program output is needed for use at a later time by another program, that output can be sent to a data file on a disk where it is retained after the computer is turned off. A file of this type is called an *external data file*. This section introduces you to files in which string data can be stored.

You can store information in a file (write to that file) or access data stored in a file (read from that file). Other file operations are discussed in Chapter 13.

To use a file, an identifier for the file must be declared in the **var** section of the program.

FORMAT: **var**
<file identifier> **: file of string;**

This statement defines <file identifier> to be a file that contains string data.

Example:

`var` `f : file of string;`	This statement defines `f` as the name of a file whose contents are strings.

Once a file identifier has been declared, the external file must be "connected" to the *internal file* that temporarily stores data in main memory.

FORMAT: **open(<file identifier>, '<file name>');**

The **open** statement sets up a channel of communication between the `<file identifier>` used in the program and `<file name>`, the name of the external file stored on the disk. If an external file already exists with the same file name, the original file is completely erased when this statement is executed.

Example:

`open(f, 'SciFi');`	This statement associates the internal file identifier `f` with the external file stored on the disk under the name `SciFi`.

The data stored in the associated internal and external files must be of the same type.

The keyword that directs output to a file of string is **write**.

FORMAT: **write(`<file identifier> , <data list>`);**

To indicate that information in `<data list>` is to be sent to an internal file identified by `<file identifier>`, that `<file identifier>` must be included in a **write** statement.

Example:

`write(f, 'Flash', 'Gordon');`	The values `Flash` and `Gordon` are stored in the internal text file identified by `f`.

When you have completed writing to an internal file, the file is "closed." This process automatically stores the contents of the internal file on a disk under the external file name associated with the program file identifier in the **open** statement. The channel that was established between them by the **open** statement is broken by the **close** statement.

FORMAT: **close(`<file identifier>`);**

Example:

`close(f);`	This statement breaks the connection between the internal file `f` and the external file `SciFi` after storing the contents of `f` in `SciFi`.

A program that contains the **open**, **write**, and **close** statements can create an internal data file, store data in that file, and make a copy of its contents in an external file.

Example:

```
open(g, 'PhoneFile');
write(g, 'Mary Q. Contrary');
write(g, '(123) 555–1212');
close(g);
```

An internal file of strings is created and identified by the file name g and associated with the external file PhoneFile. The name and phone number are stored in the internal file and transferred to the external file after **close** is executed.

PRINTER OUTPUT

You can also obtain a copy of the results of the execution of a program by sending that output to a printer. The resulting printout is called *hard copy* and is available for use when the computer is turned off.

In MacPascal, the printer is considered to be an external file of type **text**. Therefore, in order to send output to a printer, a file identifier of type **text** must be declared in the **var** section of the program. Rather than an **open** statement, the **rewrite** statement is used to associate an internal file identifier with the external file named **printer:**.

FORMAT: **rewrite(**<file identifier> ,**'printer:');**

The external file name **printer:** must be used as just specified. Once this link has been established, the output (<data list>) can be sent to the printer using a **writeln** instruction that includes <file identifier>.

FORMAT: **writeln(**<file identifier> , <data list>**);**

Example:

```
var
 f: text;
...
rewrite(f, 'printer:');
writeln(f, 'Happy birthday!');
```

If f is declared to be of type text, the **rewrite** and **writeln** statements in the main program send the message Happy birthday! to the printer.

You should make sure that the printer in your system is turned on before any program that requires printed output is executed.

In summary, results can be sent to the **Text** window, the **Drawing** window, a data file, or a printer by choosing the appropriate Pascal statements.

5.3 PROGRAM INPUT

Now that we have introduced the methods for creating output, it is time to find out how to obtain input. The two methods of gathering data are to input them from the keyboard and to input them from a data file.

KEYBOARD INPUT

As with output, there are two keywords for input: **read** and **readln**.

FORMAT: **read(<identifier>);**
readln(<identifier>);

The identifier must be declared in the **var** section of the program. Both instructions direct input from the keyboard into memory. This input is stored in the memory location represented by the named identifier. Although a list of identifiers may be used with each **read** or **readln** statement, it is good programming practice to enter one data value with each input statement. In this text, we follow this practice.

The **readln** instruction requires the pressing of the Return key after the data value for an identifier has been entered. The Return permits the program to proceed to the next instruction. The **read** instruction differs from the **readln** instruction in that the program continues when a space or a Return follows the entry of a value for its identifier.

The next example combines the **read** and **readln** statements in a program and demonstrates the difference between the two by accepting the values 4, 2, and 9 as input data.

Example:

```
readln(a);
read(b);
read(c);
writeln(a:10, b:10, c:10);
```

The **readln** instruction accepts the 4 and stores it under the identifier **a** after the Return is pressed. The program now proceeds to the next instruction, which is a **read** instruction. The **read** instruction accepts the 2 for **b** and then moves to the next instruction after a space is entered. The next **read** instruction accepts a 9 for **c**. After a space is entered to complete the entry of data, the **writeln** instruction is executed. The three data values are displayed on the same line as the 2 and the 9, because the Return key was not pressed after these values were entered.

Output:

```
4
2 9       4       2       9
```

Write and **readln** instructions are often paired when a request for data and the response appears on the same line in the **Text** window. You should always precede a **readln** statement with a **write** statement that prompts the user as to what data are expected.

Problem: Write a program that requests a person's name and age placing the responses on the same line as the request.

The IPO chart describing the identifiers is shown in Figure 5.3.

IPO CHART

Class	Identifier	Description
INPUT	name	name of person
	age	person's age
PROCESSING		

Class	Identifiers		Drawing
OUTPUT	name		
	age		

FIGURE 5.3

The outline of steps in the solution is given in Figure 5.4.

Personal Data
Request name
Enter name
Request age
Enter age
Display message

FIGURE 5.4

A list of the program and a sample run appear in Figure 5.5

```
program PersonalData;
{Enters and displays name and age within a message}

 var
  name     {name of person}
   : string;
  age      {age of person}
   : integer;

begin
 write('Enter your name:   ');
 readln(name);
 write('Enter your age:   ');
 readln(age);
 writeln('Is it true, ', name, ' that you are really ', age : 2, ' years old?')
end.
```

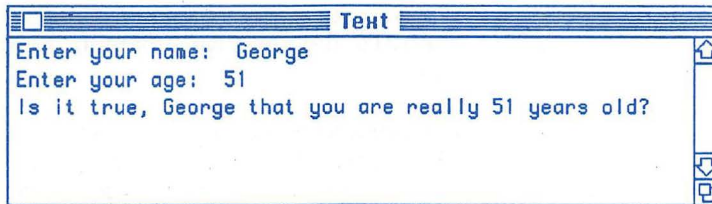


FIGURE 5.5

DATA FILE INPUT

Since we created an external data file in the preceding output discussion, we can now retrieve the information stored in that file. Again, an identifier for an internal file of the same type as the external file must be declared in the program before data can be accessed from the external file.

The keyword to open the external file on a disk and associate it with the internal file identifier declared in the program is **open**.

Once a file is opened, its data can be read and stored in main memory, using identifiers whose types match the types of data contained in the external file.

FORMAT: **read(`<file identifier>` , `<identifier>`);**

The **read** statement that includes an internal file identifier, <file identifier>, is used to access data from an external file and store those data in a location represented by <identifier> declared in the program.

Example:

`open(abc, 'FriendFile');` `read(abc, x);`	If `abc` has been declared to be a **file** of **string**, a data value is read from the external file `FriendFile` and is stored in the main memory location represented by the string identifier `x`.

As when writing to a file, a **close** statement should be used to disassociate an internal file from an external file.

In summary, data can be read into main memory locations from the keyboard or from an external data file.

5.4 SAMPLE PROGRAMS

Practical examples using the Pascal instructions presented in this chapter are shown in the sample programs that follow. The solutions illustrate the complete problem-solving process, including IPO and N-S charts. At this elementary stage, problems may not require an elaborate solution process, but the process steps are included to emphasize the importance of their use in the development of good programming skills.

Problem: Write a program to request the name and Social Security number of a person and to output both responses, appropriately documented, on the printer.

The IPO chart listing identifiers and their descriptions is shown in Figure 5.6. The file identifier, `pr`, is used to send output to the printer.

The N-S chart in Figure 5.7 shows the sequence of steps for the solution. After data have been entered into memory, those data are sent to the printer. This process of displaying data in two different output media is known as "echoing."

IPO CHART

Class	Identifier	Description
INPUT	name	name of person
	ssn	social security number
PROCESSING	pr	printer file

OUTPUT	Identifiers		Drawing
	name		
	ssn		

FIGURE 5.6

SocialSecurityInfo

Access printer
Request name
Enter name
Request social security number
Enter social security number
Display on printer

FIGURE 5.7

```
program SocialSecurityInfo;
{Prints name and social security number.}

 var
  name,            {name of person}
  ssn      {social security number}
   : string;
  pr      {internal printer file}
   : text;

begin
 rewrite(pr, 'printer:');
 write('Enter your name:  ');
 readln(name);
 write('Enter your social security number:  ');
 readln(ssn);
 writeln(pr, 'Name:  ', name : 20);
 writeln(pr, 'Social Security Number:  ', ssn : 11)
end.
```

```
Name:         Jonah Williams
Social Security Number:  123-45-6789
```

FIGURE 5.8

A complete listing and sample run of the solution are depicted in Figure 5.8.

The combination of the **write** and **readln** statements gives the output the correct appearance.

Problem: Write a program to output the message **Apples are delicious, especially Macintoshes!** in the **Text** window three times and skip a line between each occurrence.

The IPO chart for the solution is shown in Figure 5.9.

The N-S chart (see Figure 5.10) shows the order of the statements that produces the desired output.

IPO CHART

Class	Identifier	Description
INPUT	msg	message to be displayed
PROCESSING		

	Identifiers		Drawing
OUTPUT	msg		

FIGURE 5.9

Apple Message
Display message in Text window
Display a clear line
Display message in Text window
Display a clear line
Display message in Text window

FIGURE 5.10

```
program AppleMessage;
{Prints a message in the Text window}
{three times, separated by a blank line}

 const
  msg = 'Apples are delicious, especially Macintoshes!';

begin
 writeln(msg);
 writeln;
 writeln(msg);
 writeln;
 writeln(msg)
end.
```

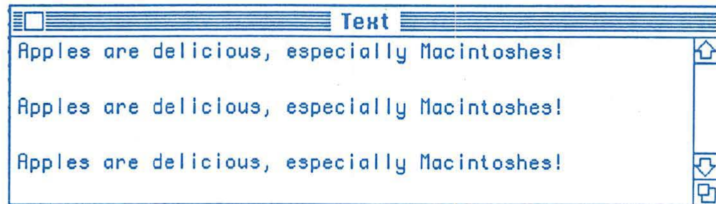


FIGURE 5.11

Figure 5.11 depicts a complete listing and run of the solution to the problem.

The use of the **writeln**; statement between the messages produces a clear line of output. You must make sure that the **Text** window is large enough to accommodate the message on one line.

Problem: Write a program to output the same message displayed in the same format as in the previous problem, but this time in the **Drawing** window. Begin 30 units from the left and have the first message displayed 20 units from the top of the window and succeeding messages 50 units below.

The IPO chart is shown in Figure 5.12.

The N-S chart is similar to the one in the previous problem, because the sequence of steps needed to satisfy the specifications of the problem is identical (see Figure 5.13).

IPO CHART

Class	Identifier	Description
INPUT	msg	message to be displayed
PROCESSING		

	Identifiers		Drawing
OUTPUT			display message

FIGURE 5.12

Apple Message In Drawing Window
Move to beginning position in Drawing window
Display message
Move to next location
Display message
Move to next location
Display message

FIGURE 5.13

```
program AppleMessageInDrawingWindow;
{Prints a message in the Drawing window}
{three times, separated by 50 units}

 const
  msg = 'Apples are delicious, especially Macintoshes!';

begin
 moveto(30, 20);
 writedraw(msg);
 moveto(30, 70);
 writedraw(msg);
 moveto(30, 120);
 writedraw(msg)
end.
```

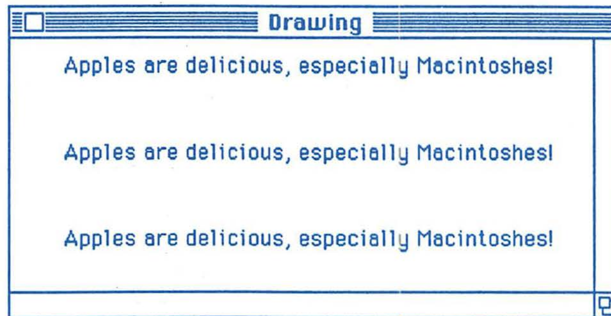


FIGURE 5.14

A complete listing and run of the solution to the problem are depicted in Figure 5.14.

Again, you may have to adjust the size of the **Drawing** window to accommodate the message display.

Problem: Write a program to output the same message in the same format as in the previous problems, but this time send output to the printer.

The IPO chart for the printed solution contains an extra identifier for the printer file (see Figure 5.15).

The N-S chart essentially mirrors those of the previous two problems (see Figure 5.16).

A complete listing and run of the problem solution are given in Figure 5.17.

IPO CHART

Class	Identifier	Description
INPUT	msg	message to be displayed
PROCESSING	pr	printer file

Class	Identifiers		Drawing
OUTPUT	msg		

FIGURE 5.15

AppleMessageOnPrinter
Display message on printer
Skip a line
Display message on printer
Skip a line
Display message on printer

FIGURE 5.16

```
program AppleMessageOnPrinter;
{Prints a message at the printer}
{three times, separated by a blank line}

 const
  msg = 'Apples are delicious, especially Macintoshes!';
 var
  pr     {internal printer file}
   : text;

begin
 rewrite(pr, 'printer:');
 writeln(pr, msg);
 writeln(pr);
 writeln(pr, msg);
 writeln(pr);
 writeln(pr, msg)
end.
```

```
Apples are delicious, especially Macintoshes!

Apples are delicious, especially Macintoshes!

Apples are delicious, especially Macintoshes!
```

FIGURE 5.17

Problem: Write a program to create an external text file of words called `Eats`. Put the names of five of your favorite foods in that file. Close the file.

The IPO chart containing descriptions of the identifiers used in the solution is shown in Figure 5.18. The name of the internal text file, f, is included in the processing section of the IPO chart.

The N-S chart that outlines the solution is shown in Figure 5.19. It contains statements that indicate where in the solution process the file is opened and where it is closed.

The program listing and sample run are given in Figure 5.20.

IPO CHART

Class	Identifier	Description
INPUT	a	names of food
	b	
	c	
	d	
	e	
PROCESSING	f	internal text file name

Class	Identifiers	Drawing
OUTPUT	a	
	b	
	c	
	d	
	e	

FIGURE 5.18

Food File
Enter the names of your 5 favorite foods
Open a text file
Write 5 food names to file
Close file

FIGURE 5.19

```
program FoodFile;
{Creates a file of 5 favorite foods}

 var
  a, b, c, d, e      {names of foods}
   : string;
  f      {internal file name}
   : file of string;

begin
 write('Enter first food:  ');
 readln(a);
 write('Enter second food:  ');
 readln(b);
 write('Enter third food:  ');
 readln(c);
 write('Enter fourth food:  ');
 readln(d);
 write('Enter fifth food:  ');
 readln(e);
 open(f, 'Eats');
 write(f, a);
 write(f, b);
 write(f, c);
 write(f, d);
 write(f, e);
 close(f)
end.
```

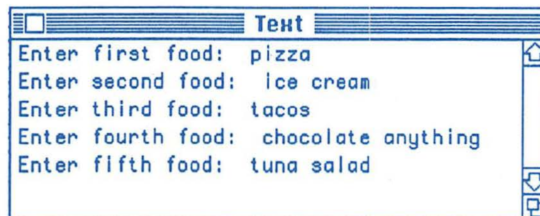


FIGURE 5.20

Problem: Write a program to open the file `Eats` created in the previous problem and read its data. Output the data in one column with each item in a 25-space field.

The IPO chart in Figure 5.21 is identical to the one shown in Figure 5.18.

The N-S chart shows the order in which the file operations are to be performed to satisfy the requirements in the problem statement (see Figure 5.22).

A complete listing and run of the program are shown in Figure 5.23.

IPO CHART

Class	Identifier	Description
INPUT	a	names of food
	b	
	c	
	d	
	e	
PROCESSING	f	internal text file

Class	Identifiers	Drawing
OUTPUT	a	
	b	
	c	
	d	
	e	

FIGURE 5.21

Read Food File

Open existing text file
Read 5 food names from file
Display names
Close file

FIGURE 5.22

```
program ReadFoodFile;
{Reads a file of 5 favorite foods}

 var
  a, b, c, d, e     {names of foods}
   : string;
  f     {internal file name}
   : file of string;

begin
 writeln('My five favorite foods are:');
 open(f, 'Eats');
 read(f, a);
 writeln(a : 25);
 read(f, b);
 writeln(b : 25);
 read(f, c);
 writeln(c : 25);
 read(f, d);
 writeln(d : 25);
 read(f, e);
 writeln(e : 25);
 close(f)
end.
```

FIGURE 5.23

5.5 TYPICAL PROGRAMMING ERRORS

The following list shows five typical programming errors:

1. Omitting the **;** at the end of a statement that does not precede an **end**.
   Example:
```
writeln('This is not the last one.')
writeln('This one is!')
end.
```
2. Opening a file and not closing it in the program.
3. Writing to or reading from a file without opening it.
4. Omitting the file identifier for output to the printer or to an external file.
5. Failing to define a file identifier in a declaration statement.

NONPROGRAMMING EXERCISES

1. Classify each of the following Pascal statements as valid or invalid. If the statement is invalid, explain why.
   a. `writeln('{This is right!});`
   b. `read(a);`
   c. `drawwrite('message');`
   d. `write(d : 8 : 2);`
   e. `writeln(g, 'filename')`
   f. `open(f, 'bananas');`
   g. `moveto('position');`
   h. `close(p,'end');`
   i. `writeln(g, 'printer:');`
   j. `rewrite(g, 'printer:');`
2. Describe the output of each of the following sets of program lines and indicate the next printing position:
   a.
```
writeln('Happy');
writeln('Birthday!');
```
   b.
```
writeln('Happy Birthday!');
```
   c.
```
writeln('Happy');
write('Birthday!');
```
   d.
```
write('Happy');
write('Birthday!');
```
   e.
```
write('Happy');
writeln('Birthday!');
```

Continued

3. Find the error(s) in each of the following short programs. If there are no errors, say so.

a.
```
program A;
  const
    bit = 'Tiny';
  var
    f : file of string;
begin
    open(f, 'Values');
    write(f, bit)
end.
```

b.
```
program B;
  var
    word : string;;
begin
  readln(word);
  writeln(word)
  writeln(f,word)
end
```

c.
```
program C;
  const
    extra = 'copy';
begin
  moveto(57,123);
  draw(extra);
end.
```

4. Describe the output of the following programs.

a.
```
program TryIt;
  const
    a = 5;
    b = -45;
    c = -34.567;
begin
  writeln('a':10,'b':10,'c':10);
  writeln(a:10, b:10, c:10:2)
end.
```

b. Input:
```
Charles Dennis
125-39-1008
```

```
program Another;
  var
    name, ssoc : string;
    f : text;
begin
  rewrite(f,'printer:');
  write(f, 'Name : ');
  readln(name);
  writeln(f, name);
```

```
  write(f, 'Social Security Number:  ');
  readln(ssoc);
  writeln(f,ssoc)
end.
```

PROGRAMMING EXERCISES

(G) 5. Write a program to display your name in the middle of the **Text** window.

(G) 6. Write a program to display your name in the middle of the **Drawing** window.

(G) 7. Write a program to display the word "corner" in each corner of the **Drawing** window.

(B) 8. Write a program to display a company's name and mailing address in the upper left corner of the **Text** window.

(B) 9. Write a program to display a company's name and mailing address on the printer.

(B) 10. Write a program to display a company's name and mailing address centered at the top of the **Text** window.

(B) 11. Write a program to create a file that stores a company's name and mailing address.

(B) 12. Write a program to read the file created in Exercise 11 and display its contents in the **Text** window.

(B) 13. Write a single program to read the file created in Exercise 11 and display its contents in the **Text** window, the **Drawing** window, and on the printer.

(G) 14. Write a program to display the words *top, bottom, right,* and *left* on the top, bottom, right and left sides, respectively, of the **Drawing** window.

(B) 15. Write a program to accept personal information from a job applicant. The program should request the following data: last name, first name, date of birth, sex, citizenship. Output the information the way it may appear on an application form.

(B) 16. Write a program to address envelopes for the Annual Calendar Company. Input should include a person's name, street address, city, state, and zip code. The output should appear with the same format as is used on an envelope. Output to the printer.

(B) 17. Write a program to create an external text file for a customer name and the name of an item purchased. Open the file and output the information in a letter to the customer, describing the purchase.

(G) 18. Write a program to accept a person's name as input and output a brief paragraph congratulating the person as a winner in a sweepstakes contest. Output to the printer.

(G) 19. Write a program, using **writedraw** and **moveto** statements, to draw the following design in the **Drawing** window.

```
   *
  ***
 *****
*******
```

(G) 20. Write a program to draw the design in Exercise 19 in the **Text** window.

(G) 21. Write a program to output the design in Exercise 19 on the printer.

(G) 22. Write a program to locate the point (33,66) in the **Drawing** window, put a star (asterisk) at that location, and write the coordinates next to it.

(B) 23. Write a program that accepts a dental patient's name and the date of his or her last visit, and use these data to produce a reminder to that patient that it is time for another visit. Send output to the printer.

(B) 24. Write a program that creates a file consisting of a dental patient's name and the date of his or her last visit in the form *mm/dd/yy,* and access the data in this file to produce reminders to the patient in the file that it is time for another visit. Send output to the printer.

CHAPTER 6

Assignment Statements and Programming Aids

6.1 INTRODUCTION

In the last chapter we remarked that finding ingredients to bake a cake is similar to finding input for a program; the actual cake that results from combining these ingredients corresponds to the output produced by a program. But we left out a very important step: how to put the ingredients together to make the cake. Obviously, you can't just throw them together, using any quantities, plop them in a pan, and throw them in an oven for an arbitrary period of time. You might not get what you expected. The same is true with programming.

We must now learn how to take inputs and process them in the proper way in order to produce desired results. Assignment statements are used to process data and make calculations with numeric data.

6.2 PERFORMING CALCULATIONS

Many problem solutions require calculations to be performed on input data. During the problem-solving process, the middle section of the IPO chart lists formulas that are needed to obtain the output required. These formulas may involve adding one numeric value to another, multiplying a value stored in main memory by a constant or variable, or finding the difference between two values. All these calculations require the use of numeric operators that indicate the arithmetic operations to be performed.

NUMERIC OPERATORS

A numeric operator is a symbol used to represent the specific arithmetic operation to be performed on two data values. The data values are called *operands* and may be numeric constants or variables. For example, in *A + 5,* A and 5 are called *operands,* + is called the *operator,* and the operation is called *addition.*

TABLE 6.1 Macintosh Pascal Arithmetic Operators

OPERATION	*ARITHMETIC*	*MACPASCAL*
Addition	+	+
Subtraction	−	−
Multiplication	×	*
Real division	÷	/
Integer division	none	**div**
Modulo	none	**mod**

TABLE 6.2 Examples of Arithmetic and Pascal Operations

ARITHMETIC	*MACPASCAL*
$3 + 2 = 5$	`3 + 2 = 5`
$3 - 2 = 1$	`3 - 2 = 1`
$3 \times 2 = 6$	`3 * 2 = 6`
$3 \div 2 = 1.5$	`3 / 2 = 1.5`
none	3 **div** 2 = 1
none	9 **mod** 5 = 4
$3^4 = 81$	none

Numeric operators in MacPascal are similar to the familiar operators in arithmetic and algebra. See Table 6.1.

The addition, subtraction, multiplication, and real division operators behave exactly as they do in standard arithmetic. The MacPascal operator **div** divides integer values only and produces the quotient portion of the result. Any decimal portion that is obtained when the two integers are divided is lost. For example, when 11 is divided by 4, the quotient is 2 and the remainder is 3. Therefore, 11 **div** 4 yields 2.

The MacPascal operator **mod** also divides integer values only but produces the remainder portion of the result. For example, 11 **mod** 4 yields 3.

In arithmetic, the superscript 4 in 3^4 is an exponent. In MacPascal there is no exponentiation operator.

Consider the examples in Table 6.2.

NUMERIC EXPRESSIONS

Combinations of operators and operands are called *expressions*. Spaces may or may not appear between an operand and an operator. Some numeric expressions may contain several operators; for example, 4 + 6 / 2. The value of this expression depends on the order in which the operations are performed. If the addition is performed before the division, the result is 4 + 6 / 2 = 10 / 2 = 5. On the other hand, if the division is done before the addition, the result is 4 + 6 / 2 = 4 + 3 = 7. To avoid such ambiguity, MacPascal adopts the conventions described in Table 6.3 for the performance of operations.

TABLE 6.3 Order of Operations

PRECEDENCE	*OPERATOR*
HIGH	*, /, **div**, **mod**
LOW	+, -

Operations with higher precedence are performed before those with lower precedence, so the example of 4 + 6 / 2 given before is evaluated in MacPascal as 7, because division has a higher precedence than addition. When an expression contains many operators, the evaluation is performed from left to right for all those operations that have the same precedence.

Sometimes, the established convention is not satisfactory. In the example 4 + 6 / 2, you may want the addition to be performed before the division operation. You can then override the conventional order given in Table 6.3 by using parentheses. For example, (4+6)/2 dictates to the computer that the operation within the parentheses should be performed first. When more than one operation is contained within a set of parentheses, they are performed according to the precedence described in Table 6.3.

In expressions that require complex calculations, more than one set of parentheses may be used within the expression, and sets of parentheses may be contained within each other. Parentheses of this type are called *nested parentheses*. In the case of nested parentheses, operations are performed from the innermost set of parentheses to the outermost. Consider the following example.

Example:

```
(2+4*(8-3) - 5)/(10 div 4)
    (2+4*5 - 5)/(10 div 4)  inner parentheses: subtract
     (2+20 - 5)/(10 div 4)  left parentheses:  multiply
       (22 - 5)/(10 div 4)                     add
            17/(10 div 4)                      subtract
            17/2            right parentheses: integer division
            8.5                                real division
```

MacPascal and standard algebra differ in substantial ways which you must recognize and treat accordingly. In algebra, whenever two variable names are typed side by side it is assumed that they are to be multiplied. No such assumption is made in MacPascal. A new variable identifier is formed when the two variable names are placed next to each other, which may cause a program error if the new identifier has not been declared in the program. For example, in algebra the expression *abc* implies multiplication of the value called *a* by the value called *b* by the value called *c*. In MacPascal **abc** does not represent an expression, but rather represents the name of an identifier.

Also, since expressions in MacPascal are to be typed on one line, it is not possible to express fractions as is done in standard algebraic form. The algebraic expression $\frac{2R}{3}$ would appear in MacPascal as 2**R*/3.

TABLE 6.4 Examples of the Difference Between Algebraic and Pascal Expressions

ALGEBRAIC	MACPASCAL	COMMENTS
$3b - 3c$	`3*b - 3*c`	No implied multiplication allowed in MacPascal; must be indicated by asterisk.
$\frac{n-7}{6.5+m}$	`(n-7)/(6.5+m)`	When coding fractions, it is a good practice to enclose both the numerator and denominator in parentheses.
x^3	`x * x * x`	No exponentiation operator available; multiplication can be used whenever practical.

These differences and others are highlighted throughout the text. Some examples are presented in Table 6.4.

In expressions that contain real and integer values, the real mode dominates; that is, the numeric result produced by the expression is real. For example, `3 * 2.1` results in the real value `6.3`, and `2.1 + 3` gives the real value `5.1`. Even two integers that are used as operands of a real operator produce a real value. For example, `10/5` yields `2.0`. However, any real operand in an expression with an integer operator (**mod** or **div**) produces an error.

ASSIGNMENT STATEMENTS

In MacPascal, the assignment statement is used to take the result of an expression involving data and store that result in a specified location.

FORMAT: `<identifier> := <numeric expression>;`

The colon and the equal sign must abut each other; spaces are optional on either side of them. The instruction directs the computer to evaluate the expression appearing on the right side of the `:=` symbol (often referred to as the *assignment operator*) and store the results in the memory location indicated by the identifier named on the left side of the `:=` sign.

Examples:

`x := 3;`	The value 3 is stored in the memory location represented by the identifier **x**.
`y := a * b + 4.5`	The expression to the right of the assignment operator is evaluated (**a** is multiplied by **b** and the product added to 4.5) and the results stored in the location represented by the identifier **y**.

Although MacPascal employs the := symbol in an assignment statement, it is easier to think of this symbol as a reverse arrow; that is, **x** ← 3, which implies that the value 3 is stored in or "goes to" the location represented by the identifier `x`.

In most cases, you must make sure that the data types of the identifier and the result of the numeric expression match. If the identifier is declared to be of integer type and the result of the numeric expression is real, an error occurs. However, if the identifier is declared to be of type real and the result of the numeric expression is an integer, the integer is stored as a real value with zero decimal portion. Needless to add, if the identifier is declared to be of numeric type and the expression is of type **string**, **char**, or **Boolean**, an error occurs. MacPascal is sensitive to this type of error, and if the error is made an appropriate error message appears. The program displayed in Figure 6.1 contains an assignment statement that stores a real value in an integer identifier. The error message that occurs is also shown.

Another important distinction should be made here between algebra and MacPascal. An assignment statement such as `count := count + 1` is nonsense in algebra, since such an equation leads to the conclusion that 0 = 1. However, this is a valid MacPascal statement and is interpreted in the following manner: The expression to the right of the assignment operator is evaluated first. The computer searches for the value stored in memory in the location designated by the identifier `count`, adds 1 to that value, and then stores the results in the location designated by `count`. If the value 3 was stored in `count`

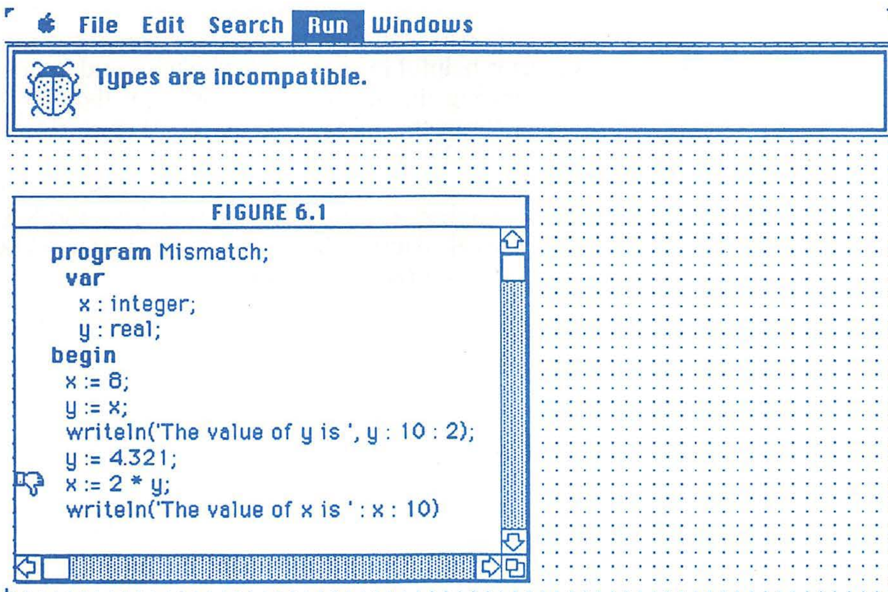


FIGURE 6.1

before this instruction, the value stored in `count` after this instruction is executed is 4. The prior value of the location identified as `count` is lost, because it is possible to store only one data value in one memory location at any one time.

6.3 SAMPLE PROGRAMS

Sample programs are shown in this section to give practical examples of the Pascal instructions we have presented. The solutions illustrate the complete problem-solving process, including the input of data, the performance of calculations, and the display of results.

Problem: Write a program to input a bowler's name and scores for three games. Compute and output the average score.

The IPO chart listing the descriptions of identifiers used in the solution is shown in Figure 6.2. The input values are also listed in the output section of the IPO chart, because these values appear in the **Text** window as output. It is not necessary to use additional **write** or **writeln** statements to display these values.

The techniques used to solve this problem are very similar to ones used for Example 2 in Chapter 3. In that problem, you were asked to find the average high temperature for a three-day period. In this problem, the average of three bowling scores is sought. The N-S chart blueprinting the solution is given in Figure 6.3.

The IPO and N-S charts can now be used to create a complete Pascal program. The IPO chart is helpful in creating declarations and in documenting the program, while following the sequencing of steps in the N-S chart helps in creating the statements in the main program. See Figure 6.4 for a complete program listing and sample run.

You should compare the N-S chart and the final Pascal solution to notice their similarities and see how the program can flow from the N-S chart.

The results of the calculations should be checked by hand before you can be relatively sure of the accuracy of your program.

IPO CHART

Class	Identifier	Description
INPUT	name	name of bowler
	score1	score on game #1
	score2	score on game #2
	score3	score on game #3
PROCESSING	average	average game score

OUTPUT	Identifiers		Drawing
	name	average	
	score1		
	score2		
	score3		

FIGURE 6.2

Bowling Average
Enter bowler's name
Enter score on game #1
Enter score on game #2
Enter score on game #3
Find average score
Display average score

FIGURE 6.3

```
program BowlingAverage;
{Determine the average bowling score for three games}

 var
  score1, score2, score3   {scores for 3 games}
   : integer;
  average    { average score for 3 games }
   : real;
  name      {name of bowler}
   : string;

begin
 write('Enter bowler'' s name:  ');
 readln(name);
 write('Enter score in game #1:  ');
 readln(score1);
 write('Enter score in game #2:  ');
 readln(score2);
 write('Enter score in game #3:  ');
 readln(score3);
 average := (score1 + score2 + score3) / 3;
 writeln;
 writeln('Average score for three games:  ', average : 6 : 2)
end.
```

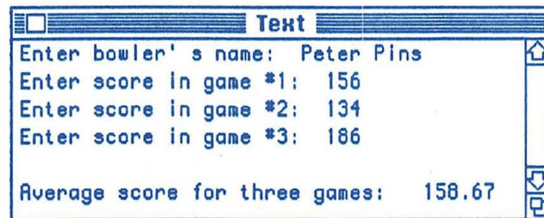


FIGURE 6.4

Problem: Write a program that requests the name of a car, the gallon capacity of its gas tank, and the estimated miles-per-gallon (mpg) rating for that car. Calculate and output the distance the car can travel on one tankful of gas.

The IPO chart for the solution is shown in Figure 6.5.

The distance that the car can travel is the product of the miles-per-gallon rating and the capacity of the tank. The N-S chart shown in Figure 6.6 uses this calculation in the outline of the solution to the problem.

IPO CHART

Class	Identifier	Description
INPUT	name	name of car
	mpg	estimated miles per gallon
	gastank	capacity of gas tank
PROCESSING	distance	distance traveled on one tankful

OUTPUT	Identifiers		Drawing
	name	distance	
	mpg		
	gastank		

FIGURE 6.5

Tankful

Enter make of car
Enter mpg rating
Enter tank capacity
Find distance traveled on one tankful
Display distance

FIGURE 6.6

```
program Tankful;
{Determination of the distance that a car can travel on }
{one tankful of gas.}

 var
  name            {make of car}
   : string;
  mpg,                {miles-per-gallon rating of car}
  gastank,          {capacity of gas tank}
  distance        {distance travelled using one tank of gas }
   : real;

begin
 write('Enter name of make of car:  ');
 readln(name);
 write('Enter estimate miles-per-gallon rating:  ');
 readln(mpg);
 write('Enter capacity of gas tank:  ');
 readln(gastank);
 distance := mpg * gastank;
 write('A ', name, ' can travel ');
 writeln(distance : 5 : 1, ' miles on one tank of gas.')
end.
```

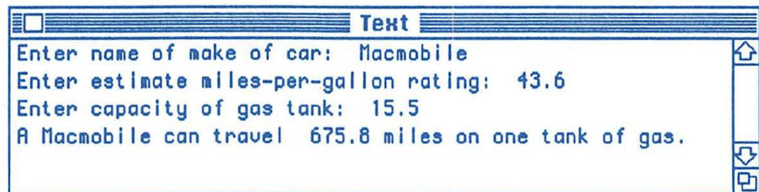


FIGURE 6.7

A complete listing and sample run for the solution are given in Figure 6.7.

Problem: Write a program that accepts the number of balloons purchased, the cost per balloon, the cost of the helium, and the selling price of the inflated balloon. Calculate and output the profit earned if all the inflated balloons are sold.

Figure 6.8 shows the IPO chart for the identifiers in the solution.

The profit is determined by first calculating the total cost. The total cost is the sum of the cost of the helium and that of the balloons (number of balloons multiplied by the cost per balloon). This total cost is then subtracted from the income from the sale of the balloons (number of balloons multiplied by the selling price per balloon) to yield the profit. The N-S chart in Figure 6.9 outlines this process.

IPO CHART

Class	Identifier	Description
INPUT	numball	number of balloons purchased
	unitcost	cost per balloon
	helcost	cost of helium
	sellprice	selling price of each balloon
PROCESSING	profit	profit from sale of balloons
	totcost	total cost of materials

	Identifiers		Drawing
OUTPUT	numball	profit	
	unitcost	cost	
	helcost		
	sellprice		

FIGURE 6.8

Balloon Sales
Enter number of balloons purchased
Enter cost per balloon
Enter price of helium
Enter selling price of a balloon
Determine total cost
Determine profit
Display profit

FIGURE 6.9

```
program BalloonSales;
{Calculation of a balloon vendor's profit}

 var
  numball       {number of balloons purchased by vendor}
   : integer;
  unitcost,        {cost of each balloon}
  helcost,          {cost of helium}
  profit,            {profit from sale of balloons}
  sellprice,       {selling price of each balloon}
  totcost        {total cost of materials}
   : real;

begin
 write('Enter the number of balloons purchased : ');
 readln(numball);
 write('Enter the cost per balloon:  $');
 readln(unitcost);
 write('Enter the price of the helium:  $');
 readln(helcost);
 write('Enter the selling price of the balloons:  $');
 readln(sellprice);
 totcost := numball * unitcost + helcost;
 profit := numball * sellprice - totcost;
 writeln('The total profit from balloon sales is $', profit : 7 : 2)
end.
```

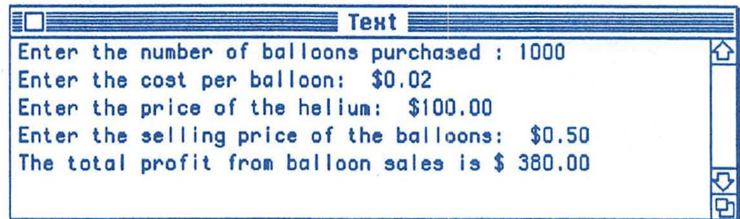


FIGURE 6.10

Figure 6.10 shows a complete listing and sample output for the problem.

Problem: Write a program to determine the net pay for an employee after deducting 25 percent for federal taxes and 6.5 percent for Social Security from the employee's gross pay.

Figure 6.11 shows an IPO chart for the solution.

The federal and Social Security taxes are products of the respective tax rates and the gross pay. The total of these deductions is then subtracted from the

gross pay to give the net pay. This solution is similar to Example 1 in Chapter 3, because it too involves a deduction of a percentage of a total from the total. The N-S chart in Figure 6.12 shows a blueprint of the solution.

Figure 6.13 provides a complete listing of the program and a sample run.

IPO CHART

Class	Identifier	Description
INPUT	fedrate	federal tax rate
	ssrate	social security rate
	grosspay	pay before taxes
PROCESSING	fedtax	amount of federal tax
	sstax	amount of social security
	netpay	pay after deductions

	Identifiers		Drawing
OUTPUT	fedrate	fedtax	
	ssrate	sstax	
	grosspay	netpay	

FIGURE 6.11

Net Pay
Enter employee name
Enter gross pay
Determine federal tax deduction
Determine social security deduction
Determine netpay
Display results

FIGURE 6.12

```
program NetPay;
{Determine the netpay for an employee after deducting 25% for}
{federal taxes and 6.5% for social security from the gross pay.}

 const
  fedrate = 0.25;        {federal tax rate}
  ssrate = 0.065;        {social security rate}
 var
  grosspay,                  {pay before taxes}
  fedtax,                      {amount of federal tax deducted}
  sstax,                       {amount for social security deducted}
  netpay                   {pay after taxes}
   : real;
  name                     {name of employee}
   : string;

begin
 write('Enter employee'' s name : ');
 readln(name);
 write('Enter employee''s pay before taxes:  $');
 readln(grosspay);
 fedtax := fedrate * grosspay;
 sstax := ssrate * grosspay;
 netpay := grosspay - fedtax - sstax;
 writeln;
 writeln('Deductions :  ');
 writeln('   Federal tax:  $', fedtax : 7 : 2);
 writeln('   Social security:  $', sstax : 6 : 2);
 writeln;
 writeln('Pay after taxes:  $', netpay : 7 : 2)
end.
```

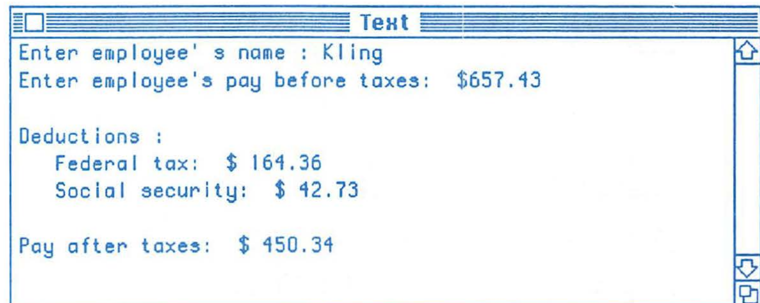


FIGURE 6.13

6.4 PROGRAMMING UTILITIES

Now that we have written programs that accept input, manipulate data, and provide output, we should investigate some of the programming aids that are unique to the MacPascal system. A program often contains errors that are detected on its first run. The system can locate and identify syntactical errors; logical errors are more difficult to locate and correct.

The **Run** and **Windows** menus provide many of the utilities valuable in finding logical errors. These menus are shown in Figure 6.14.

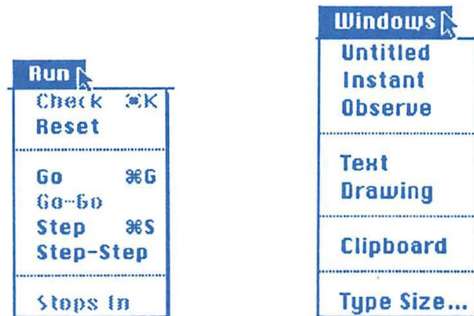


FIGURE 6.14

THE **OBSERVE** WINDOW

The **Observe** window, located on the **Windows** menu, can be used to trace the contents of memory locations at various points during the program execution. Often a change in the value of an identifier at a particular time during a program run can signal the location of a logical error. Once you see the point at which the error occurs, you should find and correct the error in the program's logic.

The following step-by-step exercise is designed to help you become familiar with this programming aid and enable you to use it in a programming and debugging session.

In Figure 6.15, the program `WatchIt` appears in the programming window. We are going to walk through this program and observe the contents of the memory location identified as `a`.

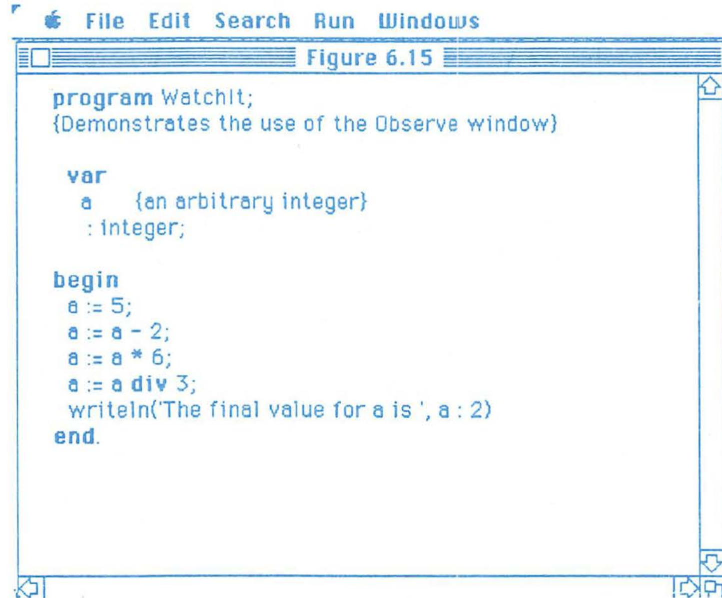
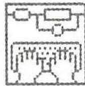


FIGURE 6.15



HANDS-ON EXERCISE

WALKING THROUGH A PROGRAM

Copy the program `WatchIt` into the program window.

Select **Observe** from the **Windows** menu. A window entitled **Observe** appears on the screen. It is divided into several rows and two columns. On the first row in the first column, the directive **Enter an expression** appears. The cursor is flashing in the second column of the same row. See Figure 6.16.

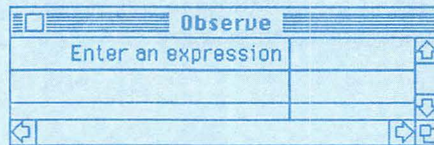


FIGURE 6.16.

Enter the expression `a` and press Return. Recall that an expression may be an identifier or any expression involving an identifier. For example, the expression `a+10` could also be entered.

continued

The **Enter an expression** request moves to the second row of the window. At this time another identifier or expression may be entered. See Figure 6.17.

FIGURE 6.17

After each expression is entered, the cursor moves down to the next line in the **Observe** window. Each line that is completed by you represents a request to observe the activity of a specific identifier or an expression containing an identifier. As with any window, the **Observe** window can be made larger or smaller, or moved.

STEP

Select the **Step** option in the **Run** menu. A pointing-finger icon points to **begin**, and the value 0 appears in the first column next to **a** in the **Observe** window. This means that the value in the location identified by **a** is 0. See Figure 6.18.

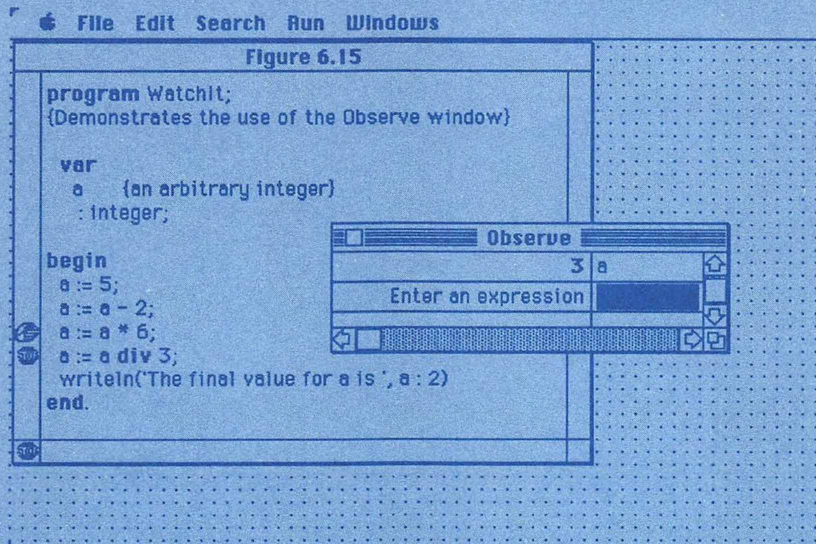


FIGURE 6.18

continued

When the **Step** option is selected again, the line pointed to by the finger icon is executed, and that icon moves to the next line in the program. Therefore, the value displayed in the **Observe** window represents the value of an expression before execution of the program line where the icon is presently located.

Select the **Step** option repeatedly until the program is completed. You should see the values 0, 5, 3, 18, and finally 6 appear next to **a** in the **Observe** window.

STEP-STEP

The **Step-Step** option in the **Run** menu has the same effect as the repeated selection of **Step.** Select **Step-Step**. The finger icon moves quickly down the program lines. You have to watch the **Observe** window very carefully to see all the values that appear.

Select **Go** from the **Run** menu and notice that only a brief flash of activity appears in the **Observe** window while output is sent to the **Text** window.

The **Observe** window can also be used to follow the progress of one or more expressions at selected points in a program by temporarily stopping program execution at these points. To insert stops in the program, select the **Stops In** option on the **Run** menu. A stop-sign icon appears in the lower

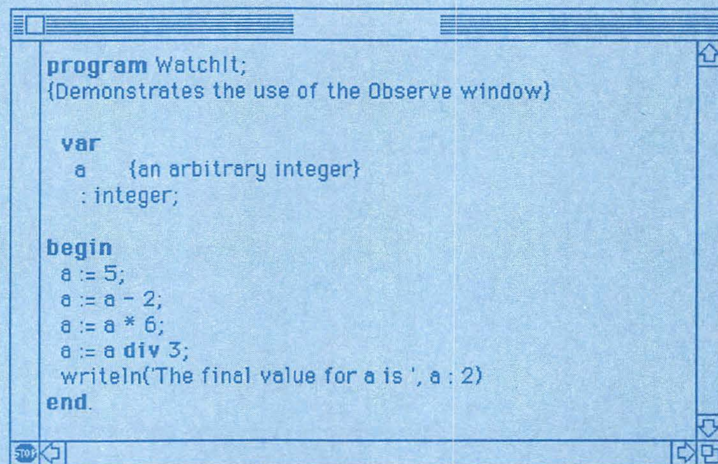


FIGURE 6.19

continued

left corner of the program window. See Figure 6.19.

When the screen arrow moves into the left column of this window, it appears as a stop sign. Depressing the mouse button deposits a stop sign icon. These stop icons signify the points in the program at which the value of **a** is sought. After execution is initiated, the program stops at every line that is prefixed with a stop sign icon, and the value of **a** before that line is executed is displayed in the **Observe** window.

Put in the two stops as shown in Figure 6.20.

```
program WatchIt;
{Demonstrates the use of the Observe window}

  var
   a    {an arbitrary integer}
    : integer;

begin
  a := 5;
  a := a - 2;
  a := a * 6;
  a := a div 3;
  writeln('The final value for a is ', a : 2)
end.
```

FIGURE 6.20

Notice that the **Stops In** option has been replaced by a **Stops Out** option in the **Run** menu.

Execute the program by selecting **Go** from the **Run** menu. The program runs until the first stop-sign icon is encountered. That sign is changed to a pointing-hand icon, and the value 3 is displayed in the **Observe** window. See Figure 6.21.

Select **Go**, and program execution continues until the next stop-sign icon is encountered. The hand icon moves to the location of the second stop icon, and the value 18 is shown for **a** in the **Observe** window.

A final selection of the **Go** option completes program execution, sending the final value of **a** to the output window. At this point **Unknown name** appears next to each expression listed in the **Observe** window.

continued

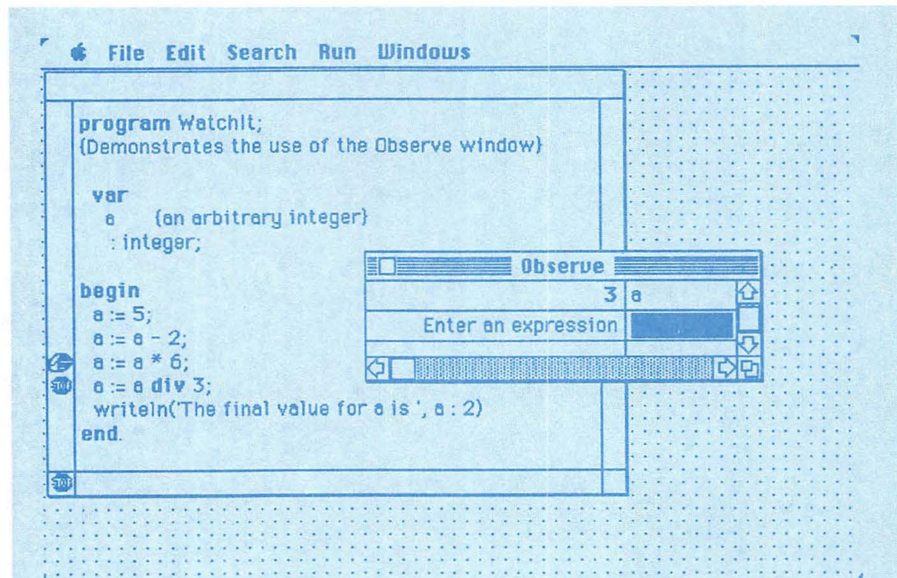


FIGURE 6.21

GO-GO

Select the **Go-Go** option from the **Run** menu. You should see a slight hesitation in the execution of the entire program at the two stop-sign icons. They are quickly changed to hand icons, and the value of **a** in the **Observe** window is 3 and then 18.

Selection of the **Step** option in the **Run** menu is not affected by the stops and executes the program one line at a time, as was done before stop icons were added. However, the selection of the **Step-Step** option causes the program to stop at each stop icon and displays the values in the locations specified in the **Observe** window. **Step-Step** must be selected again to activate execution until the next stop is encountered. After all stops have been passed, selection of the **Step-Step** option causes the program to complete execution.

To remove the stops, select **Stops Out** in the **Run** menu. This option is available only when the program window is activated. Activate the program window either by using the option in the **Window** menu or clicking the mouse when the arrow is in the window. Now select the **Stops Out** option in the **Run** menu, and all the stop signs disappear from the program window.

6.5 TYPICAL PROGRAMMING ERRORS

The following errors often arise.

1. Omission of the multiplication operator, *, between two values.
   Example: `x := 5(a+b);`
2. The use of the division operator, **div**, when dividing two real values.
   Example:
   ```
   var x, y, z : real;
   ...
   x := y div z;
   ```
3. The use of unmatched parentheses in a Pascal expression.
   Example: `y := ((a + b) / c));`
4. The division by a value of zero.
   Example:
   ```
   a := 0;
   d := b / a;
   ```
5. The use of an undeclared identifier in a program.
6. The placement of space between : and = for the assignment operator.
   Example: `z : = d / 5.6;`
7. The assignment of a real result to a declared integer memory location.
   Example:
   ```
   var i : integer;
       r : real;
   ...
   i := r;
   ```
8. The use of an improper expression on the left side of the assignment operator.
   Example: `x + y := z;`
9. Using two operators next to each other.
   Example: `x := y * -5;`
10. Typing the letter *l* (el) instead of the number 1 (one).
11. Typing the letter *O* (oh) instead of the number 0 (zero).

NONPROGRAMMING EXERCISES

1. Change the following algebraic expressions into MacPascal numeric expressions. (Consider each identifier to be a single letter.)
   a. $B^2 - 4AC$
   b. $abc + cde$
   c. $\frac{x - y}{x + y}$
   d. $3X^2 + 2X - 7$
   e. $P(1 + I)$
   f. $\frac{5x + 3(y - 15)}{y}$

2. Change each of the following MacPascal expressions into algebraic expressions. (Consider each identifier to be a single letter.)
   a. `x*y*y*y`
   b. `(a+b)*(a+b)`
   c. `m/n*a`
   d. `a/x*t/r/k`
   e. `d-e/j+k`
   f. `d-e/(j+k)`
   g. `(d-e)/(j+k)`
3. Evaluate each of the following MacPascal numeric expressions using the following values: $w = 3$, $x = 2$, $y = 4$, $z = 3$.
   a. `x+w*w`
   b. `y*y*x*w`
   c. `(y/x)*(y/x)`
   d. `w div x`
   e. `y/(x*x)`
   f. `y/x*x`
   g. `y/x/x`
   h. `(w+x)/z`
   i. `x+w/z`
   j. `w*z mod x`
4. Given the following declaration section of a program, identify any errors in statements a through f. Treat each statement individually.

```
program PickIt;
const
  a = 8;
  b = 15;
  c = 1.2;
var
  d, e: integer;
  f, g: real;
```

   a. `d := a + b;`
   b. `g : = b/c;`
   c. `f := (a + c)/g;`
   d. `g := 2 * b div a;`
   e. `e:= (2 + b) / c`
   f. `a := b * 100`
5. Describe the output of the following program.

```
program Numbers;

  a = 7;
  b = 2;
var
  c, d, e: integer;
begin
  c := a * b;
  d := 5 + c;
```

```
  e := d - 10;
  c := c div 6;
  e := e mod 6;
  writeln ('a':5,'b':5,'c':5,'d':5,'e':5);
  writeln (a:5,b:5,c:5,d:5,e:5)
end.
```

6. Take the program in Exercise 5 and assume an **Observe** window has been opened for the expression **c**. Trace the value of **c** through the entire program; begin the pointing-finger icon on the line below **begin**.

7. Describe the output for the following program if the values 70, 90, and 84 are entered as data.

```
program Demo;
  var
   testscore1, testscore2, final : integer;
   average : real;
begin
  write('Enter first test score: ');
  readln(testscore1);
  write('Enter second test score: ');
  readln(testscore2);
  write('Enter final exam score: ');
  readln(final);
  average := 0.30*testscore1+ 0.30*testscore2+ 0.40*final;
  writeln('Course grade is: ',average:5:2)
end.
```

PROGRAMMING EXERCISES

(G) 8. Write a program to accept two real, nonzero values and to output their sum, difference, product, quotient, and average with appropriate headings.

(G) 9. Write a program that asks the user to enter the numbers of different coin types he or she is carrying, and outputs their total dollar value.

(G) 10. Write a program to accept as input the name of a football player, the total yards gained in a game, and the total number of carries, and to output the player's name and average yardage gained per carry in the game.

(B) 11. Write a program to accept (1) a person's total gross weekly wage and (2) the total amount of payroll deductions. Output the take-home pay.

(G) 12. Write a program to accept as input the dimensions (length and width) of a room in your home, and to output the floor area in square feet.

(B) 13. Write a program that calculates the cost of tin needed to produce a cylindrical can. The formula for the surface area of a closed can is

$$2\pi r^2 + 2\pi r\, h$$

where r is the radius of the can and h is its height (see Figure 6.22).

If tin costs $0.75 a square foot, determine the cost of the tin for any can, given the radius and height measurements.

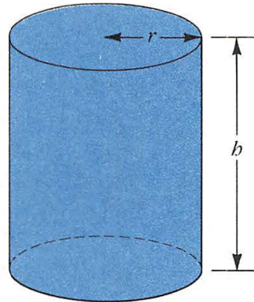


FIGURE 6.22

(B) 14. A salesperson earns a weekly salary plus a commission on sales for the week. Her regular weekly salary is \$150.00. If the commission rate is 7 percent and the salesperson totals \$2,000 in sales for the week, write a program to calculate and output her total weekly salary.

(G) 15. Write a program to accept as input the name of a city and the Celsius temperature at noon on a certain day. Output the name of the city and the equivalent Fahrenheit temperature, using the following formula:

$$\text{Fahrenheit} = \frac{9}{5}\text{Celsius} + 32$$

(G) 16. Write a program to convert a linear measurement from inches to centimeters. Enter the measurement in inches and make the conversion using the formula

$$\text{centimeters} = 2.54 \times \text{inches}$$

(B) 17. The BTU Fuel Oil Company makes fuel oil deliveries. Write a program to accept as input the customer account number and the number of gallons of fuel oil delivered. If fuel oil costs \$1.19 per gallon, output to the printer the customer account number and customer cost.

(B) 18. Write a program to calculate the weekly salary for an employee of the ABC Company. Input should provide the employee name, Social Security number, hourly wage, and the number of hours worked during the week. Assume that federal taxes take 14 percent of the gross salary, state taxes 7 percent, and Social Security 6.5 percent. Output the employee name, Social Security number, gross weekly wage, amounts of deductions, and the net weekly salary.

(G) 19. Write a program to accept the three coefficients (a, b, and c) of the standard linear equation ($ax + by + c = 0$, where a and b are not both 0). The program should output the equation and calculate and output (1) the x- and y-intercepts and (2) the slope of the line.

(G) 20. Write a program to enter a student's name and five test scores, and calculate and output the student's test average.

(G) 21. Write a program to create a file. Put three integers into a file. Close the file. Check to see if the file exists. Open the file for reading, and calculate and output the three integers.

(B) 22. Campaign buttons cost \$0.27 each to produce. If you produce 100 buttons and can sell them for \$1 each, write a program to determine your gross income (before deducting costs) and your net profit (after costs are deducted).

(B) 23. Given the annual gross sales for Acme Products, write a program to determine the average monthly gross sales.

(B) 24. Lonesome Acres is worth $175,000 and is taxed at an annual rate of $0.007 for each $1,000 that the property is worth. Write a program to determine and output the amount of taxes paid annually on Lonesome Acres.

(G) 25. Dead-Eye Hanrahan plays for the East Fargo Stonemasons. During their basketball season, he attempted 178 foul shots and made 123. Write a program to find and output his foul-shooting percentage.

(G) 26. Write a program that prints the following example in the **Drawing** window, calculates the correct answer, and displays it in its proper place:

```
   23
+  86
```

(B) 27. Gene Thumb owns the Seedy Landscaping Company. He needs seven juniper bushes and three maple trees for George Weeds's new property. Each juniper bush costs him $12.95, and each maple tree costs $22.50. Determine and output the total amount Gene charges George if his profit is to be 47 percent of his cost.

PART

TWO

PROGRAMMING STRUCTURES

C H A P T E R 7

System-Defined Functions and Procedures

7.1 INTRODUCTION

Suppose you have to prepare a meal for a large family gathering, but you don't have the time to make appetizers, main courses, salads, and desserts. You can call a caterer, tell him or her what you want and when you want it, and the meal is delivered, ready to eat. You may also want to prepare some of the meal yourself (your own secret recipe for chili), but your primary job is to make sure all the parts of the meal are ready on time. Designing a program is similar to preparing an elaborate meal.

At this stage, you should have some idea of what it takes to design and create elementary programs. Chapter 3 illustrated a problem solving method that can be used for this process. Macintosh Pascal, as do most other programming languages, acts as a "caterer" and offers additional assistance by supplying prewritten miniprograms that perform many of the activities commonly used by programmers like yourself. It would not be practical for MacPascal to include all the miniprograms that any programmer would ever need, but you can incorporate any available miniprograms in your own programs (that secret chili) to create a complete, well-designed product.

System-defined miniprograms are called *subprograms* and are of two types: *functions* and *procedures*. Both must receive data from the program (as the caterer would ask, "What kind of salad? How do you want the chicken seasoned?") and return the finished product (the Caesar salad and Chicken Kiev) to the place you specify, because you must tell the caterer where to deliver the meal.

A function or a procedure may be thought of as a "black box." You put some data into the box, and the answer pops out. You don't know how it was done, but you have the answer, so the solution method used in the box is not important. The one thing you must make sure of is that you use the box properly; that is, you must enter and remove data according to a set of rules.

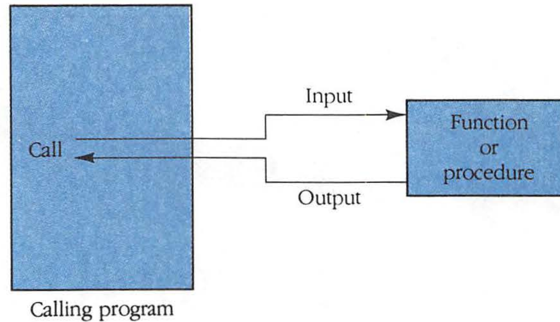


FIGURE 7.1

Figure 7.1 illustrates how functions and procedures operate. They accept a data list from a program statement that calls them and return the results of their calculation(s) to the calling statement. The data items sent to functions and procedures are called *arguments* or *parameters* and must be enclosed within parentheses.

Although system-defined functions and procedures operate in a similar fashion, they differ in how they send data to the calling program. Functions send data to memory locations, while procedures can send data to specific devices, memory locations, or screens in the system.

Actually, **write, writeln, writedraw, open, rewrite, read**, and **readln** are all procedures. **Write** and **writeln** take the information provided by the list of items enclosed in parentheses and move that information to a file or the **Text** window. **Writedraw** takes the information provided by the list of items enclosed in parentheses and moves that information to the **Drawing** window. **Open** and **rewrite** are procedures that are used to set up a link to an external file. Finally, **read** and **readln** are procedures used to accept data from the keyboard or a file and move these data to designated memory locations. So, you do have some experience using system-defined subprograms.

The distinction between functions and procedures becomes clearer as we discuss several of the more common numeric, string, and graphic functions and procedures. A complete listing of Macintosh Pascal functions and procedures may be found in Appendix F, at the end of this book.

7.2 NUMERIC FUNCTIONS

A numeric function accepts a numeric value from a calling program and returns the results of a calculation using that value to the calling program. These functions are generally used wherever identifiers are used, except when an identifier represents a location for the storage of results such as on the left side of the assignment operator (:=), or in a **read** or **readln** statement.

THE ABS FUNCTION

The **abs** function returns the absolute value of its argument; that is, it returns the identical number if its argument is zero or positive and returns the positive value of that argument if the argument is negative. For example, the absolute value of -5 is 5, and the absolute value of 35.8 is 35.8.

To call this function, you must use its name, **abs**, with the following format:

FORMAT: **abs(**`<real number or integer>`**)**

The function name **abs** must be followed by an argument of a single numeric variable, constant, value, or an expression and returns the absolute value of that argument. If the argument is an integer, the returned value is an integer; if the argument is a real number, the returned value is a real number.

Examples:

`x := abs(-7.02);`	The absolute value of -7.02 is 7.02, and that value is stored in the location represented by `x`.
`pd := abs(tr1 - tr2)/avg;`	The absolute value of `tr1 - tr2` is divided by `avg`, and the results are stored in the location represented by `pd`.

THE SQR FUNCTION

The **sqr** of a number returns the value of its argument multiplied by itself. For example, the square of -5 is $(-5)^2$ or 25, and the square of 7 is 7^2 or 49.

FORMAT: **sqr(**`<real number or integer>`**)**

The function name **sqr** must be followed by a real or integer argument of a single numeric variable, constant, value, or an expression, and returns the square of that argument. If the argument is an integer, the returned value is an integer; and if the argument is a real number, the returned value is a real number.

Examples:

`h := sqr(2.5);`	2.5^2 is 6.25, and that value is stored in the location represented by `h`.
`d := 2 * sqr(x) + x;`	The value of `x` is squared, multiplied by 2, added to `x`, and the result is stored in the location represented by `d`.

THE **SQRT** FUNCTION

The **sqrt** function returns the square root of its argument, that is, a value that when multiplied by itself gives the argument. For example, the square root of 36 is 6 since $6 \times 6 = 36$, and the square root of 81 is 9 since $9 \times 9 = 81$.

FORMAT: **sqrt(<nonnegative real number or integer>)**

The function name **sqrt** must be followed by an argument having a non-negative value or an expression evaluating to a nonnegative value and returns a real value representing the square root of that argument. If the argument is negative, an error message results.

Examples:

`a:=sqrt(121);`	$11 \times 11 = 121$, so 11 is returned and stored in the location represented by `a`.
`c:=sqrt(sqr(a)+sqr(b));`	The argument of **sqrt** is composed of the sum of two squares and is evaluated first. The square root of this result is returned and stored in the location represented by `c`.

The previous example shows that one function can be used as part of the argument of another function. In such a usage, the functions are said to be *nested*, and the innermost function is evaluated first.

THE **ODD** FUNCTION

The **odd** function returns a Boolean value that indicates whether its integer argument is an odd number or not. For example, 36 would return false since 36 is an even number, and 81 would return true since 81 is an odd number.

FORMAT: **odd(<integer>)**

Examples:

`a := odd(222);`	222 is an even integer, and therefore a value of false is stored in the location represented by `a`. This identifier must be declared to be of type Boolean.
`test := odd(f + s - t);`	If the expression `f + s - t` has a value that is an odd integer, the value true is stored in the location represented by `test`. Otherwise, the value false is stored in `test`.

The following two functions are similar in that both accept real numbers as arguments and return integer values. The same examples in both are used to clearly distinguish between the functions.

THE **ROUND** FUNCTION

The **round** function returns the value of its argument rounded to the nearest integer. For example, 456.7 would return 457 since .7 is greater than or equal to .5, and 456.4 would return 456 since .4 is less than .5.

FORMAT: **round(<real number>)**

The function name **round** must be followed by an argument that is a real number or an expression, and it returns an integer value representing the rounded argument.

THE **TRUNC** FUNCTION

The **trunc** function returns the truncated value of its real argument, that is, the argument value with its decimal or fractional part removed. For example, 456.7 would return 456, because the decimal part (.7) is removed, and 456.4 would return 456, because its decimal part (.4) is removed.

FORMAT: **trunc(<real number>)**

Examples:

`rpos := round(36.6);`	.6 is greater than or equal to .5, so the value 37 is stored in the location represented by `rpos`.
`tpos := trunc(36.6);`	The fractional part (.6) is removed, so the value 36 is stored in the location represented by `tpos`.
`rneg := round(-36.6);`	.6 is greater than or equal to .5, so the value −37 is stored in the location represented by `rneg`.
`tneg := trunc(-36.6);`	The fractional part (.6) is removed, so the value −36 is stored in the location represented by `tneg`.

THE **RANDOM** FUNCTION

The **random** function has no argument and is used to return a random integer in the range from −32768 to 32767. This function provides an air of unpredictability and is commonly used to simulate real-world events or games of chance.

FORMAT: **random**

Although you may not always want a number in the given range, you can use the **random** function in an expression to give a random integer in any range. If you are simulating the toss of a coin, you want only two results, because there are two possible outcomes of that event, for example, 0 and 1. You can use the following formula to produce consecutive random integers in a specified range:

```
random mod <number of consecutive outcomes>+<smallest outcome>
```

Using the coin toss, <number of consecutive outcomes> is 2 (the values 0 and 1), and <smallest outcome> is 0. So in this case the formula is `random mod 2 + 0` or `random mod 2`.

Example:

`dietoss := random mod 6 + 1;`	Since **mod 6** is used, only the values 0 through 5 are produced. The addition of the value **1** changes this range to values from 1 to 6. The results are stored in the location represented by the name `dietoss`.

The program and output in Figure 7.2 illustrate how some numeric functions are used in a program.

The same identifier, `arg`, is used as the argument for all the functions and is declared to be of type **real**. The identifiers **a**, **b**, and **c** are also of type **real** since they respectively indicate locations that store the absolute value, square, and square root of the given argument. The identifiers **d** and **e** are of type **integer** since they represent locations that store the results of **round** and **trunc**, respectively. The identifier `f` stores the result of an **odd** operation and so must be declared to be of type **Boolean.**

The **abs** function serves as the argument of the **sqrt** function in the assignment statement `c := sqrt(abs(arg))` and ensures that the **sqrt** function does not operate on a negative number, thereby avoiding an error message.

Problem: Write a program to determine the maximum diameter of a round table that can fit through a door whose height and width are entered at the keyboard.

The solution to the problem requires the entry of the height and width of a door and yields the diameter of the largest round table that can fit through the door. Figure 7.3 shows the physical setup.

The diameter of the table can be considered as the hypotenuse of a right triangle with the height and width as legs. In case you don't remember, the sum

```
program NumericFunctions;
{Illustrates the use of several numeric functions}

 var
  arg,                  {argument of functions}
  a,                     {absolute value}
  b,                     {square value}
  c                    {square root value}
   : real;
  d,                       {rounded value}
  e                     {truncated value}
   : integer;
  f                     {oddness value}
   : boolean;


begin
 write('Enter a number:  ');
 readln(arg);
 a := abs(arg);
 b := sqr(arg);
 c := sqrt(abs(arg));
 d := round(arg);
 e := trunc(arg);
 f := odd(d);
 writeln('absolute' : 10, 'square' : 10, 'square' : 10, 'rounded' : 10, 'truncated' : 10, 'oddness' : 10);
 writeln('value    ' : 10, 'value ' : 10, 'root ' : 10, 'value ' : 10, 'value   ' : 10, 'value ' : 10);
 writeln(a : 10 : 2, b : 10 : 2, c : 10 : 2, d : 9, e : 9, f : 9)
end.
```

```
Text
Enter a number:  34.56
  absolute    square    square   rounded truncated   oddness
   value      value      root     value    value      value
     34.56   1194.39      5.87       35       34       True
```

FIGURE 7.2

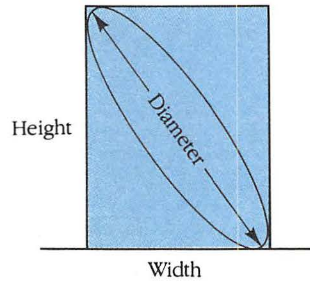


FIGURE 7.3

IPO CHART

Class	Identifier	Description
INPUT	height	height of door opening
	width	width of door opening
PROCESSING	diameter	maximum diameter of circular table

OUTPUT	Identifiers	Drawing
	height	
	width	
	diameter	

FIGURE 7.4

of the squares of the legs of a right triangle is equal to the square of the hypotenuse, or in this case,

$$\text{diameter}^2 = \text{height}^2 + \text{width}^2 \textit{ or } \text{diameter} = \sqrt{\text{height}^2 + \text{width}^2}$$

The IPO chart and N-S chart required for the solution process are given in Figures 7.4 and 7.5, respectively.

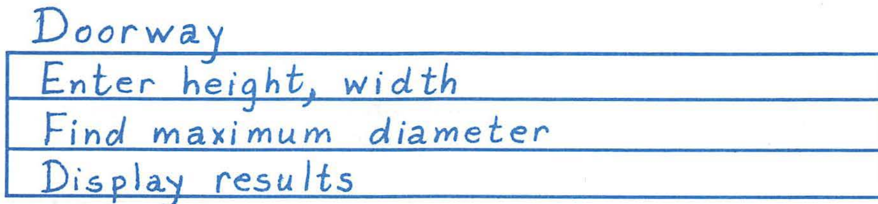


FIGURE 7.5

```
program Doorway;
{Determine the maximum diameter of a circular table that will fit}
{through a doorway, given the height and the width of the opening.}

 var
  height,                        {height of door opening}
  width,                         {width of door opening}
  diameter                 {diameter of circular table}
   : real;

begin
 write('Enter the height of the doorway (in feet):  ');
 readln(height);
 write('Enter the width of the doorway (in feet):  ');
 readln(width);
 diameter := sqrt(sqr(height) + sqr(width));
 writeln;
 writeln('The maximum diameter of a table that can fit');
 writeln('through this doorway (in feet) is ', diameter : 5 : 2)
end.
```

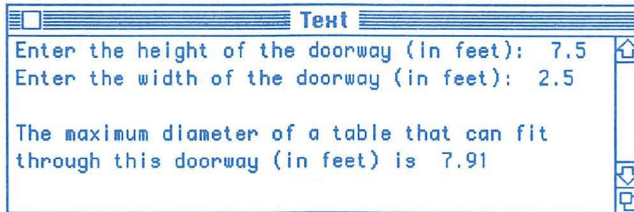


FIGURE 7.6

A program listing and run are shown in Figure 7.6.

The assignment statement that calculates the maximum diameter makes use of both the **sqr** and **sqrt** functions where the results of the **sqr** functions are parts of the argument of the **sqrt** function. The results are displayed in the **Text** window.

Problem: Write a program to simulate the choice of three digits to form a three-digit number that is the winning number in a pick-three lottery.

There are 10 digits (0 through 9) to choose from and the smallest one is 0, so the desired expression is `random mod 10 + 0` or `random mod 10.`

The IPO chart for the solution is shown in Figure 7.7 and the N-S chart in Figure 7.8.

A program listing and run of the solution are shown in Figure 7.9.

IPO CHART

Class	Identifier	Description
INPUT		
PROCESSING	digit1	first digit of winning lottery no.
	digit2	second digit of winning lottery no.
	digit3	third digit of winning lottery no.

	Identifiers		Drawing
OUTPUT	digit1		
	digit2		
	digit3		

FIGURE 7.7

Lottery
Choose first digit
Choose second digit
Choose third digit
Display results

FIGURE 7.8

```
program Lottery;
{Chooses the winning number in a pick-three lottery}

 var
  digit1,              {first digit of the winning number}
  digit2,              {second digit of the winning number}
  digit3           {third digit of the winning number}
   : integer;

begin
 write('The winning lottery number is ');
 digit1 := random mod 10;
 digit2 := random mod 10;
 digit3 := random mod 10;
 writeln(digit1 : 1, digit2 : 1, digit3 : 1)
end.
```

FIGURE 7.9

7.3 STRING FUNCTIONS AND PROCEDURES

A string function or procedure accepts one or more data items as arguments from a calling program, one of which must be a string, and returns the results of an operation using the arguments. These string functions can be used in the creation of word processing software packages.

THE **LENGTH** FUNCTION

The **length** function returns an integer representing the number of characters (letters, digits, spaces, and special keystrokes) in a given string. For example, the length of the string *Good Luck!* is 10. The space between the words and the exclamation point count as 2 of the 10 characters.

FORMAT: **length(<string>)**

Examples:

`nchar := length('$%^ *&@');`	The string constant $%^ *&@ contains seven characters, so 7 is stored in the location represented by **nchar**, an integer identifier.
`number := length(name);`	As long as **name** is declared to be a string, **number** receives an integer representing the number of characters in that string.

THE POS FUNCTION

The **pos** function searches a given string for the appearance of a particular substring and returns an integer that represents the position where the first occurrence of the substring starts in the given string. For example, if the string is abcdefg and the substring is **def**, the **pos** function returns 4, because **def** starts in the fourth character position in the string **abcdefg**.

FORMAT: **pos(<substring>, <string>)**

The function **pos** must have two arguments, both of type **string**. The first one is the substring being sought, and the second one is the string in which the search is done. It returns the position of the character in <string> where the first occurrence of the <substring> is found. If <substring> is not found in <string>, a value of 0 is returned.

Examples:

`place:=pos('sin','saint');`	Since there is no **sin** in **saint**, the value 0 is stored in the integer location represented by **place**.
`index:=pos('she',sentence);`	The position of the first occurrence of **she** in the string **sentence** is returned and its value stored in the location represented by **index**.

THE CONCAT FUNCTION

The **concat** function takes a sequence of strings and attaches them together to form one large string; that is, it concatenates the strings. For example, if the strings Mac, in, and tosh are concatenated in the order given, the result is the string Macintosh.

FORMAT: **concat(<string>, <string>, ...)**

The function **concat** can have any number of string arguments. Each string must be separated from the previous one by a comma, and the string returned by the function cannot exceed 255 characters. When this function is used, the second string is attached to the end of the first, the third is attached to the end of the second, and so on.

Examples:

`plural:=concat('dog','s');`	The string `s` is appended to `dog` and the resulting string `dogs` is stored in the string location represented by `plural`.

`a := 'pu';`
`c := 'ter';`
`m := 'com';`
`box := concat(m,a,c);`
	The strings `m`, `a`, and `c` are attached in that order, and the results, `computer`, are stored in the string location represented by `box`.

THE COPY FUNCTION

The **copy** function makes a copy of a given number of characters from a given string, starting at a specified position in the string. The results leave the original string unaltered. For example, if the original string is `abcdefg`, the position 2 (start at Position 2 of `abcdefg`, or `b`), and the number of characters 3 (copy three characters, including `b`), the substring `bcd` is stored in a given location.

FORMAT: **copy(**`<source string>,<index>, <count>)`

The function **copy** has three arguments; the first is a string and the others integers. The argument <index> indicates where the copying is to start; <count> indicates the number of characters to copy, starting with the character at position <index>; and <source string> indicates the string from which the copy is to be made.

Examples:

`d := copy('Macintosh', 4, 2);`	Two characters are copied from `Macintosh` starting from position 4. The two characters, `in`, become the contents of the string identifier `d`.

`i := 5;`
`c := 4;`
`s := 'Songbirds';`
`e := copy(s, i, c);`
	The substring `bird` is removed from `Songbirds` and becomes the contents of the string identifier `e`.

THE **DELETE** PROCEDURE

The next subprogram, **delete**, is a procedure rather than a function. Instead of being used as *part* of another Pascal statement, a procedure is, by itself, a complete Pascal statement. The values that are returned by a procedure are returned "through" the arguments listed for that procedure.

The **delete** procedure removes a specific set of characters from a given string and returns the altered string. The starting position and number of characters deleted are included as arguments in the procedure statement. For example, if three characters are removed from a string `operations`, starting at the third character in that string, the string would change to `options`. Unlike the previous functions discussed, the original string value *is* changed.

FORMAT: **delete(<string identifier>, <index>, <count>);**

The procedure **delete** causes the string stored in the location represented by <string identifier> to be changed by removing <count> characters from it starting at position <index>.

Examples:

`sample := 'asterisk';` `delete(sample, 2, 5);`	The five characters `steri`, starting at position 2, are removed from the string `sample`, whose contents are now `ask`.

`n := 1;`
`p := 3;`
`w := 'insert';`
`delete(w, p, n);`
	The character `s` is removed from the string `insert`, changing the contents of `w` from `insert` to `inert`.

It is worthwhile to emphasize again that the **delete** procedure, as with all procedures, is a complete statement in itself and cannot be used as part of another Pascal statement. The new value for a string argument is returned to the calling program through that string argument and not stored in the location represented by another identifier.

THE **INSERT** PROCEDURE

The **insert** procedure adds a substring to an existing string starting at a specified location in the given string. For example, if the string `e` is added to the string `quit` at the fourth position, the string changes from `quit` to `quiet`.

FORMAT: **insert(<source>, <destination>, <index>);**

The procedure **insert** causes the string <source> to be inserted into the string stored in the location represented by <destination> at the position <index>.

Examples:

```
source := 'prom';
insert('gra', source, 4);
```

The string **gra** is inserted into the string **source** at position 4, changing the contents of **source** to **program**.

```
d := 'reformation';
s := 'in';
delete(d, 1, 2);
insert(s, d, 1);
```

The first two letters of the string **reformation** are removed by the **delete** procedure, and **d** now contains **formation**. The characters in **s** are inserted at the beginning of the altered string **d** to again change its contents to **information**.

Problem: Write a program to create a password for entry into a computer file by taking the first three letters of the user's first name, the last two letters of his or her last name, and the third digit in the year of that user's birth and concatenating them in the order given.

The solution requires the entry of three pieces of information with the extraction of different substrings from each. The resulting substrings are concatenated and become the contents of a location represented by a string identifier. The resulting password is then displayed.

The IPO chart for the solution is shown in Figure 7.10, and the N-S chart in Figure 7.11.

IPO CHART

Class	Identifier	Description
INPUT	firstname	user's first name
	lastname	user's last name
	birthyear	user's year of birth
PROCESSING	a	first 3 letters of first name
	b	last 2 letters of last name
	c	3rd digit of birth year
	password	created system password

	Identifiers	Drawing
OUTPUT	firstname	
	lastname	
	birthyear	
	password	

FIGURE 7.10

MakePassword

Enter firstname, lastname, birthyear
Find first 3 letters of firstname
Find last 2 letters of lastname
Find 3rd digit of birthyear
Find password
Display password

FIGURE 7.11

```
program MakePassword;
{Create a password composed of parts of your last name,}
{first name, and year of birth.}

 var
  firstname,               {your first name}
  lastname,                {your last name}
  birthyear,                {year of birth}
  a,                              {first three letters of your first name}
  b,                              {last two letters of your last name}
  c,                              {the third digit of your year of birth}
  password            {concatenated strings to use for your password}
   : string;

begin
 write('Enter your first name:  ');
 readln(firstname);
 write('Enter your last name:  ');
 readln(lastname);
 write('Enter the year of your birth in the form XXXX:  ');
 readln(birthyear);
 a := copy(firstname, 1, 3);
 b := copy(lastname, length(lastname) - 1, 2);
 c := copy(birthyear, 3, 1);
 password := concat(a, b, c);
 writeln('Your secret password is ', password, '.  Don''t spread it around.')
end.
```

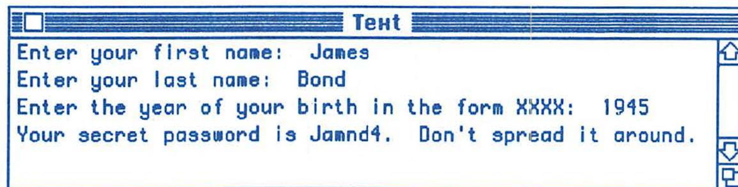


FIGURE 7.12

A program listing and run are shown in Figure 7.12.

The program uses the **copy** function to selectively copy portions of the input data. The three resulting strings are concatenated and then displayed.

Problem: Assume a lock combination is given in the form *LXXRXXLXX*, where *L* and *R* represent turns to the left and right, respectively, and the *X*'s represent digits. Write a program to change the combination by changing the last three characters.

The original combination and the new last term are entered as strings. The old last term is deleted from the original combination and the new last term inserted.

The IPO chart for the solution is shown in Figure 7.13, and the N-S chart in Figure 7.14.

A list and sample run of the final solution are shown in Figure 7.15.

The solution uses both the **delete** and **insert** procedures to determine the final result. There are no assignment statements in this program, because the value of the combination is passed through the argument `combo`.

IPO CHART

Class	Identifier	Description
INPUT	combo	lock combination
	last	change in last term
PROCESSING		

Class	Identifiers		Drawing
OUTPUT	combo		
	last		

FIGURE 7.13

Combination
Enter combo, last
Delete old last term
Add new last term
Display new combination

FIGURE 7.14

```
program Combination;
{Change the combination to a lock}

 var
  combo,                    {lock combination}
  last                      {change in last term}
   : string[10];

begin
 write('Enter original combination in the form LXXRXXLXX:  ');
 readln(combo);
 writeln;
 write('Enter new last term in the form LXX:  ');
 readln(last);
 writeln;
 delete(combo, 7, 3);
 insert(last, combo, 7);
 writeln('The new combination is ', combo)
end.
```

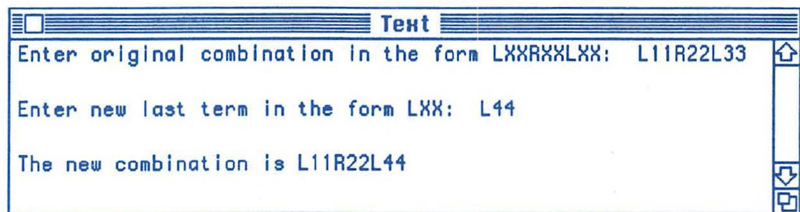


FIGURE 7.15

7.4 GRAPHIC PROCEDURES

Macintosh Pascal has the capability of producing a wide variety of graphic images in the **Drawing** window. You have been introduced to some of these graphic capabilities in Chapter 5. A more complete discussion of MacPascal graphics is presented in this section.

All graphic images are displayed in the **Drawing** window. The largest **Drawing** window size consists of an invisible grid containing approximately 150,000 dots, each of which may be illuminated. Each dot, point, or position is called a *picture element,* or *pixel.* Pixels in the largest window are arranged in a grid form that contains about 500 columns and 300 rows (see Figure 7.16).

In the figure, the point with coordinates (0,0), called the *origin,* is located at the upper left corner of the **Drawing** window. The horizontal axis, or x axis, ranges from 0 on the left to about 500 on the right; and the vertical axis, or y axis, ranges from 0 on the top to about 300 on the bottom of the window.

Recall that you control the size of the **Drawing** window, and pixels may be plotted only in a visible **Drawing** window. The maximum window size is available only if the entire screen is used for graphic output. If the visible window is smaller than this maximum size, the user may have to scroll the contents of this window to observe the entire graphic display. The size of the **Drawing** window that initially appears on the screen contains approximately 200 rows and 200 columns.

To identify a particular pixel, it is necessary to specify a pair of numbers called *coordinates.* The first coordinate represents the horizontal position or column number for the pixel, and the second coordinate represents the vertical position or row number. For example, the pixel represented by the coordinates (256,171) is located at the intersection of the 257th column and the 172nd row of the **Drawing** window, because both the row and column numbers start at 0. Figure 7.17 illustrates some pixels and their corresponding coordinates.

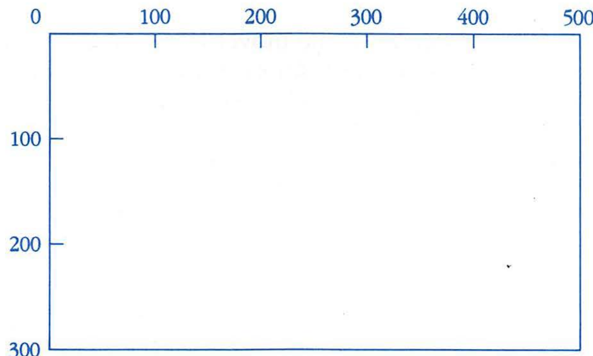


FIGURE 7.16

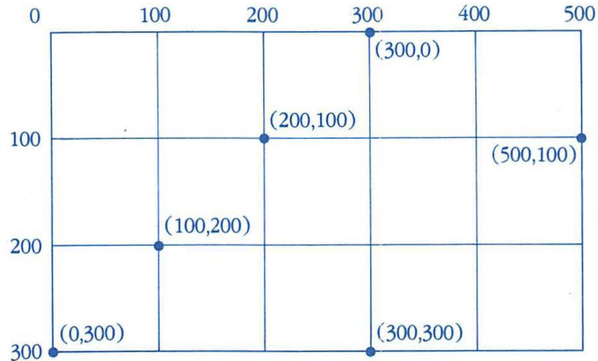


FIGURE 7.17

In order to draw graphic images in the **Drawing** window, a sequence of pixels that describes the image must be illuminated. For example, a horizontal line can be displayed by illuminating pixels with the same row value.

The discussion that follows introduces some system-defined procedures that are used to create graphic images. It may be helpful to think of these procedures as moving a pen around the **Drawing** window.

THE **MOVETO** PROCEDURE

The **moveto** statement introduced in Chapter 5 is actually a system-defined procedure. This procedure moves the drawing pen to the location with coordinates specified by its arguments. Nothing is drawn in the **Drawing** window. For example, the pen position is moved to the 31st column and 56th row when coordinates (30,55) are used in this procedure.

FORMAT: **moveto(`<horiz coord>`, `<vert coord>`);**

The procedure name **moveto** must be followed by the integer horizontal and vertical coordinates, <horiz coord> and <vert coord>, of a location. These coordinates are separated by a comma.

Examples:

`moveto(150, 75);`	The drawing pen is moved to the intersection of the 151st column and 76th row of the **Drawing** window.

`h := 200;`
`v := 100;`
`moveto(h, v);`
	The drawing pen is moved to the intersection of the 201st column and 101st row of the **Drawing** window.

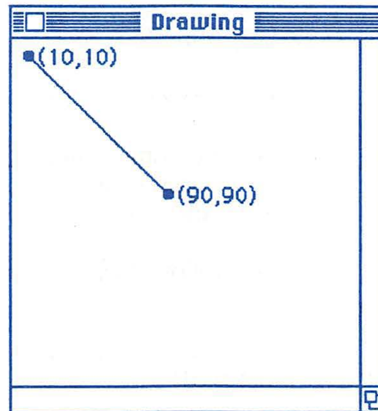


FIGURE 7.18

THE LINETO PROCEDURE

The **lineto** procedure draws a line from the current position of the pen to the location with coordinates specified by the arguments of this procedure. The pen is now located at the argument coordinates. For example, if the point (100,20) is used for arguments of the procedure, the drawing pen is moved from its current position to the location with coordinates (100,20), leaving a line as it moves.

FORMAT: **lineto(<horiz coord>, <vert coord>);**

The procedure name **lineto** must be followed by the integer coordinates of a location, separated by a comma.

Examples:

`moveto(10,10);` `lineto(90,90);`	A line is drawn from (10,10) to the point (90,90). See Figure 7.18.

`readln(x);`
`readln(y);`
`moveto(x,y);`
`lineto(x+50,y+30);`
	A line is drawn from the point whose coordinates are `(x,y)` to a point whose coordinates are 50 pixels to the right and 30 pixels below its starting point.

THE DRAWLINE PROCEDURE

The **drawline** procedure draws a line between the positions of two points, both of whose coordinates are used as arguments. This command has the same effect as moving to one point using **moveto** and drawing a line to a second point using

lineto. For example, a line could be drawn from (20,30) to (100,160) by using both pairs of coordinates in the argument of a **drawline** procedure.

FORMAT: **drawline(<horiz1>, <vert1>, <horiz2>, <vert2>);**

Drawline draws a line from a point with horizontal and vertical coordinates <horiz1> and <vert1>, respectively, to a point with horizontal and vertical coordinates <horiz2> and <vert2>. Commas are needed to separate the arguments.

The examples that follow mirror those used for **lineto**.

Examples:

`drawline(10,10,90,90);`	A line is drawn from (10,10) to the point (90,90).
`drawline(x,y,x+50,y+30);`	A line is drawn from `(x,y)` to a point whose coordinates are 50 pixels to the right and 30 pixels below `(x,y)`.

Although the previous procedures can be used to draw rectangles, Macintosh Pascal provides you with specific procedures to make this task easier.

THE **FRAMERECT** PROCEDURE

The **framerect** procedure accepts the upper, left, lower, and right boundaries of a rectangle and shows a line drawing of a rectangle with these boundaries. For example, if the top boundary is row number 10, the left boundary is column number 20, the bottom boundary is row number 50, and the right boundary is column number 70, a rectangle is drawn whose corners have coordinates (20,10), (70,10), (70,50), and (20,50) as shown in Figure 7.19.

FORMAT: **framerect(<top>, <left>, <bottom>, <right>);**

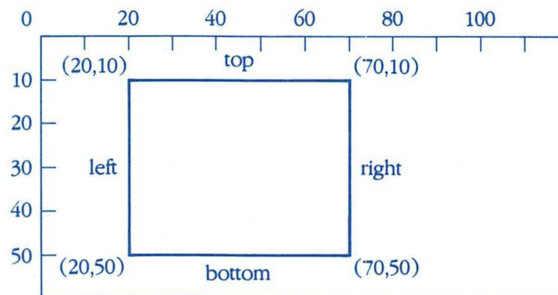


FIGURE 7.19

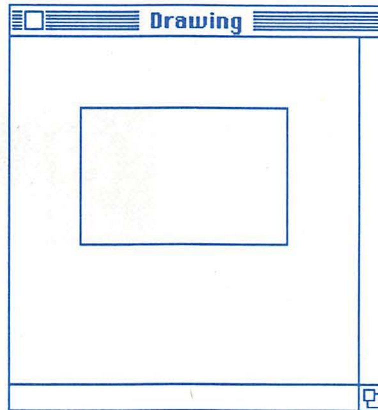


FIGURE 7.20

The arguments <top>, <left>, <bottom>, and <right> have integer values and must appear in the indicated order, separated by commas.

Example:

`readln(x);`
`readln(y);`
`framerect(y,x,y+80,x+120);`
	After the coordinates of the upper left corner of a rectangle have been entered, a line drawing of a rectangle is shown with width 80 and length 120 as shown in Figure 7.20.

THE **PAINTRECT** PROCEDURE

The **paintrect** procedure accepts the upper, left, lower, and right boundaries of a rectangle, as does **framerect**, but completely shades the inside of that rectangle rather than displaying its outline.

FORMAT: **paintrect(<top>, <left>, <bottom>, <right>);**

The arguments <top>, <left>, <bottom>, and <right> have integer values and must appear in the indicated order, separated by commas.

Consider the "paint" equivalent of the example shown in the frame section.

Example:

`readln(x);`
`readln(y);`
`paintrect(y,x,y+80,x+120);`
	After the coordinates of the upper left corner of a rectangle have been entered, a shaded rectangle is drawn with width 80 and length 120, as shown in Figure 7.21.

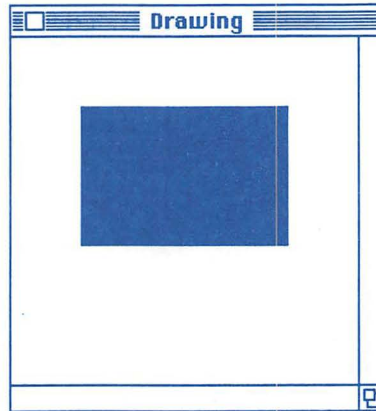


FIGURE 7.21

THE **INVERTRECT** PROCEDURE

The **invertrect** procedure accepts the upper, left, lower, and right boundaries of a rectangle, as do **framerect** and **paintrect**, but inverts each pixel enclosed by those boundaries. That is, every white pixel becomes black and every black pixel becomes white. This procedure can be used to erase a rectangle from the **Drawing** window.

FORMAT: **invertrect(<top>, <left>, <bottom>, <right>);**

The arguments <top>, <left>, <bottom>, and <right> have integer values and must appear in the indicated order, separated by commas.

Example:

`invertrect(y1,x1,y2,x2);`	All the white pixels enclosed in the rectangle with these boundaries become black, and all the black pixels, white. In effect, it erases the shaded rectangle with the same arguments.

Although rectangles can be drawn using horizontal and vertical lines, the introduction of **framerect** and **paintrect** make the process easier. The same is not true for circles. It is considerably more difficult to draw a circle with "straight line" functions. Luckily, MacPascal again comes to the rescue with the **paintcircle** and **invertcircle** procedures.

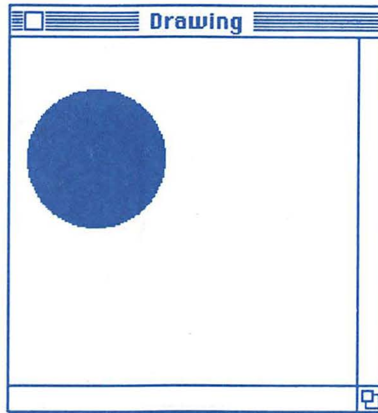


FIGURE 7.22

THE **PAINTCIRCLE** PROCEDURE

In order to display a circle, only three arguments are required: the horizontal coordinate of the center of the circle, the vertical coordinate of the center, and the radius of the circle. The **paintcircle** procedure accepts these quantities as arguments and displays the corresponding circle.

FORMAT: **paintcircle(<xcenter>, <ycenter>, <radius>);**

When **paintcircle** is executed, a shaded circle with center coordinates <xcenter> and <ycenter> and radius <radius> is drawn in the **Drawing** window. The arguments must be integers and must appear in the indicated order, separated by commas.

Example:

`cx := 50;`
`cy := 70;`
`rad := 40;`
`paintcircle(cx,cy,rad);`
	A shaded circle, centered at (50,70) with radius 40, is drawn in the **Drawing** window as depicted in Figure 7.22.

THE **INVERTCIRCLE** PROCEDURE

The **invertcircle** procedure accepts the same arguments as **paintcircle**, the center coordinates and radius of a circle, but inverts each pixel enclosed by the specified circle.

FORMAT: **invertcircle(<xcenter>, <ycenter>, <radius>);**

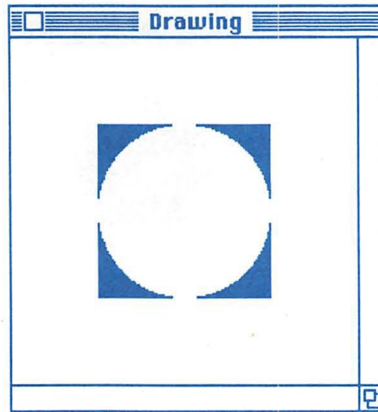


FIGURE 7.23

Any white points in the circle change to black, and any black points in the circle change to white.

Example:

`paintrect(50,50,150,150);` `invertcircle(100,100,50);`	A shaded rectangle is drawn and a circular area inside the rectangle is changed back to white, with the parts of the rectangle not included in the circle remaining black, as in Figure 7.23.

There is no system-defined procedure for producing a framed circle.

Problem: Write a program to draw a framed circle given the coordinates of the center and the radius.

Input values are needed for the coordinates of the framed circle and the radius of the desired circle. The only output value is the display of the circle in the **Drawing** window. A painted circle is drawn, its radius decreased by 1, and an inner circle inverted. This gives the appearance of a framed circle.

The IPO chart for the solution is shown in Figure 7.24, and the N-S chart in Figure 7.25.

A program listing and run are shown in Figure 7.26.

IPO CHART

Class	Identifier	Description
INPUT	xcen	x coordinate of center of circle
	ycen	y coordinate of center of circle
	radius	radius of circle
PROCESSING	innerradius	radius of inner circle

	Identifiers		Drawing
OUTPUT	xcen		framed circle
	ycen		
	radius		

FIGURE 7.24

Enter circle center coordinates
Enter radius of circle
Draw painted circle
Find inner radius
Erase inner circle

FIGURE 7.25

```
program FrameCircle;
{Produces a framed circle}

 var
  xcen, ycen,                    {coordinates of center of circle}
  radius,                          {radius of circle}
  innerradius               {radius of cut-out circle}
   : integer;

begin
 write('Enter x coordinate of center of circle:  ');
 readln(xcen);
 write('Enter y coordinate of center of circle:  ');
 readln(ycen);
 write('Enter radius of circle:  ');
 readln(radius);
 paintcircle(xcen, ycen, radius);
 innerradius := radius - 1;
 invertcircle(xcen, ycen, innerradius)
end.
```

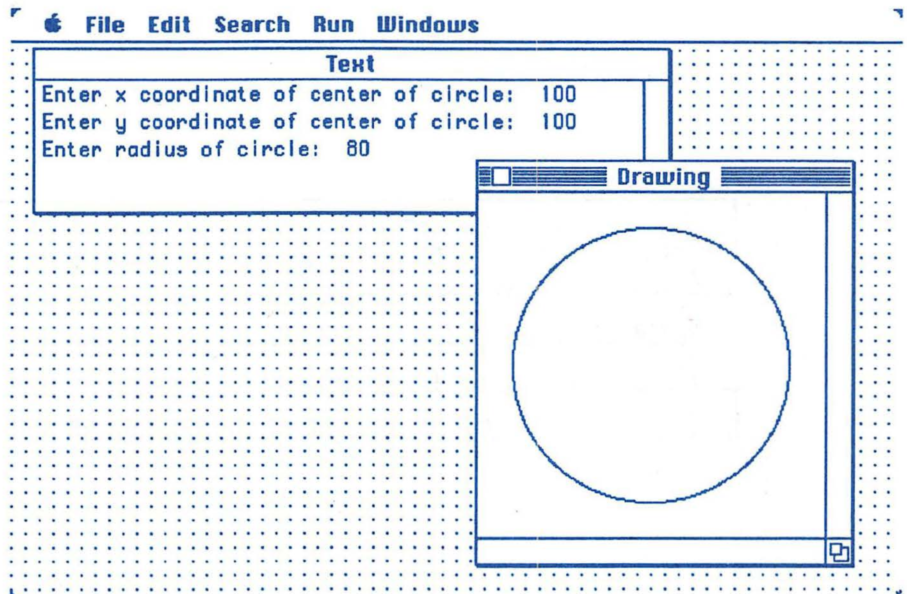


FIGURE 7.26

Problem: Write a program to draw a framed rectangle and a painted circle, and label both figures.

The **framerect** and **paintcircle** procedures can be used to draw the figures, and the **writedraw** statement to properly label each. The **moveto** procedure is used to center the descriptions under each of the figures. It may take you a few trials to get this labeling just the way you want it.

The IPO chart is not needed, because no identifiers are involved, but the N-S chart is shown in Figure 7.27.

Figure 7.28 gives a list and run of the solution.

RectangleAndCircle
Draw frame rectangle
Label frame rectangle
Draw painted circle
Label painted circle

FIGURE 7.27

```
program RectangleAndCircle;
{Draw a framed rectangle and painted circle}

begin
 framerect(10, 10, 75, 150);
 moveto(25, 90);
 writedraw('Framed Rectangle');
 paintcircle(80, 140, 35);
 moveto(35, 190);
 writedraw('Painted Circle')
end.
```

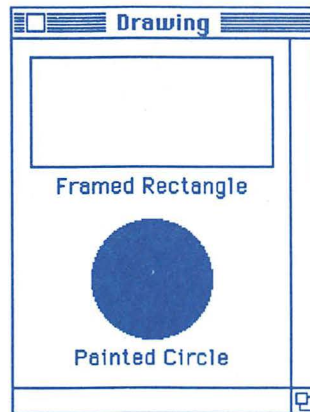


FIGURE 7.28

Problem: Write a program to simulate the horizontal motion of a ball by drawing the ball in one location, entering a horizontal change, erasing the ball at the old location, and drawing it at the new location.

The motion of an object along a straight line is called a *translation*. This program asks for a horizontal translation. In such a translation, the vertical or *y*-coordinate points on the object do not change; only the first argument in the **paintcircle** procedure changes. After the ball is drawn in its original location using **paintcircle**, it is erased using **invertcircle** before being redrawn at the new location. The entry of the horizontal translation value, `xtran`, between the initial display and the erasing ensures that the old circle remains on the screen until you want to see it move. The erasing and redrawing take place after you press the Return key for the entry of `xtran`.

Figure 7.29 shows the IPO chart for the solution, and Figure 7.30 the N-S chart.

The Pascal solution, text output, and **Drawing** window output after the movement are illustrated in Figure 7.31.

IPO CHART

Class	Identifier	Description
INPUT	xtran	horizontal translation value
PROCESSING		

	Identifiers		Drawing
OUTPUT	xtran		moving ball

FIGURE 7.29

MoveOval
Draw circle at original location
Get horizontal translation value
Erase circle
Draw circle at new location

FIGURE 7.30

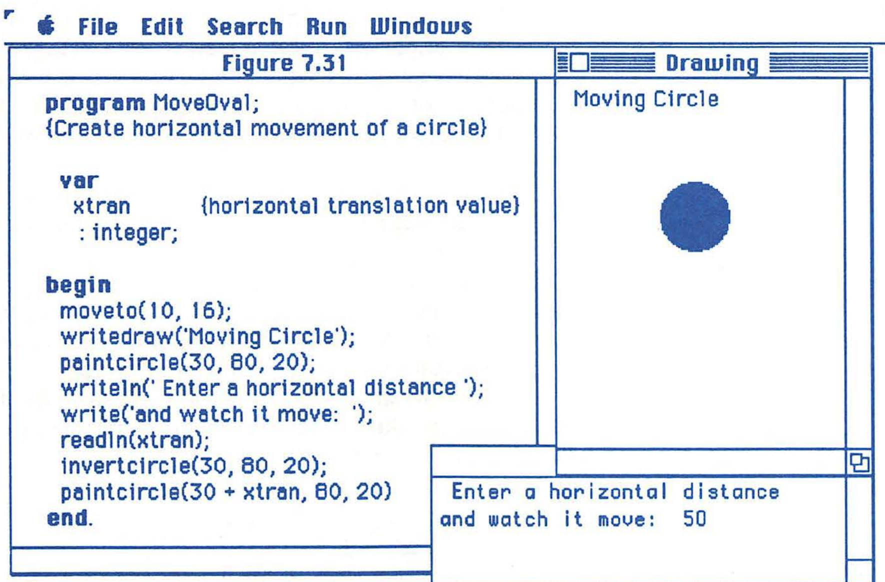


FIGURE 7.31

Problem: The Dandy Candy Company is test-marketing the new candy bar Marshmallow Surprise. It conducts a survey by sending a sample to the members of a targeted audience and asking them whether they would buy Marshmallow Surprise. Write a program that accepts the number of yes and no responses, calculates the percentage of respondents answering yes and no, and displays the results in a vertical bar graph as shown in Figure 7.32.

Once the percentages of yes and no answers have been calculated, these quantities are used to determine the height of the bars representing their respective percentages.

The IPO chart defining identifiers is given in Figure 7.33.

The solution is outlined in the N-S chart in Figure 7.34.

A complete program listing and sample run are presented in Figure 7.35.

The identifiers `topyes` and `topno` store the top boundaries of the yes and no rectangles. The boundaries are calculated by subtracting the percentage of yes (or no) answers from the bottom boundary (arbitrarily chosen at 150). For example, if 60 percent of the respondents answered yes, the top boundary is 150 − 60 or 90. After the values for these quantities are known, the graph is drawn.

The planning of what to put where should be done on graph paper prior to or as part of the design stage to determine the parameters in the **paintrect** procedure. In this case, only the top boundary changes with a new input of data. The other three boundaries are fixed in the solution.

The graph should be given a title, an axis (or axes) drawn, and the rectangles properly described. The `yes` and `no` titles below the rectangles and the percentages above the rectangles complete the description of the graph. It is important to specify a field width when you display values in the **Drawing** window using a **writedraw** procedure. If you are not careful, the descriptions may not appear where you want them to.

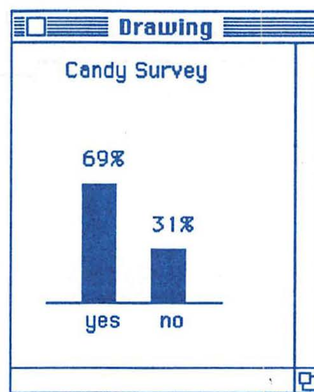


FIGURE 7.32

IPO CHART

Class	Identifier	Description
INPUT	yes	number of "yes" responses
	no	number of "no" responses
PROCESSING	total	total number of responses
	pctyes	percentage of "yes" responses
	pctno	percentage of "no" responses
	topyes	top boundary of "yes" bar
	topno	top boundary of "no" bar

Class	Identifiers		Drawing
OUTPUT			bar graph of percentages of "yes" and "no" responses

FIGURE 7.33

CandySurvey

Enter number of "yes" and "no" responses
Find total number of responses
Find percentage of "yes" and "no" responses
Find top boundaries for rectangles
Put title on graph
Draw horizontal axis
Draw "yes" rectangle
Label "yes" rectangle
Draw "no" rectangle
Label "no" rectangle

FIGURE 7.34

```
program CandySurvey;

 var
  yes, no, total, topyes, topno, pctyes, pctno : integer;

begin
{Enter data and calculate percentages}
 write('Enter number who would buy:  ');
 readln(yes);
 write('Enter number who would not buy:  ');
 readln(no);
 total := yes + no;
 pctyes := round(yes / total * 100);
 pctno := round(no / total * 100);

{Draw bar graph}
 topyes := 150 - pctyes;    {Find top limits for rectangles}
 topno := 150 - pctno;

 moveto(30, 20);                  {Label graph}
 writedraw('Candy Survey');

 drawline(20, 150, 120, 150);      {Draw base line}

 paintrect(topyes, 40, 150, 60);    {Draw and label 'yes' rectangle}
 moveto(42, 165);
 writedraw('yes');
 moveto(40, topyes - 10);
 writedraw(pctyes : 2, '%');

 paintrect(topno, 80, 150, 100);    {Draw and label 'no' rectangle}
 moveto(85, 165);
 writedraw('no');
 moveto(80, topno - 10);
 writedraw(pctno : 2, '%')
end.
```

Text

```
Enter number who would buy:  5234
Enter number who would not buy:  2338
```

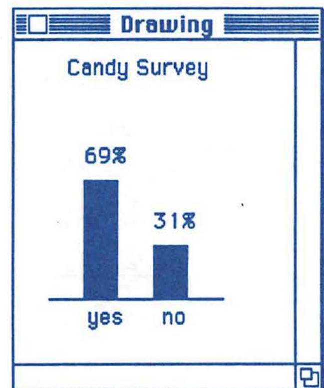


FIGURE 7.35

This chapter has introduced some of the more common numeric, string, and graphic functions and procedures that are available for your programs. You are encouraged to use them whenever possible. You should also consult your *Macintosh Pascal Manual* for additional helpful functions and procedures. You've now learned how to use a "caterer" for your "dinner party." The next step is to try to create some of your own "dishes" to add to the menu, that is, creating your own special functions and procedures.

7.5 TYPICAL PROGRAMMING ERRORS

Some typical errors made in Pascal programming are as follows:

1. Using an improper value for the argument of a function or procedure.
   Example: `x := sqrt(-121);`
2. Using the results of a function as an incorrect data type.
   Example:
   ```
   var  x : integer;
   begin
    x := odd(24);
   ```
3. Improper number of arguments for a function or procedure.
   Example: `delete(from, to);`
4. Incorrect order of arguments in a function or procedure.
   Example: `paintcircle(radius, xcenter, ycenter);`
5. Failure to activate the **Drawing** window to observe the results of graphics commands.

NONPROGRAMMING EXERCISES

1. Classify each of the following Pascal statements as valid or invalid. If the statement is invalid, explain why.
   a. `sqr(a) := sqr(b) + sqr(c);`
   b. `framerect(100, 10, 50, 150);`
   c. `x := paintcircle(30, 30, 30);`
   d. `round(money);`
   e. `y := trunc(sqrt(x));`
   f. `p := pos(source, length(source)-5);`
   g. `answ := concat(a, b, c, d);`
   h. `insert(a, b, 3);`
   i. `invertcircle(a, b, c, d);`
   j. `y := round(length('test string');`
2. Evaluate each of the following expressions involving functions.
   a. `trunc(3.999)`
   b. `sqr(round(4.711))`
   c. `concat('your', 'self')`
   d. `length('This is a test.')`

   e. `odd(round(sqrt(50)))`
   f. `abs(trunc(sqrt(7)))`
   g. `copy('galaxy', 4, 2)`
   h. `pos('def', 'abcdefghijklm')`
   i. `odd(2 * random)`
   j. `concat(copy('deposit', 5, 3), copy('comfort', 1, 3))`

3. Describe what each of the following procedures does.
   a. `delete('representation', 1, 2);`
   b. `insert('te', 'ingrate', 3);`
   c. `moveto(80, 90);`
   d. `lineto(150, 60);`
   e. `drawline(10, 20, 30, 40);`
   f. `framerect(100, 35, 121, 136);`
   g. `paintrect(100, 35, 121, 136);`
   h. `invertrect(30, 30, 70, 70);`
   i. `paintcircle(50, 50, 25);`
   j. `invertcircle(50, 50, 20);`

4. Write an expression containing a function or procedure to implement the tasks described in each of the following statements.
   a. Change the string *make* to *mistake* by inserting *ist*.
   b. Choose a random integer between 0 and 36.
   c. Round a number represented by x to two decimal places.
   d. Draw a solid circle centered at (30, 30) with radius 20.
   e. Find the difference of the square roots of x and y.
   f. Find the square root of the difference of x and y.
   g. Determine whether or not the rounded value of x is odd or not.
   h. Find the length of the string stored in the location represented by `str`.
   i. Determine the character position of the string m in the string *abcdefghijklmnopqrstuvwxyz*.
   j. Change the string *bookkeeper* to *beeper* by removing *ookk*.
   k. Make a copy of the middle three characters of a nine-character string called `test`.

PROGRAMMING EXERCISES

(B) 5. Write a program to find the federal tax on an employee's gross earnings if the tax rate is 37 percent. Round the amount of the tax to two decimal places and display the rounded result.

(G) 6. Write a program to draw the letter *W* in the **Drawing** window.

(G) 7. Write a program to draw a tic-tac-toe grid in the **Drawing** window.

(G) 8. Write a program to find the length of the diagonal of a rectangle given the lengths of two consecutive sides.

(G) 9. Write a program to simulate the roll of a pair of dice by having the computer twice choose random integers between 1 and 6 and displaying their sum as the result of the roll.

(B) 10. A house and the land that it's built on are valued at $175,000. The owners pay annual taxes of $3,320. Find the tax rate, in percent, and round it to three decimal places.

Assume the house and its property are taxed at 80 percent of its value. Display the rounded result.

(G) 11. Write a program to draw a capital *H* on the **Drawing** screen, using three rectangles.

(G) 12. Write a program to draw a solid square in the middle of the **Drawing** window, and place a round hole at the center of the square.

(B) 13. Write a program that accepts an employee's Social Security number as a nine-digit string and inserts hyphens in the appropriate places. Output the hyphenated Social Security number.

(G) 14. Write a program to find the area of a triangle given the lengths of its three sides using the formula

$$A = \sqrt{s(s - a)(s - b)(s - c)}$$

where a, b, and c are the lengths of the sides and s is one-half the sum of the sides.

(G) 15. Write a program to find the time (in seconds) it takes a ball to fall from the top of a building h feet high. Use the following formula:

$$h = 16t^2$$

where t is the time in seconds, and h is the height of the building in feet.

(B) 16. The Quik-Stop Food Store has been sold to an owner that would like to change its name to The Slo-Start Food Store. Write a program that accepts a string containing the old name of the store and automatically changes it to the new name.

(B) 17. Write a program that accepts the two-word name of a firm and forms an acronym consisting of the first letter in each part of the name; for example, if the firm's name were Highland Industries, the program would produce *HI.*

(B) 18. The codes for access to money machines are determined by the first letter of the user's last name, the last letter of his or her first name, and the middle three numbers of his or her Social Security number. Write a program to create such a password, given the pertinent information.

(B) 19. The travel magazine *Ports Illustrated* would like to create a mailing label for subscribers. In addition to the name and complete address of the subscriber, *PI* uses a line of code representing an account number across the top of the label. This code consists of the first four letters of the subscriber's last name, followed by the subscriber's first initial, street name, zip code, and subscription expiration date. Write a program to accept the subscriber's name and address, as well as the expiration date of the subscription, and to produce the mailing label.

(G) 20. Write a program that accepts the four boundaries of a square, divides that square into four identical squares, and displays the drawing as shown in Figure 7.36.

(G) 21. Write a program that displays a rectangle, asks for a vertical translation parameter, erases the rectangle, and displays it at the location determined by adding the input vertical parameter to the vertical coordinates of the original position.

(B) 22. Write a program that accepts the following total sales for Tico's Tacos and displays these results using a horizontal bar graph similar to the one shown in Figure 7.37.

YEAR	*SALES*
1984	$12,500
1985	$21,700
1986	$23,800

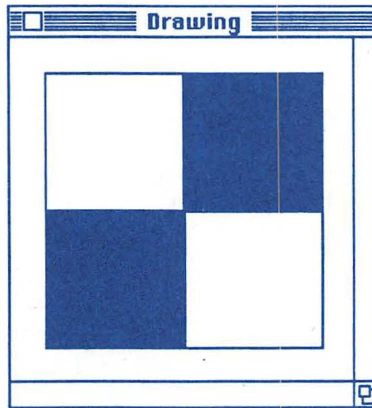


FIGURE 7.36

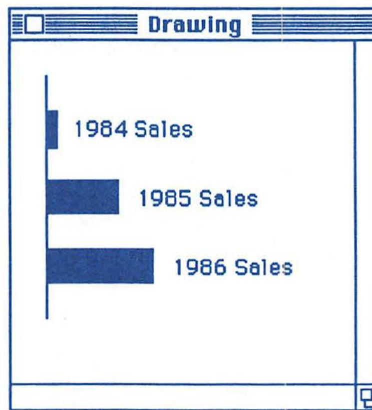


FIGURE 7.37

(G) 23. Write a program to draw a solid rectangle given the coordinates of its center, its length, and its width. Label the rectangle by displaying the word `Rectangle` immediately below the drawn figure.

(B) 24. Write a program that displays a bar graph (similar to the one in Figure 7.32) to illustrate what percentage of company money is paid for each employee in the areas of salary, medical benefits, and pension. The program should accept the salary, amount paid in medical benefits, and pension contribution for a typical employee in a week.

CHAPTER 8

Programmer-Defined Functions and Procedures

8.1 INTRODUCTION

In the previous chapter, we used the catered dinner party analogy to discuss system-defined functions and procedures and show how they could enhance and facilitate the creation of programs as a caterer enhances and facilitates a dinner party by providing dishes that require no preparation on the part of the host. Functions and procedures were "black boxes"; you didn't know how they got their answers, but the answers were useful and correct if we sent them the proper information. Now you're ready to add your own secret dishes to the big dinner party, that is, to create your own functions and procedures.

Let's take a look at this situation from your point of view as the creator of a particular entree for the meal. You have to gather all the ingredients for the entree, use a specific sequence of steps to combine these ingredients to produce the entree, and bring the entree to the party. When creating a function or procedure, you have to know what data are to be used, what results are required, and what steps you have to take to produce these results. This process sounds very much like the process you have to go through to create a program. That is why functions and procedures are sometimes called *subprograms:* they are smaller programs that are used by other functions, procedures, or main programs.

The problem of creating a subprogram boils down to the following parts:

- How to name that subprogram so that it can be called
- How to pass to the subprogram the data that the subprogram needs
- How to perform the task(s) required to produce the desired results
- How to return these results to the calling program

You have already had some experience in writing programs, so let's look at how this experience can be used in the creation of functions and procedures.

8.2 FUNCTIONS

The only major difference between the structure of a programmer-defined function and that of a program is in the header line. In a program, the header line begins with the keyword **program**, followed by the program name. In a function, the header line begins with the keyword **function**, followed by the name of the function, a parameter list, and a function type.

The format for an entire function is as follows:

FORMAT:

```
function <identifier>(<parameter list>):<fct type>;
  < local identifier declarations>
begin
  <body of function>
end;
```

The keyword **function** must be followed by a name, <identifier>, that uniquely identifies this function.

Enclosed within parentheses is a list of identifiers, called *parameters,* that are used to transfer data from the calling program to the function. Each set of parameters consists of an identifier or identifiers of the same type, separated by commas and followed by their common type. The type is separated from the identifier list by a colon, and a semicolon separates identifiers of different types. Each identifier receives a value from the calling program when the function is activated. A function need not have any parameters, as illustrated by the system-defined function **random.**

A colon appears after the closing parenthesis of the parameter list followed by <fct type>, the type of the data that are passed to the calling program through the function with the name <identifier>. Recall that when you use a function a value is passed back through the function name. The header line for the function ends with a semicolon.

Following the function header line is the function's declaration section. This section may include **const**, **var**, and **type** declarations, <local identifier declarations>, for identifiers that are used solely within the confines of the given function. Identifiers defined in the declaration section and those appearing in the function parameter list are called *local identifiers* and are not considered declared outside of their use in the function.

As in a main program, the statements that perform the desired task in a function, <body of function>, must start with the keyword **begin** and terminate with the keyword **end.**

There must be at least one statement, in the body of the function, that assigns a value to the function name <identifier> before returning to the calling program. Failure to do this sends unpredictable results back to the main program.

Unlike a main program, the last **end** for a function is followed by a semicolon rather than a period. The reason for this is that when the function is placed into the program, it precedes the main body of the program.

Example:

```
function Avg(a,b:integer):real;
 var
  sum : integer;
 begin
  sum := a + b;
  Avg := sum / 2
 end;
```

The keyword **function** is followed by the identifier `Avg`. This is the same name that is used when this function is called. The parameter list consists of the identifiers `a` and `b`, both of type **integer**. A colon follows the parameter list and separates that list from the type **real**. This type indicates that the function `Avg` returns a real number to the calling program. The identifier `sum` is of type **integer**. It is local and not considered declared outside of the confines of this function. The body of the function includes a statement that assigns a value to `Avg`, the function identifier, and is placed in a **begin-end** pair, with a semicolon following **end**.

Assume that a calling statement to this function is

```
grade := Avg(x, y);
```

The values sent to this function by this calling statement pass specific values that are stored in locations represented by these identifiers. The values stored in the locations, `x` and `y`, known in the main program are copied into locations `a` and `b`, respectively, declared in the function. In Figure 8.1, there is a total of four distinct memory locations involved. Both `a` and `x` contain the same data, as do `b` and `y`.

Parameters `a` and `b` and all parameters that act in this way are known as *value parameters.*

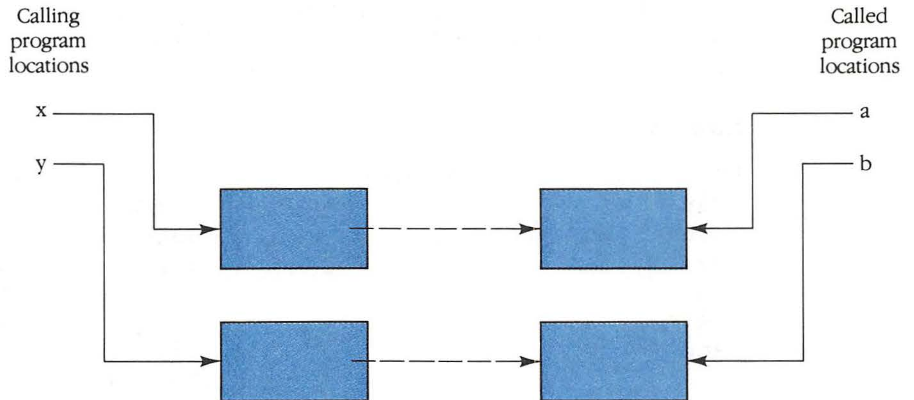


FIGURE 8.1

Example:

```
function Find(word : string;
         p : integer) : char;
begin
 Find := copy(word, p, 1)
end;
```

The function **Find** accepts a string **word** and an integer **p**, locates the character in position **p** in **word**, and returns that character through the identifier **Find**. There are no local identifiers in this function. The function **Find** calls the system-defined string function **copy** and illustrates the fact that one function can call another.

8.3 PROCEDURES

The structure of a programmer-defined procedure also resembles the structure of a program. The header line for a procedure uses the keyword **procedure** rather than the keyword **program** and contains the procedure identifier followed by a parameter list.

A major difference between a programmer-defined function and a programmer-defined procedure is that the function must return one value to the calling program through the function name. Additional values may be returned through the identifiers in the parameter list. A procedure, on the other hand, can return any number of values, and these must be returned through the identifiers declared in the parameter list. It is also possible to have a procedure that has no parameters and returns no values.

The format for an entire procedure is as follows:

FORMAT: **procedure** <identifier> (<parameter list>);
<local identifier declarations>
begin
<body of procedure>
end;

The keyword **procedure** must be followed by a name, <identifier>, that uniquely identifies this procedure.

The parameter list that follows contains a list of identifiers that are used to transfer data to and from the procedure. Again, each set of parameters consists of an identifier or identifiers of the same type, separated by commas and followed by their common type. The type is separated from the identifier list by a colon, and a semicolon separates identifiers of different types. The procedure header ends with a semicolon.

Recall that if a value or values are to be returned to a calling program, they are returned through the parameters themselves, rather than through the name of the procedure. Value parameters are not sufficient for this purpose. A change in a subprogram's local memory locations does not automatically mean that the corresponding locations in the calling program are also changed. In Figure 8.1, even though the values for the locations a and b are changed by the called program, the values for locations x and y are not affected. Value parameters can be thought of as one-way streets, allowing the movement of data only from the calling program to the called function or procedure.

MacPascal also allows the use of two-way or *variable parameters.* If parameters are declared to be of type variable in a procedure statement or a function statement, values may be sent to the procedure or function through these parameters and returned from the called subprogram through these parameters back to the calling program. This process is implemented by having the corresponding identifiers from the calling program and from the called subprogram both reference the same memory location, as illustrated in Figure 8.2.

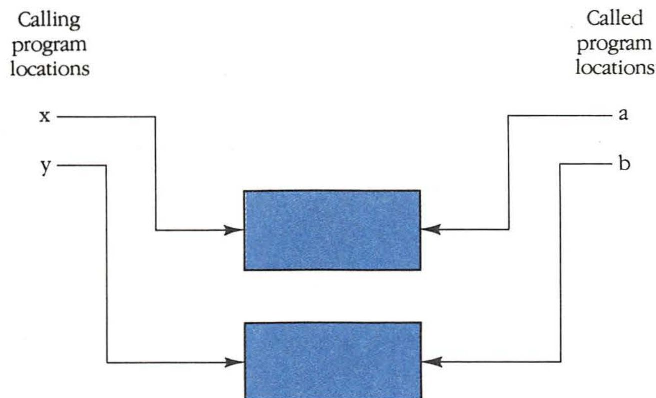


FIGURE 8.2

The identifiers x and y from the calling program and a and b from the called program both point to the same physical memory locations. Therefore, a change made in the contents of a in the called subprogram necessarily signifies a change in the contents of x, because they both "point to" the same physical memory location. A change in the contents of b also changes the contents of y. Likewise, a change in the contents of x and/or y signifies a change for a and/or b, respectively.

In order to distinguish between value and variable parameter, the keyword **var** must precede the declaration of any identifier in the parameter list that is to be a variable parameter. If **var** does not precede an identifier declaration, the identifier is assumed to be a value or one-way parameter.

Any local identifiers are declared following the procedure header and before the body of the procedure. The body of the procedure is enclosed between a **begin** and an **end**, with the **end** followed by a semicolon.

Examples:

```
procedure Withdraw(var bal:real;
        item : real);
 begin
  bal := bal - item
 end;
```

The procedure Withdraw accepts an account balance, bal, and an item, decreases bal by the value of item, and returns the new value of bal to the calling program. The procedure name Withdraw is not used to return any information. The identifier bal is a variable parameter. It accepts data from the calling program, and any change to bal made in Withdraw is reflected in a corresponding identifier in the calling program. Item is a value parameter. There is no need to send item information back to the calling program.

```
procedure RectInfo(tp,lt,bot,rt:integer;
        var ar,pr:integer);

 var
  len, wid : integer;
begin
 len := rt - lt;
 wid := bot - top;
 ar := len * wid;
 pr := 2 * len + 2 * wid;
 paintrect(tp, lt, bot, rt);
end;
```

The procedure `RectInfo` accepts data from the calling program through the value parameters `tp`, `lt`, `bot`, and `rt`. The identifiers `ar` and `pr` are variable parameters. They do not receive any useful data from the calling program but must be declared as variable so that data can be returned through these parameters. In addition to these six, `len` and `wid` are also local identifiers. The statements in the body of the procedure determine the length and width of the rectangle whose boundaries are given by `tp` (top), `lt` (left), `bot` (bottom), and `rt` (right). The area and perimeter are calculated, and these values are stored in the variable parameters `ar` and `pr` for return to the calling program. Before this return, the rectangle is drawn in the **Drawing** window. This is an example of one procedure, `RectInfo`, calling another, **paintrect**.

8.4 THE STRUCTURE OF A PROGRAM CONTAINING SUBPROGRAMS

You have seen some examples of functions and procedures standing by themselves. The task at hand now is to see how these pieces fit into an entire program, that is, (1) where subprograms are placed, (2) how they are called, and (3) how the identifiers in a subprogram are related to those of another subprogram, or even the main program.

THE PLACEMENT OF FUNCTIONS AND PROCEDURES

Functions and procedures are placed after declarations for the main program and before the body of the main program (see Figure 8.3).

Enclosing functions, procedures, the main program, and its declarations in boxes is done because it is useful to consider the entire program as being composed of blocks of code.

Subprogram blocks must be complete in that they contain their own declarations and documentation. The only major restriction is that a subprogram called by a statement must come before the block containing the statement that calls it.

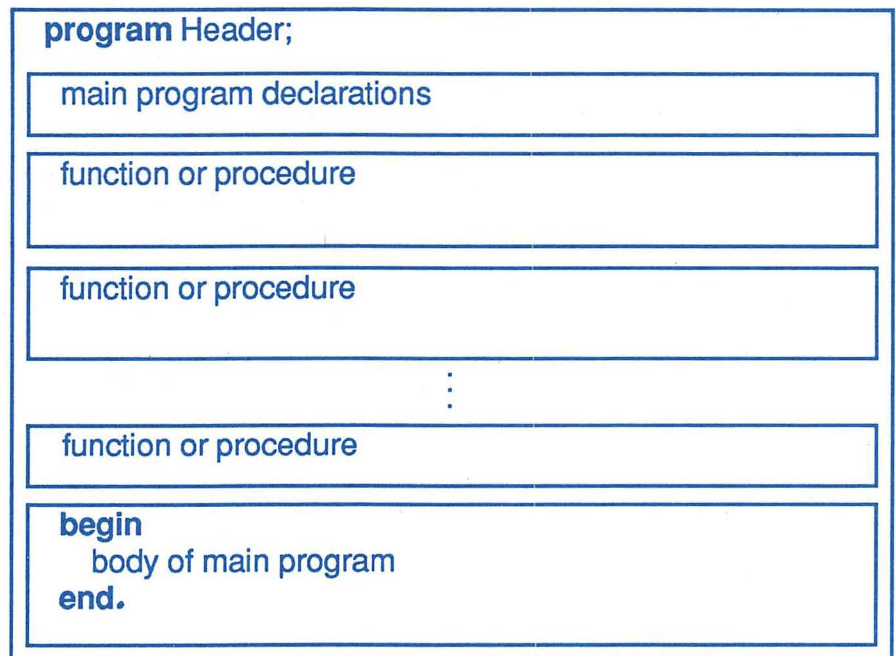


FIGURE 8.3

THE CALLING OF FUNCTIONS AND PROCEDURES

Programmer-defined subprograms are called in a manner similar to the way system-defined subprograms are called. First, let's consider a function call.

The call to a function must appear as part of another Pascal statement and must contain parameters that represent values sent to the function. Parameters in the call are known as *actual parameters.* The actual parameters must match, in number and type, those declared in the parameter list of the function being called and may be either actual values or identifiers representing actual values. Parameters in the called function are known as *formal parameters.* It is not necessary to declare a type for actual parameters in the function call, because these parameters should have already been declared in the calling program. The formal parameters have declared types in the function header of the called subprogram. An example should serve to clarify a function call.

Example:

```
function Diff(c, d:real):real;
 begin
  ...
 end;

begin
 ...
 x := Diff(12.5, b);
end.
```

The actual parameters, 12.5 and **b**, in the function call are only listed. **b** should have been declared in the program containing the function call. The result is passed back through the function identifier `Diff` and becomes the contents of **x** in the calling program. `Diff` has formal parameters **c** and **d**. The value 12.5 is sent to **c**, so 12.5 and **c** should be of the same type, **real**. Likewise, **b** and **d** should be of the same type since a value is sent through **b** to **d**. You cannot switch the order of **c** and **d**, since to do so might alter the results.

A call to a procedure is a statement in the calling block. The actual parameters must again match the formal parameters in the procedure header in both number and type. As with functions, it is necessary to list only the actual parameters, while the formal parameters in the called procedure have specific types declared.

Example:

```
procedure Mult(m,n : integer;
        var prod : integer);
 begin

  ...
 end;

begin
  ...
 Mult(r, s, ans);
  ...
end.
```

The actual parameters `r`, `s`, and `ans` are listed in the procedure call. Their types should have been declared in the calling program. The contents of `r` are passed to `m` and the contents of `s` to `n`. The result of the calculation in the called procedure produces a value for the variable formal parameter `prod`, which is then sent back to the calling program through the actual parameter `ans`.

THE SCOPE OF IDENTIFIERS

The scope of an identifier is the program block or blocks in which an identifier is considered to be legally declared. Since Pascal programs are composed of blocks, it is important to know where an identifier can be used and how it is related to other identifiers used in the same program.

Consider the program skeleton shown in Figure 8.4.

In the main program, four identifiers are declared: `a`, `b`, `c`, and `d`. Any identifier declared in the main program is also legally declared in all subprograms contained in that main program and is known as a *global identifier.* These four identifiers can be used in both `First` and `Second`.

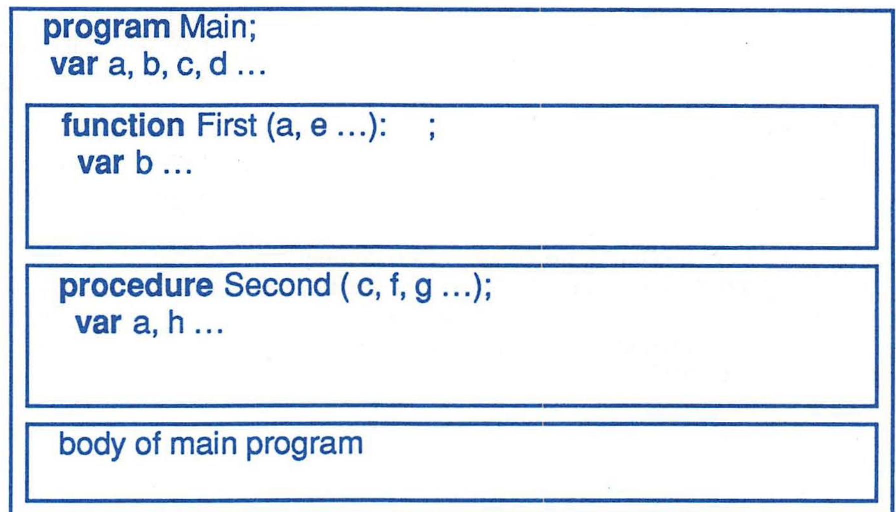


FIGURE 8.4

But hold on a minute! First has its own a and b declared, a in the function heading and b in the body of the function. Second has its own a and c declared. There are now three declarations for a, and two for b and c. Doesn't the system get confused? Well, declaring an identifier in a block overrides the global scope of that identifier, and it is said to have *local scope.* The identifier a has three different meanings in this program: one in the main program, one in First, and another in Second. In fact, it is not even necessary that all the a's be of the same type, because each of these identifiers refers to a different location in main memory. Similar arguments can be used for b and c.

The identifier e is declared in First only and has no meaning outside of this function. If e were used in the main program or Second, an error message would result, because it is considered an undeclared identifier in these blocks. Also, the identifiers f, g, and h in Second are legal only in Second.

The identifier d is declared in the main program and can be used in both First and Second without any ambiguity.

To help clarify the distinction between global and local identifiers, consider the complete program shown in Figure 8.5.

The identifier b is declared in both the main program and the function Change. When used in Change, it is local to that function. Any changes made to b do not affect the identifier b declared in the main program. The output of this program illustrates this point. The value 12 is entered for b in the main program. The function call passes this value of b to the parameter b in the header line of Change. This value is increased by 3, yielding 15, and this value is stored in the location represented by the local b. The value of b in the main program has not been changed since the global b refers to a different memory location. This changed value of b is multiplied by 4 and the result, 60, is "stored" in Change for return to the main program and stored in the location represented by the global identifier c. The contents of the locations represented by the main program b and c are displayed, indicating that the global b has the original value entered by a user while the global c has a value that was determined by the changed value of b local to the Change.

If global identifiers can be used all the time, why have local identifiers at all? Two of the major problems in programming are subprogram independence and maintenance. The same subprogram often can be used with many different programs. If that subprogram is written with global identifiers, all its identifiers must be changed to conform to the program of which it becomes a part. The subprogram "depends" on its main program. Program maintenance concerns the changing of a program or subprogram. The logic of each subprogram can be understood and changed without knowledge of the main program if that program contains local rather than global identifiers.

Figure 8.6 illustrates how the scope can affect variable parameters in procedures. The identifier x is given an initial value of 2, and the procedure TwoWay is called. Since the formal parameter x in TwoWay is a variable parameter, a value can be sent back to the main program through this parameter. The initial value of x is multiplied by 20, changing the local value of x to 40, its value in TwoWay is then displayed, and this new value is returned to the x declared in

```
program FunctionDemo;
{Demonstrates parameter passing using functions}

 var
  b, c      {test integers}
   : integer;

 function Change (b : integer) : integer;
  {Changes the value of the passed parameter}
  {b  :  local test integer for function Change}
 begin
  writeln('Now in function change');
  writeln('   Original value of b is ', b : 2);
  b := b + 3;
  writeln('   Changed value of b is ', b : 2);
  writeln;
  Change := b * 4
 end;

begin
 writeln('In main program');
 write('   Enter a number:  ');
 readln(b);
 writeln;
 c := Change(b);
 writeln('Meanwhile, back in main program');
 writeln('   The value of b is ', b : 2);
 writeln('   The value of c is ', c : 2);
end.
```

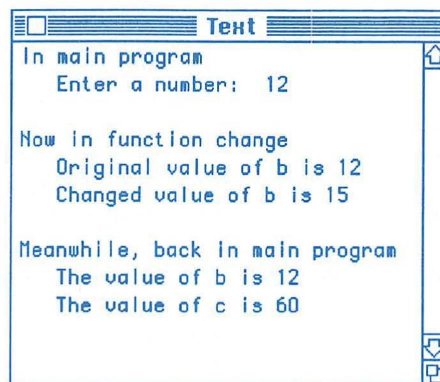


FIGURE 8.5

```
program ProcedureDemo;
{Demonstrates parameter passing in procedures}

 var
  x   {test integer}
   : integer;

 procedure OneWay (x : integer);
  {Demonstrates use of formal parameter}
  {x  :  local test integer in OneWay}
 begin
  x := x * 10;
  writeln('   The value of x in OneWay is ', x : 3)
 end;

 procedure TwoWay (var x : integer);
  {Demonstrates use of variable parameter}
  {x  :  local test integer in TwoWay}
 begin
  x := x * 20;
  writeln('   The value of x in TwoWay is ', x : 3)
 end;

begin
 x := 2;
 writeln('The initial value of x in the main program is ', x : 3);
 TwoWay(x);
 writeln;
 writeln('The value of x in the main program is now ', x : 3);
 OneWay(x);
 writeln;
 writeln('The value of x in the main program is now ', x : 3)
end.
```

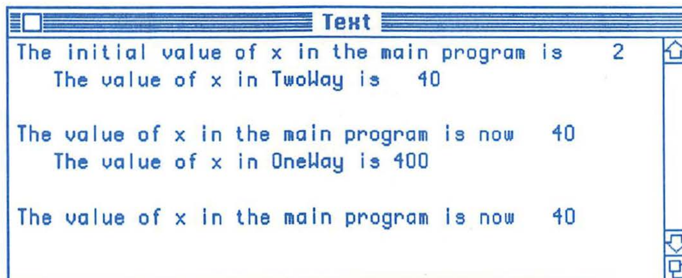


FIGURE 8.6

the main program. The value 40 is now displayed as it is seen by the main program.

The value 40 is then sent to `OneWay`. Procedure `OneWay` also has `x` declared as a local identifier, but it is not a variable parameter. Therefore, any change in `x` in this procedure cannot be sent back to `x` in the main program. So when `x`—currently 40—is multiplied by 10, the result (400) is local to `OneWay` as shown in the next output line, but the value of `x` in the main program has not been altered, as shown by the last line.

The confusion between identical identifiers in different blocks can be avoided by using different identifier names for different quantities whenever possible.

8.5 PROBLEM SOLUTIONS USING FUNCTIONS AND PROCEDURES

The problem-solving methodology developed in Chapter 3 can be applied, with slight modifications, to programs containing functions and procedures. Several examples are used to illustrate this revised process. Some of the subprograms are very short. It may not seem worth the trouble creating such a subprogram. The intent is to illustrate how functions and procedures are integrated into programs. By using smaller subprograms, we emphasize the interconnection and not the code. Once you have become familiar with using functions and procedures, you can concentrate on how to subdivide a program without worrying about how it is to be done.

The IPO chart is presented before the N-S chart, as was done in Chapter 3. In practice, these charts are completed simultaneously. That is, decide on identifiers, fill out the IPO chart, and discover when trying to design the solution using an N-S chart that you need a function or procedure. Then, you have to create an IPO chart for that subprogram and design its structure. So, although the completed IPO charts are presented before the completed N-S charts, this order doesn't reflect the exact chronological process employed in their creation.

As with program identifiers, programmer-defined procedure and function identifiers in this text begin with uppercase letters.

Problem: Write a program to determine the amount of an investment accumulating simple interest given the initial principal, the number of years the principal has been accumulating interest, and the annual interest rate. Create one function to change the input interest rate from a percentage to a decimal and another to calculate the amount of the investment. The formula used to determine the amount is

$$\text{amount} = \text{principal} * (1 + \text{years} * \text{rate})$$

Each block in the program (the main program and two functions) has its own set of input, processing, and output identifiers, so each has its own IPO chart. To distinguish between the IPO charts for the various blocks, a name and type section is included, as illustrated in Figure 8.7.

IPO CHART

Main Program **Function** **Procedure** (circle one)

Name____

Class	Identifier	Description
INPUT		
PROCESSING		

	Identifiers		Drawing
OUTPUT			

FIGURE 8.7

Functions and procedures called by a program or subprogram are included in the processing section of the IPO chart for that program or subprogram.

The IPO charts for the solution to the stated problem are shown in Figure 8.8.

The value parameters for each of the two functions are included in the input sections of the chart, because their values are input to the functions from the calling program. The function names **DecimalRate** and **Amount** are categorized as processing identifiers since these quantities are calculated in their respective functions. Output is performed from within the main program only, so the IPO chart for **SimpleInterest** is the only one that lists output identifiers.

N-S charts must also indicate functions and procedures. The box illustrated in Figure 8.9 shows how a subprogram call is to be represented.

The double-walled box containing a description of the purpose of the subprogram is followed by the name given to the function enclosed within parentheses. The subprogram itself constitutes another block. Each of the blocks relating to the same program is labeled for easy identification. Figure 8.10 shows the N-S chart for the program under consideration.

After the principal and interest rate are entered, the rate is changed to a decimal by calling **DecimalRate.** This function returns the result to the main program. After the number of years is entered, this rate, along with the principal and number of years, is sent to **Amount.** This function uses the equation given earlier to find the amount and sends it back to the main program, where it is sent to the **Text** window for display.

A listing and a run of the program are shown in Figure 8.11.

Each block is individually documented with an explanation of what that block does and a description of each identifier used in that block. Parameter descriptions are shown under function headers. Although none is necessary, a blank line is left between each of the blocks to emphasize the structure of the program.

Problem: Write a program to accept the combination to a safe in the form *LXXRXXLXX,* where the *X*'s represent single digits, and a new middle term. Create and display a new combination with the new middle term replacing the old one. Use a function **Change** that accepts the old combination and the new middle term and returns the new combination.

The IPO charts for the main program (**SafeCombination**) and the function (**Change**) are shown in Figure 8.12. Figure 8.13 depicts the N-S chart of the solution.

The main program contains three identifiers and the string function. Different identifier names were chosen for identical quantities in both the main program and function, although the same names could have been used. This choice was made because it is thus easier to describe the use of parameters in the passing of values between blocks of a program.

After the old combination and new middle term have been entered, the main program calls **Change** and passes to it the values of the old combination and the new middle term so that it can find the new combination. The contents

IPO CHART

(Main Program — circled) Function Procedure (circle one)

Name: Simple Interest

Class	Identifier	Description
INPUT	p	principal invested
	r	annual interest rate in %
	n	no. of years principal is invested
PROCESSING	r	interest rate in decimal form
	a	amount accumulated
	Decimal Rate	(function)
	Amount	(function)

OUTPUT	Identifiers		Drawing
	p		
	r		
	n		
	a		

FIGURE 8.8

IPO CHART

Main Program (Function — circled) Procedure (circle one)

Name: Decimal Rate

Class	Identifier	Description
INPUT	percentage	interest rate in %
PROCESSING	Decimal Rate	interest rate in decimal form

OUTPUT	Identifiers		Drawing

IPO CHART

Main Program (Function — circled) Procedure (circle one)

Name: Amount

Class	Identifier	Description
INPUT	p	principal invested
	r	interest rate
	n	number of years invested
PROCESSING	Amount	amount accumulated

OUTPUT	Identifiers		Drawing

FIGURE 8.9

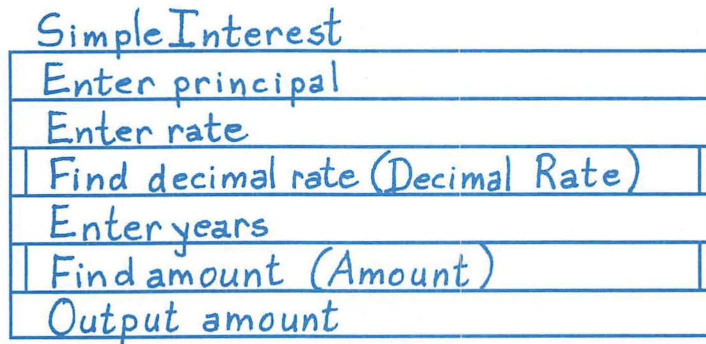


FIGURE 8.10

of the locations represented by local variables `p`, `len`, `first`, and `last` are accessed only in the function. It is not necessary for the main program to "see" these values, because only the final result (`Change`) is sent back to the main program. After receiving that value, the main program displays the new combination.

In `Change`, the position of the `R` is used to separate the first part of the string combination from the second. The first part remains unchanged, so it is saved in the location represented by `first`. The **length** function is used to find the number of characters in the string so that the last three characters can be removed from the old combination. These three characters also remain intact and are saved in the location represented by the identifier `last`. Finally, the contents of `first`, the new middle term (`term`), and `last` are concatenated in that order and the result sent back to the main program through `Change`. In this case, system-defined string functions (**pos**, **length**, and **copy**) are used

```
program SimpleInterest;
{Determines amount of an investment accumulating simple}
{interest at a given annual rate}

 var
  p,          {principal invested}
  r,          {annual interest rate in percent}
  a        {amount accumulated}
   : real;
  n        {number of years principal is invested}
   : integer;

 function DecimalRate (percentrate : real) : real;
  {Determines interest rate in decimal form}
  {percentage : rate in percent}
 begin
  DecimalRate := percentrate / 100
 end;

 function Amount (p, r : real;
          n : integer) : real;
  {Determines amount accumulated}
  {p : principal}
  {r : rate}
  {n : number of years}
 begin
  Amount := p * (1 + n * r)
 end;

begin
 write('Enter principal invested:  $');
 readln(p);
 write('Enter annual interest rate (in percent):  ');
 readln(r);
 r := DecimalRate(r);
 write('Enter number of years:  ');
 readln(n);
 a := Amount(p, r, n);
 writeln('The accumulated amount is $', a : 7 : 2)
end.
```

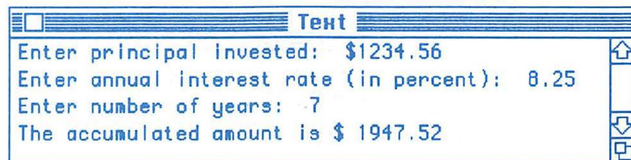


FIGURE 8.11

IPO CHART

(Main Program) Function Procedure (circle one)

Name: Safe Combination

Class	Identifier	Description
INPUT	old	old combination
	add	new middle term
PROCESSING	new	new combination
	Change	(function)

OUTPUT	Identifiers		Drawing
	old		
	add		
	new		

IPO CHART

Main Program (Function) Procedure (circle one)

Name: Change

Class	Identifier	Description
INPUT	combo	old combination
	term	new middle term
PROCESSING	p	position of "R" in combination
	len	no. of characters in combination
	first	first part of new combination
	last	last part of new combination
	Change	new combination

OUTPUT	Identifiers		Drawing

FIGURE 8.12

Safe Combination

Safe Combination
Enter old combination
Enter new middle term
Find new combination (Change)
Display new combination

Change

Change
Find position of "R"
Extract first part of combination
Find length of combination
Extract last part of combination
Put new combination together

FIGURE 8.13

inside the programmer-defined function **Change** to find the new combination. Once the new combination has been received by the main program, it is displayed.

A documented listing and sample run of the final solution are given in Figure 8.14.

```
program SafeCombination;
{Changes the combination to a safe (in the form LXXRXXLXX)}
{by altering the middle term.}

 var
  old,          {old combination}
  add,          {new middle term}
  new        {new combination}
   : string;

 function Change (combo, term : string) : string;
  {Changes the combination}
  {combo : old combination}
  {term : new middle term}
  var
   p,             {position of 'R' in combination}
   len        {number of characters in combination}
    : integer;
   first,         {first part of new combination}
   last       {last part of new combination}
    : string;
 begin
  p := pos('R', combo);
  first := copy(combo, 1, p);
  len := length(combo);
  last := copy(combo, p + 3, len + 1 - (p + 3));
  Change := concat(first, term, last)
 end;

begin
 write('Enter old combination in the form LXXRXXLXX:  ');
 readln(old);
 write('Enter new two-digit middle term:  ');
 readln(add);
 new := Change(old, add);
 writeln('The new combination is ', new)
end.
```

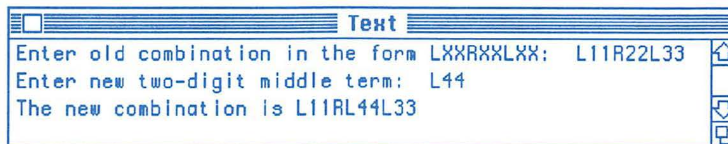


FIGURE 8.14

Problem: Write a program to draw the six different faces of a die. Use a different procedure for each face.

The solution process is fairly straightforward. You just have to determine the coordinates of the arguments of the graphic procedures used to draw the faces. At this stage in programming, using separate identifiers for each set of arguments is too time consuming, so constant values are used. As a result, the IPO charts for the main program and six procedures are almost trivial (see Figure 8.15).

Likewise, the N-S chart for the main program contains only calls to the six procedures (see Figure 8.16).

The main program `DieFace` merely calls the six drawing procedures (indicated by placing the procedure names in the processing portion of the chart), while each of the IPO charts for the procedures contains a description of what is to be drawn in the **Drawing** section of the output portion of the chart.

Each drawing procedure involves drawing the outline of the die and then filling in the appropriate number of dots in the appropriate locations. The actual arguments used for this process do not have to be shown here since the N-S chart serves as an outline of the solution.

A complete listing and run of the solution are shown in Figure 8.17.

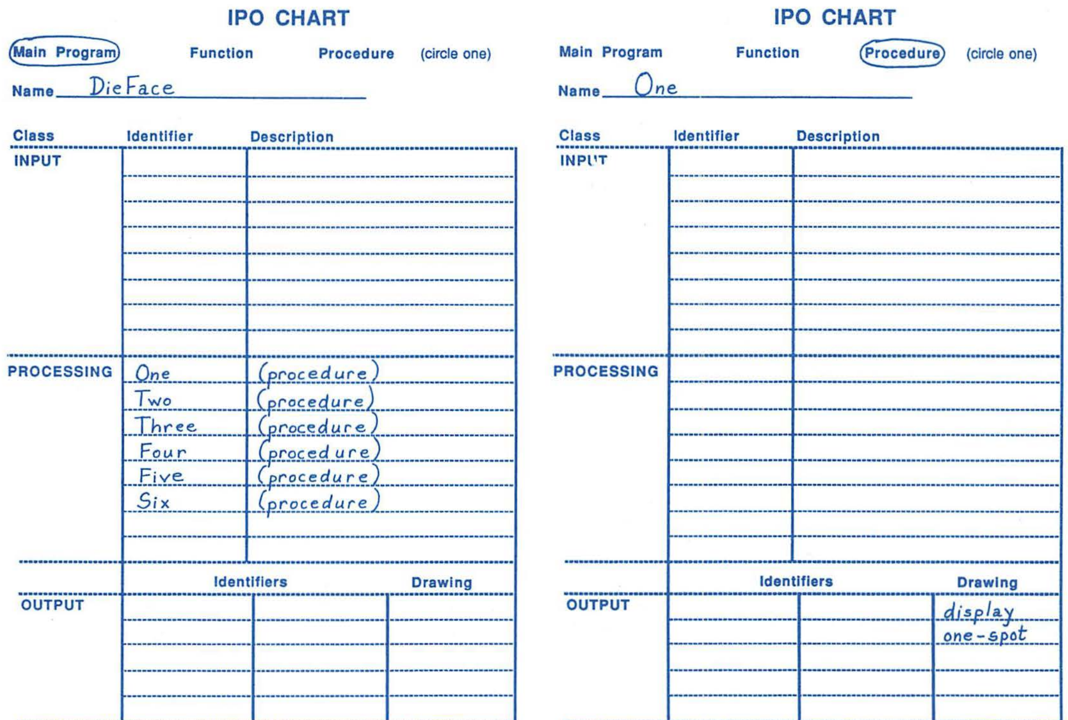


FIGURE 8.15 *(continued)*

IPO CHART

Main Program   Function   (Procedure)   (circle one)

Name Two

Class	Identifier	Description
INPUT		
PROCESSING		

OUTPUT	Identifiers		Drawing
			display two-spot

IPO CHART

Main Program   Function   (Procedure)   (circle one)

Name Three

Class	Identifier	Description
INPUT		
PROCESSING		

OUTPUT	Identifiers		Drawing
			display three-spot

FIGURE 8.15 *(continued)*

IPO CHART

Main Program   Function   (Procedure)   (circle one)

Name Four

Class	Identifier	Description
INPUT		
PROCESSING		

OUTPUT	Identifiers		Drawing
			display four-spot

IPO CHART

Main Program Function (Procedure) (circle one)

Name Five

Class	Identifier	Description
INPUT		
PROCESSING		

	Identifiers		Drawing
OUTPUT			display five-spot

IPO CHART

Main Program Function (Procedure) (circle one)

Name Six

Class	Identifier	Description
INPUT		
PROCESSING		

	Identifiers		Drawing
OUTPUT			display six-spot

FIGURE 8.15

DieFace

Display one-spot (One)
Display two-spot (Two)
Display three-spot (Three)
Display four-spot (Four)
Display five-spot (Five)
Display six-spot (Six)

One

Draw boundary rectangle
Draw one dot

Two

Draw boundary rectangle
Draw two dots

Three

Draw boundary rectangle
Draw three dots

Four

Draw boundary rectangle
Draw four dots

Five

Draw boundary rectangle
Draw five dots

Six

Draw boundary rectangle
Draw six dots

FIGURE 8.16

```
program DieFace;
{Displays 6 different faces of a die}

 procedure One;
  {Displays one-spot}
 begin
  framerect(10, 10, 70, 70);
  paintoval(35, 35, 45, 45)
 end;

 procedure Two;
  {Displays two-spot}
 begin
  framerect(10, 80, 70, 140);
  paintoval(15, 85, 25, 95);
  paintoval(55, 125, 65, 135)
 end;

 procedure Three;
  {Displays three-spot}
 begin
  framerect(80, 10, 140, 70);
  paintoval(85, 15, 95, 25);
  paintoval(105, 35, 115, 45);
  paintoval(125, 55, 135, 65)
 end;

 procedure Four;
  {Displays four-spot}
 begin
  framerect(80, 80, 140, 140);
  paintoval(85, 85, 95, 95);
  paintoval(125, 85, 135, 95);
  paintoval(85, 125, 95, 135);
  paintoval(125, 125, 135, 135)
 end;

 procedure Five;
  {Displays five-spot}
 begin
  framerect(150, 10, 210, 70);
  paintoval(155, 15, 165, 25);
  paintoval(195, 15, 205, 25);
  paintoval(175, 35, 185, 45);
  paintoval(155, 55, 165, 65);
  paintoval(195, 55, 205, 65)
 end;
```

FIGURE 8.17 *(continued)*

```
  procedure Six;
   {Displays six-spot}
  begin
   framerect(150, 80, 210, 140);
   paintoval(155, 85, 165, 95);
   paintoval(175, 85, 185, 95);
   paintoval(195, 85, 205, 95);
   paintoval(155, 125, 165, 135);
   paintoval(175, 125, 185, 135);
   paintoval(195, 125, 205, 135)
  end;

begin
 One;
 Two;
 Three;
 Four;
 Five;
 Six
end.
```

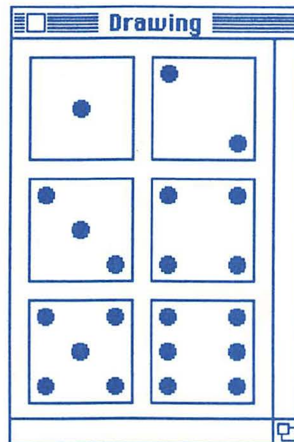


FIGURE 8.17

Each procedure uses the **framerect** procedure to outline the square that serves as the die face. The procedures then use **paintcircle** to put the correct number of dots inside each square. The arguments used in each procedure can be determined by using graph paper to map out the drawing before the actual values for the arguments are entered.

It is worthwhile to note that no values are sent to or received from each of the six procedures in the program. A function cannot be used, because at least one value must be returned to the calling program.

Problem: Write a program to accept the total sales of a product for the year 1984 and the total sales for the year 1985, and find the change in sales and the percent difference in the sales between those two years. The change is determined by subtracting the 1984 sales total from the 1985 sales total, while the percent difference is the absolute value of the change divided by the 1984 sales total and the result multiplied by 100.

Figure 8.18 shows the IPO chart for the solution and Figure 8.19 the N-S chart.

The data must be entered, the results calculated, and the answers displayed. This process lends itself to a division into three procedures to implement each one of the previously mentioned processes. The main program merely calls each of the procedures in succession.

The procedure `DataEntry` accepts the necessary data for the calculations. Since the data to be entered must apply to the entire program, no local identifiers are declared. The identifiers used to represent locations for input data are global and valid in all other procedures in the program.

The procedure `Calculate` finds the change and the percent difference. As you can see from the N-S chart, this procedure calls a function `Pd` to find the percent difference. Although the calculation of this quantity could have just as

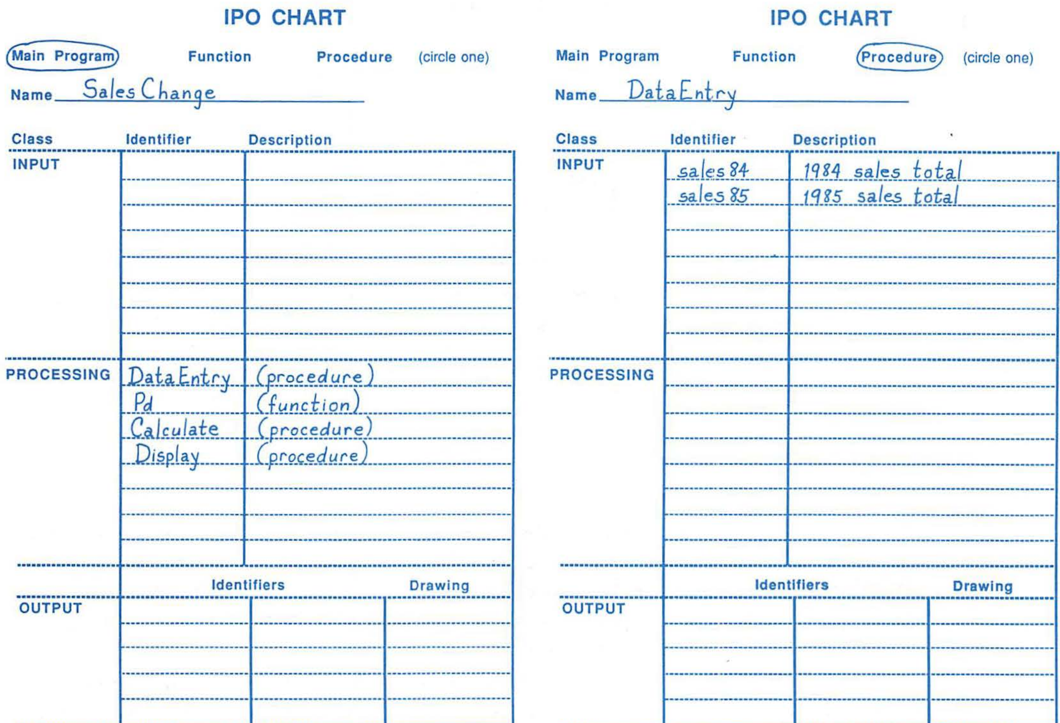


FIGURE 8.18 *(continued)*

IPO CHART

Main Program — **Function** (circled) — Procedure (circle one)

Name: Pd

Class	Identifier	Description
INPUT	chg	increase or decrease in sales
	prevtotal	sales total for previous year
PROCESSING	Pd	percent difference

OUTPUT	Identifiers		Drawing

FIGURE 8.18

IPO CHART

Main Program — Function — **Procedure** (circled) (circle one)

Name: Calculate

Class	Identifier	Description
INPUT	total84	1984 sales total
	total85	1985 sales total
PROCESSING	chg	increase or decrease in sales
	difference	percent difference

OUTPUT	Identifiers		Drawing

IPO CHART

Main Program — Function — **Procedure** (circled) (circle one)

Name: Display

Class	Identifier	Description
INPUT		
PROCESSING		

OUTPUT	Identifiers		Drawing
	change		
	perdiff		

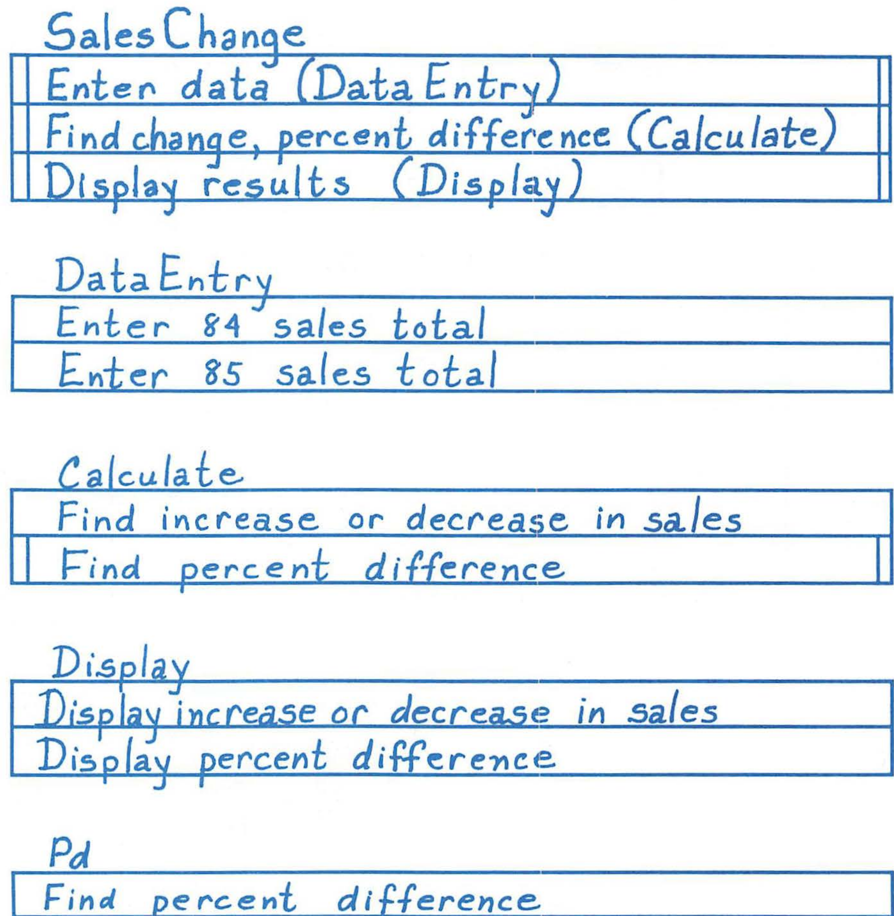


FIGURE 8.19

easily been done in `Calculate`, it is our intention to show a case of a procedure calling a function. Different identifiers are used in these subprograms so that you can more easily follow the passing of the values between the subprograms. Remember, since `Calculate` calls `Pd`, `Pd` must precede `Calculate` in the structure of the program.

The procedure, `Display`, shows the contents of `Change` and `perdiff`, the global identifiers used by the program to represent the change and percent difference.

A complete listing and a sample run of the solution to this problem are shown in Figure 8.20.

```
program SalesChange;
{Determines change in sales and percent difference for sales}
{in two consecutive years}

 var
  sales84,             {1984 sales total}
  sales85,             {1985 sales total}
  change,               {increase or decrease in sales}
  perdiff              {percent difference of sales between 1984 and 1985}
   : real;

 procedure DataEntry;
  {Input required values}
 begin
  write('Enter 1984 sales total:  ');
  readln(sales84);
  write('Enter 1985 sales total:  ');
  readln(sales85)
 end;

 function Pd (chg, prevtotal : real) : real;
  {Calculate percent difference}
  {chg : increase or decrease in sales}
  {prevtotal : sales total for previous year}
 begin
  pd := abs(chg) / prevtotal * 100
 end;

 procedure Calculate (total84, total85 : real;
         var chg, difference : real);
  {Calculate change and percent difference}
  {total84 : 1984 sales total}
  {total85 : 1985 sales total}
  {chg : increase or decrease in sales}
  {difference : percent difference}
 begin
  chg := total85 - total84;
  difference := Pd(chg, total84)
 end;

 procedure Display;
  {Display results}
 begin
  writeln('The change in sales is ', change : 7 : 2);
  writeln('The percent difference between');
  writeln('   1984 and 1985 sales is ', perdiff : 5 : 2)
 end;

begin
 DataEntry;
 Calculate(sales84, sales85, change, perdiff);
 Display
end.
```

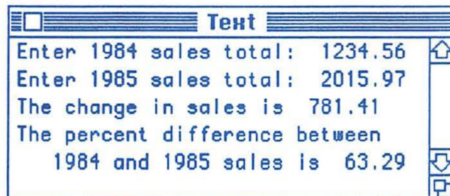


FIGURE 8.20

Before continuing, you should thoroughly review the four programs presented in this section, paying special attention to the structure of the programs and how values are passed from one block to another. The idea of breaking a large program up into smaller, more manageable parts is used extensively throughout the remainder of the text.

8.6 TYPICAL PROGRAMMING ERRORS

The following types of errors may crop up in programming:

1. Definition of a function or procedure after its call.

Example:
```
procedure A(              );
 begin
  B(          )
 end;

procedure B(              );
```

2. Failure to declare a type for a function identifier.

Example:
```
function Change(              );
```

3. Declaration of a type for a procedure identifier.

Example:
```
procedure Static(              ) : real;
```

4. Failure to end a function or procedure with a semicolon.

Example:
```
procedure C(              );
 begin
  ...
 end.
```

5. Improper function call.

Example:
```
function Test(              ) : integer;
 begin
  ...
 end;

begin
   ...
 Test(          );
   ...
end.
```

6. Improper procedure call.

Example:
```
procedure D(            );
 begin
  ...
 end;
```

```
begin
 ...
 x := D(            );
 ...
end.
```

7. Improper passing of parameters by mixing order and/or type.

NONPROGRAMMING EXERCISES

1. Classify the following Pascal statements as valid or invalid. If the statement is invalid, explain why.
   a. `function M(c, d : string);`
   b. `procedure R(var a : integer);`
   c. `function S(var f : real) : integer:`
   d. `function Third;`
   e. `procedure Parte(c, d : integer; var f : real) : boolean;`
   f. `procedure Seven;`
   g. `procedure var quest;`
   h. `function Answ(a, b : real) : string;`
2. Walk through the following programs and give the exact output produced by each program.
   a.
```
program One;
 var
  a, b : integer;

 function C(a : integer) : integer;
  begin
   a := a * 2;
   writeln('a in function C equals ',a:2);
   C := a
  end;

begin
 a := 6;
 b := C(a);
 writeln('a in the main program equals ',a:2);
 writeln('b in the main program equals ',b:2)
end.
```
   b.
```
program Two;
 var
  x, y : string;

procedure W(var r : string);
 begin
  writeln('r in procedure W before the change is ',r);
  delete(r, 2, 3);
  writeln('r in procedure W after the change is ',r)
 end;
```

```
begin
 x := 'computer';
 writeln('x in the main program before the call is ',x);
 W(x);
 writeln('x in the main program after the call is ',x)
end.
```

c.
```
program Three;
 var
  x, y : integer;

 function A(c : integer) : integer;
  var
   b : integer;
  begin
   b := c * 2;
   writeln('b in function A equals ',b:2);
   A := b
  end;

 procedure D(f : integer; var g : integer);
  var
   h : integer;
  begin
   h := f + 2;
   g := A(h);
   writeln('g in procedure D equals ',g:2)
  end;

begin
 x := 5;
 D(x, y);
 writeln('y in the main program equals ',y:2)
end.
```

d.
```
program Four;
 var
  x, y : string;

 procedure A(c : string; var d : string);
  begin
   d := copy(c, 3, 6);
   writeln('d in procedure a equals ',d)
  end;

 function B(m : string) : string;
  var
   n : string;
  begin
   A(m, n);
   B := concat(n, 'able');
   writeln('B in function B equals ',B)
  end;
```

```
begin
 x := 'independence';
 y := B(x);
 writeln('y in the main program equals ',y)
end.
```

3. Write a function or procedure to accomplish each of the following tasks.
   a. To accept two numbers and return their product.
   b. To accept two numbers and return both their sum and difference.
   c. To choose and return a random integer between 1 and 3.
   d. To accept a value of x and return the corresponding function value for y if $y = 3x^2 + 2x - 5$.
   e. To accept 3 numbers and return their average.
   f. To accept the top, left, bottom, and right boundaries of a rectangle and display that rectangle with the left half shaded and the right half empty.
   g. To accept three words and return a single word comprised of the first letters of each.
   h. To accept the coordinates of the center of a circle and its radius, display the shaded circle with these parameters, and put a circular hole in the middle of it where the radius of the hole is half the radius of the painted circle.

PROGRAMMING EXERCISES

(B) 4. Given the number of shares of a stock held by an investor and the current price per share, write a program to find and display the total worth of this stock. Use separate subprograms to enter the data, find the total worth of the stock, and display the results.

(B) 5. Write a program to find and display the selling price of the item given the cost of producing an item and the desired markup. Use separate subprograms to enter the data, find the selling price, and display the results.

(G) 6. The Fahrenheit temperature on a given night can be determined by counting the number of chirps that a cricket makes in a minute. The formula that gives the Fahrenheit temperature is

$$t = (n + 160) / 40$$

where t is the Fahrenheit temperature and n the number of chirps per minute. Write a program that accepts the number of chirps per minute and displays the corresponding Fahrenheit temperature. Use separate subprograms to enter the data, find the temperature, and display the temperature.

(G) 7. Write a program that contains a function that finds the cube of a number (a number raised to the power 3), and use it to find the volume of a sphere given the sphere's radius. The formula for the volume of a sphere is

$$V = 1.33 \pi r^3$$

where π is approximately 3.1416 and r is the radius of the sphere.

(G) 8. Write a program that accepts the speed of a satellite at a particular altitude in miles per hour and determines the time, in hours, for one complete pass of that satellite around Earth. Use the following formula:

$$t = 1600000 / s$$

where s is the speed in miles per t the time in minutes. Use separate subprograms to enter the data, find the time for a pass, and display the results.

(B) 9. Write a program that congratulates the winner of a contest, using a form letter, by accepting the name and address of the winner and by placing that information in appropriate places in a congratulatory note consisting of string constants. Use separate subprograms to enter the name and address and to print the note.

(B) 10. Write a program that accepts the name of an employee and that employee's net pay and prints a check with this information in appropriate places. Use separate subprograms to accept the data and to print out the check.

(B) 11. Write a program to determine and display the monthly, biweekly, and weekly take-home pay options for an employee given that employee's gross annual salary. Assume that the only deduction is 15 percent for federal tax. Use a function that accepts the number of pay periods per year and returns the net pay per pay period.

In each of the following programs, you should use at least one function or procedure. Although it may be tedious to do this now, it makes it easier to design and implement more complex programs at a later date.

(B) 12. The economic order quantity is the most cost-efficient number of items to manufacture for a given order. It is determined by

$$Q = \sqrt{\frac{2RS}{C}}$$

where Q is the economic order quantity, R is the annual number of units required, S is the setup cost for each order, and C is the cost of storage of one unit of the item manufactured for one year. Write a program that accepts R, S, and C and finds Q. Make sure your answer is rounded, because the answer should be an integer to make sense.

(G) 13. Write a program to draw a shaded circle, accept horizontal and vertical translation parameters, erase the circle, and redraw it at the new position determined by adding its translation parameters to the original location's coordinates.

(G) 14. Write a program to accept the miles-per-gallon rating and tank capacity of a car, and find and display the driving range of the car (the product of the mpg rating and the tank capacity). Also find the cost of a complete tank of gas if gas sells for $1.29 per gallon.

(B) 15. The Other Foot Shoe Company offers its starting salespeople a choice of payments. A salesperson can receive either (a) a weekly base salary of $220 plus a 12 percent commission on all sales; or (b) an hourly salary of $4.50 per hour, for a 40-hour week, and a 14 percent commission for sales. If a new salesperson averages sales of $1,000 per week, write a program to display the total pay under each plan, for this person.

(B) 16. Customers of the Bright Light Electric Company are charged at a rate of $0.075 per kilowatt-hour (kWh) of energy used. Write a program that accepts the number of kWh used by a customer and determines the cost of electricity. The cost must be rounded to two decimal places. Display a computer-generated bill from all the pertinent information.

(B) 17. Joe Ficksit would like to redo his den. He would like to put carpeting on the floor, and paint the four walls and ceiling. If the length of his den is 20 feet, the width 15 feet, and the height 8.5 feet, determine how much it would cost to redo his room if a gallon of paint costs $8.99 and covers 400 square feet of wall and the carpeting

costs $8.99 a square yard. The room has one door that measures 7 feet by 2.5 feet and one window that measures 5 feet by 3 feet.

(G) 18. Dealin' Sam, a car dealer, surveyed his customers as to whether they would buy a new car within the next three years or not. Write a program that accepts the number of yes and no responses and show the results of the survey using a horizontal bar graph similar to the one shown in Figure 7.37 of Chapter 7.

(B) 19. The American Survey Company has been commissioned to survey the public on which brand of ginger ale it prefers. Each questionnaire contains three choices: Desert Dry, Sweet and Sassy, and Other. Write a program that accepts the number of respondents preferring each and draw a vertical bar graph (similar to the one in Figure 7.35 of Chapter 7) illustrating the results of the survey.

(G) 20. Write a program that accepts two sets of x and y coordinates for points in a plane and plots the points connecting them with a straight line. Make sure your graph is properly labeled.

(B) 21. The Mail Order Palace receives orders for its unique left-handed forks. Each order contains a customer's name, address, and number of forks desired. If each fork costs $0.79, write a program that finds the total cost of the order, adds 2 percent for shipping and handling, and displays an invoice showing all pertinent information.

(G) 22. The general form of a linear equation is $Ax + By + C = 0$, where A, B, and C are constants. Write a program that accepts the three coefficients and prints the slope of the line, its x intercept, and its y intercept. Assume the values entered for A, B, and C are nonzero.

(G) 23. Write a program that draws a shaded circle in the middle of the **Drawing** window, accepts a size factor, erases the original circle, and redraws the circle taking the size factor into account. The size factor is multiplied by the radius to change the size of the circle. For example, a size factor of 1 would leave the circle unchanged. Values greater than 1 would enlarge the size of the circle, and values less than 1 would reduce the size of the circle. Also determine and display the increase in area caused by this size factor. The area of a circle is the product of 3.1416 (π) and the square of the radius of the circle.

C H A P T E R 9

Choices

9.1 INTRODUCTION

People make decisions every day. Decisions may involve taking some action or not, such as to paint an old car or not. Decisions may also involve selecting between different courses of action. For example, you may have a choice among the following: repair an old car, lease a car, or buy a new car.

Computers also make decisions. The introduction of decision logic into a program allows execution along different paths in that same program. The ability to make choices from within a program permits you to design and code solutions to problems that you could not have solved before. This chapter describes how selection options can be employed in writing MacPascal programs that use conditional statements.

9.2 ALTERING THE FLOW OF CONTROL

The normal flow of control in a program is from one instruction to the next, in sequence. That is, after an instruction line is executed, processing control automatically proceeds to the next instruction line. However, in problems where choices must be made during the running of a program, a new programming structure is required. This structure permits you to change the sequential order of statement execution and to continue execution at any of several places in the program, and is known as a *selection structure*.

A selection structure directs the flow of control along alternative paths called *branches*. The path that the program follows is determined by a condition or conditional expression that produces a Boolean value.

MacPascal contains several statements that can be used to implement a selection structure. In one type, if the condition is true, a single option is selected. If the condition is false, the program continues in sequential fashion as if the

decision structure were not even there. We call this type of structure a *single-option structure.* For example, you would travel along a highway directly toward your destination unless you encountered a detour.

Another selection structure is called the *double-option structure.* If the condition is true, one option is selected; if the condition is false, a different option is selected. For example, a road may branch into two paths, each of which can get you to your final destination.

A *multi-option structure,* containing many branches, is also available. In this structure, an expression produces a value that permits the transfer of control from one point in the program to any number of different locations. Continuing with the road example, you might come to a major intersection where several roads meet and one must be selected.

9.3 CONDITIONAL EXPRESSIONS

Since conditional expressions are essential parts of the selection structures just described, you should learn exactly what they are and what they are composed of.

RELATIONAL OPERATORS

The conditions contained in selection structures are frequently evaluated using relational operators. These operators compare expressions of the same type and produce Boolean values of true or false. The values that are compared are referred to as *operands.*

If A and B represent expressions of the same data type, they may be compared in six different ways. Each of these comparisons is listed in Table 9.1 with its associated MacPascal code.

The relational operator = is entirely different from the assignment operator (:=) and should not be confused with it.

Example:

If A is 10, B is 3, X is 2, and Y is 1, then the relationship (A + X − Y) >= (B * X) is true since 11 >= 6.

TABLE 9.1

RELATIONSHIP	*MACPASCAL CODE*
A equal to B	A = B
A not equal to B	A <> B
A greater than B	A > B
A less than B	A < B
A greater than or equal to B	A >= B
A less than or equal to B	A <= B

When a relational operator compares two strings, the American Standard Code for Information Interchange (ASCII Code) is used to determine the order of precedence. This code orders the characters on a standard keyboard according to a fixed numerical hierarchy. All the letters of the alphabet, both uppercase and lowercase, are arranged in the standard order, but the codes for the uppercase letters from *A* to *Z* precede the codes for the lowercase letters from *a* to *z*. Other keyboard characters are also included in the hierarchy. A complete listing of the ASCII Codes for keyboard characters may be found in Appendix C, at the end of this book.

A string comparison can be thought of as a simple alphabetization of the string. For example, if string identifier `A` contains `JASON` and another string identifier `B` contains `JAX`, then `A < B` is true, despite the fact that `A` is the longer of the two strings. The string `JASON` precedes the string `JAX` in alphabetical order. However, `JAX` precedes `jason` since the uppercase `J` precedes the lowercase `j` in the ASCII Code.

LOGICAL OPERATORS

Boolean values of true and false are produced by other operators called *logical operators*. These operators are specifically designed to combine Boolean values. MacPascal contains three such logical operators: **not**, **and**, and **or**. The first of these, **not**, is *unary*, since it operates on a single Boolean operand. The **not** operator reverses the Boolean value of the operand that follows it. For example, the expression `not (5 <= 3)` has a value of *true* since `(5 <= 3)` is *false* and **not** reverses that value.

The second of the logical operators, **and**, is *binary* (it combines two logical operands). Since each of the two operands has a value of either *true* or *false*, there are four possibilities to consider (see Table 9.2).

The only combination that produces the value true occurs when both operands are true. For example, if `answer` is of type Boolean and equivalent to `(3 > 1) and (8 * 2 <= sqr(4))`, then the value of `answer` is *true*.

The third logical operator, **or**, is also a binary operator. Again there are four possibilities to consider (see Table 9.3).

The only case in which this operator produces a Boolean value false is when both operands are false. For example, if `P` is of type Boolean, the expression `P`

TABLE 9.2

COMBINATION	*RESULT*
True and true	True
True and false	False
False and true	False
False and false	False

TABLE 9.3

COMBINATION	*RESULT*
True or true	True
True or false	True
False or true	True
False or false	False

TABLE 9.4 The Order of Precedence of Operators

LEVEL	*OPERATORS*
1.	not
2.	*, /, div, mod, and
3.	+, −, or
4.	=, <>, <, >, <=, >=

`or (3 < 4)` evaluates to *true* regardless of the value of `P`, since the **or** operator produces a *true* value as long as one of its operands is *true,* and 3 is always less than 4.

ORDER OF OPERATIONS

The order of precedence of numeric operators was treated in Chapter 6. A specified order also exists for relational and logical operators. Table 9.4 shows the order of precedence of MacPascal operators. It lists these operators from the highest order of precedence (1) to the lowest (4). Operators that appear on the same line have equal precedence. When MacPascal executes an instruction, it scans it from left to right, and expressions within parentheses are evaluated first. If nested parentheses are contained in an instruction, evaluation begins with the innermost set of parentheses.

Example:

Given that A = 3, B = 7, and C = 5, evaluate the following expression:

not((B * C) > (A + B)) or (A >= C) and (A = B)

Substituting the values into the expression gives

not((7 * 5) > (3 + 7)) or (3 >= 5) and (3 = 7)

Evaluating the inner parentheses yields

not(35 > 10) or (3 >= 5) and (3 = 7)

Evaluating the rest of the parentheses reduces the expression to

not(true) or (false) and (false)

The **not** operator has precedence over the other operators, and the expression now becomes

false or false and false

The **and** is used next since it has a higher precedence than **or**:

false or false

Using **or** gives a final value of false.

Careful use of parentheses is especially important in complex expressions involving many different operators to ensure that the intent of your expression is interpreted properly.

9.4 THE IF... THEN...ELSE STRUCTURE

We now show how to use conditional expressions in the creation of Pascal structures. We begin by considering how these expressions are used in single- and double-option statements.

THE SINGLE-OPTION STRUCTURE

The format for the single-option conditional structure is as follows:

```
FORMAT: if <condition> then
          <statement>;
```

The condition portion of the **if** is evaluated, and if its value is true, <statement> is executed. Program execution continues with the instruction immediately following <statement>. However, if the value of <condition> is false, the statement following **then** is ignored, and the program continues execution with the instruction immediately following <statement>. <statement> is followed by a semicolon indicating the end of this single-option structure.

Example:

```
if hours > 40 then
  write('overtime and ');
writeln('regular pay');
```

If the expression `hours > 40` is true, the statement following **then** is executed and the output of the program is `overtime and` immediately followed by `regular pay,` the next statement in the program. If `hours <= 40,` the **write** statement is ignored and execution continues with the **writeln** statement giving an output of only `regular pay.`

You may want to execute more than one statement if the condition is true. All you have to do is enclose those statements in a **begin-end** set and follow the end with a semicolon. A combination of statements enclosed by a **begin-end** set is called a *compound statement.*

Example:

```
if student = 'yes' then
 begin
  write('ft or pt? ');
  readln(status)
 end;
write('Are you employed? ');
readln(empl);
```

If the string identifier `student` has a value `yes,` the two lines after the **then** are executed followed by the employment query. If `student` has any value other than `yes,` only the employment query is executed.

The general form of the N-S chart for selection structures uses a triangle with its base at the top (see Figure 9.1).

The condition to be tested is inserted in the triangular area. A vertical line is drawn from the lower tip of the triangle to form two columns. The left column is headed "NO" and the right column is headed "YES." The statements to be executed when the condition in the triangle is true are listed in the right or "YES" column. The left column is blank because in a single-option conditional structure, a false or "NO" condition implies that no action is taken.

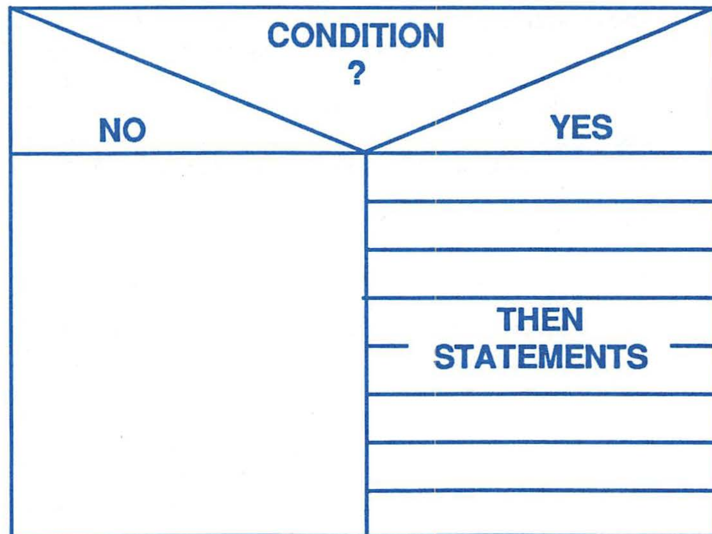


FIGURE 9.1

Problem: Write a program to enter the speed of a car and to output a message if the speed exceeds the limit of 55 miles per hour.

The IPO chart is shown in Figure 9.2. Notice that *Main Program* is circled, and the name *SpeedCheck1* appears at the top of the form.

A single-option structure is used, because a message is printed only if the car is speeding. No message is displayed if the speed is less than or equal to the limit. The N-S chart is displayed in Figure 9.3. The left side of the decision portion is blank because only the yes option requires that an action be taken.

A complete program listing and two sample runs (one with a speed above the limit and another with a speed below) are shown in Figure 9.4.

THE DOUBLE-OPTION STRUCTURE

The double-option structure contains a second option that is executed when the condition evaluates to false.

FORMAT:
```
if <condition> then
   <statement a>
else
   <statement b>;
```

If the condition evaluates to true, <statement a> is executed, <statement b> is ignored and the program continues with the instruction following <statement b>. If the condition evaluates to false, <statement b> is executed, <statement a> is ignored, and the program continues with the instruction following <state-

IPO CHART

(Main Program) Function Procedure (circle one)

Name SpeedCheck 1

Class	Identifier	Description
INPUT	limit	speed limit
	speed	speed of traveling car
PROCESSING		

	Identifiers		Drawing
OUTPUT	speed		

FIGURE 9.2

SpeedCheck 1

Enter speed of car

speed > limit ?

No

Yes

Print message

FIGURE 9.3

```
program SpeedCheck1;
{Prints a message if speed of car}
{exceeds speed limit}
 const
  limit = 55; {speed limit}
 var
  speed   {speed of traveling car}
   : integer;
begin
 write('Enter speed of approaching car:  ');
 readln(speed);
 if (speed > limit) then
  writeln('Slow down, you''re going tooooo fast!')
end.
```

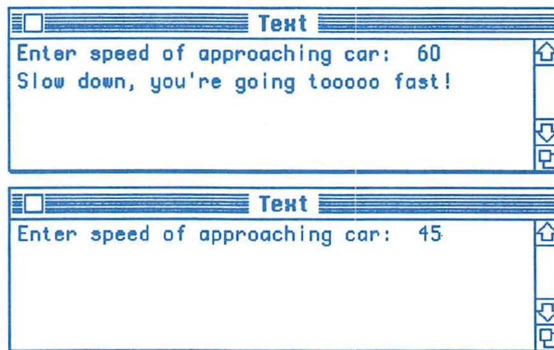


FIGURE 9.4

ment b>. When this structure is evaluated, program execution follows either the **then** branch or the **else** branch, but never both. No semicolon follows the **then** part in a double-option structure.

Example:

```
if name < 'N' then
  write('Room 212')
else
  write('Room 323');
writeln(' for registration.');
```

If `name` begins with letters A through M, `Room 212` is displayed. If it does not, `Room 323` is displayed. In either case, the message `for registration` follows the room number.

As in the single-option structure, you may want to execute more than one statement in either or both of the **then** and **else** parts. Again, all you have to do is enclose this compound statement in a **begin-end** set. A semicolon should not appear before the **else** branch of this structure.

Example:

```
if odd(n) then
  begin
   m := (n + 1)/2;
   b := m + 4
  end
else
  begin
   m := n/2;
   b := m + 3
  end;
writeln(b);
```

Since **odd** returns a Boolean value, it alone can act as the condition in this selection structure. If **n** is odd, the statements following **then** are executed; otherwise, the statements following **else** are executed. In either case, the value of **b** is displayed.

The N-S chart for the double-option structure is shown in Figure 9.5.

In this chart, both the yes and no columns are filled. The statements from the **then** branch are listed in the yes portion of the chart, while the statements from the **else** branch are listed in the no portion.

FIGURE 9.5

Problem: Write a program to enter the speed of a car and output a message if the speed exceeds the limit of 55 miles per hour and another message if the speed is at or below the limit.

The IPO chart is shown in Figure 9.6.

A double-option structure is used, because a message is printed whether the car is speeding or not. The N-S chart is displayed in Figure 9.7. Both the left side and right side of the decision structure are filled, because an action is taken regardless of the result of the decision.

A complete program listing and two sample runs (one with a speed above the limit, and another with a speed below) are shown in Figure 9.8.

IPO CHART

(Main Program) Function Procedure (circle one)

Name Speed Check2

Class	Identifier	Description
INPUT	speed	speed in mph of approaching car
	limit	55
PROCESSING		

	Identifiers		Drawing
OUTPUT	speed		
	message		

FIGURE 9.6

Speed Check2

Enter speed of approaching car	
speed > limit ?	
NO	YES
message	message

FIGURE 9.7

```
program SpeedCheck2;
{Prints a message if speed above the limit}
{and a different message if speed at or below the limit}
 const
  limit = 55; {speed limit}
 var
  speed   {speed of traveling car}
   : integer;
begin
 write('Enter speed of approaching car:  ');
 readln(speed);
 if (speed > limit) then
  writeln('Slow down, you''re going tooooo fast!')
 else
  writeln('No speeding tickets for you today!')
end.
```

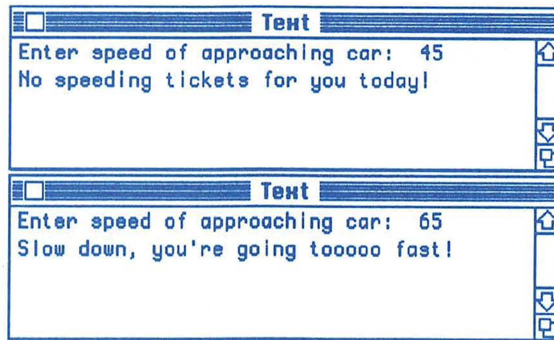


FIGURE 9.8

NESTED SELECTION STRUCTURES

Certain problem solutions require repeated use of **if...then...else** structures where one of these structures is contained entirely within the **then** or **else** portion of another. The resulting structure is referred to as a *nested structure,* where one **if...then...else** "nests" or lies entirely within another.

Problem: Input a student's exam score, and output a grade. If the score is equal to or greater than 90, the grade is *honors.* If the score is less than 90 but greater than or equal to 75, the grade of *passing* is issued. If the score is below 75, the grade is *failing.*

The solution technique used removes the top scores in the first selection, thus making it unnecessary to test the upper-range limit in a second structure. For example, if the exam score is 93, the grade is *honors,* and no further testing of that exam score occurs. If the exam score is 86, there is no need to test for

an upper limit in the next test, because control would never reach this test unless the score is less than 90.

There is no need to test for the third option, because it can be assumed that if the score is not greater than or equal to 90 and not greater than or equal to 75, it must be less than 75 and therefore would merit the grade of failing. This assumption makes the inner **else** branch the *default case,* that is, the branch the program takes if the other options are not selected. This is an example of a double-option structure nested within another double-option structure.

The occurrence of erroneous data entry has been overlooked in the previous discussion. Test scores of −88 and 105 produce grades when no such scores are possible. This kind of problem may be averted by designing the structure to offer four options: re-entry of data, honors, passing, and failing. The **then** branch in the first **if...then...else** structure filters out all erroneous data and prints the message `Data not in valid range.` The placement of the test for erroneous data in the first option eliminates the need for continued testing if the entered data are not valid.

Figure 9.9 shows the IPO chart for the identifiers in the solution.

The decision structures are enclosed within one another. There are four options, and their implementation requires three double-option structures, because each structure compares only two values at a time.

IPO CHART

(Main Program) Function Procedure (circle one)

Name Grading

Class	Identifier	Description
INPUT	testscore	student exam score
PROCESSING	grade	message to be printed

OUTPUT	Identifiers		Drawing
	testscore		
	grade		

FIGURE 9.9

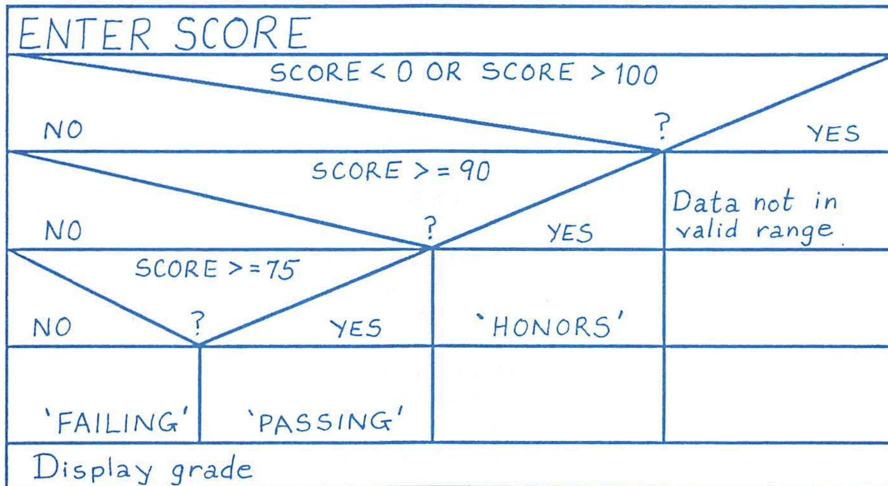


FIGURE 9.10

The first option tests for data that are either above 100 or below 0. The logical operator **or** makes it easier to code the condition used for this decision. The N-S chart (see Figure 9.10) shows the logical structure of the program.

Figure 9.11 shows a program listing and sample runs with data in different ranges to completely test the validity of the nested structure.

9.5 THE CASE STRUCTURE

Although a problem that requires a selection from several options may be solved by using a nested conditional structure, this technique can prove cumbersome. MacPascal provides an additional selection structure that can be employed when nested structures become unwieldy. This structure is called **case**, and is the most complicated, and at the same time the most flexible, of the selection structures.

```
FORMAT: case <var expr> of
        <option 1>:
         <statement a>;
        <option 2>:
         <statement b>;
        <option 3>:
         <statement c>;
             . . .
        otherwise
        <default statement>
        end;
```

```
program Grading;
{Determine grade from numeric test score}
{and identify erroneous data entry}
 var
  testscore : integer; {test score to be graded}
  grade : string; {grade}
begin
 write('Enter test score : ');
 readln(testscore);
 if (testscore > 100) or (testscore < 0) then
  grade := 'Data not in valid range.'
 else if testscore >= 90 then
  grade := 'Honors'
 else if testscore >= 75 then
  grade := 'Passing'
 else
  grade := 'Failing';
 writeln('Grade   :   ', grade)
end.
```

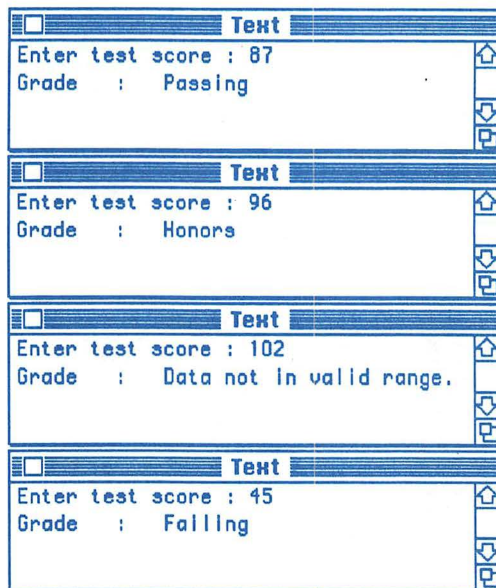


FIGURE 9.11

This structure may include any number of options, and each may include a statement or compound statement. Multiple statements for a particular option must be enclosed within a **begin-end** set. The value of the *selector,* <var expr>, can be an integer or a character. This value determines which option is executed. A match between <var expr> and one of the options determines which of those options is chosen. For example, if <var expr> matches <option 2>, then <statement b> is executed. The last possibility, **otherwise**, is known as the *default case* and is selected if all the other options fail to be chosen. The **otherwise** option is not followed by a colon.

Although the **otherwise** option is not required, it should be included in every **case** structure to avoid the error that occurs when no match is found.

Each option may offer the possibility of a list of values for the selector. Items in the list are separated by commas, and the last item in the list is followed by a colon. If the options overlap, only the first one to satisfy the selector is chosen.

Example:

```
case k of
 -1, 0 :
  writeln('invalid');
 1, 2, 3 :
  writeln('valid');
 otherwise
  writeln('too much')
end;
```

If `k` equals 0 or −1, the message `invalid` is output. If `k` equals 1, 2, or 3, then `valid` is displayed. Any other value causes `too much` to appear.

Once a particular option is selected, the program executes the single statement or compound statement listed for that option and continues at the instruction following the **case**'s **end** statement.

The N-S chart for the **case** structure is shown in Figure 9.12.

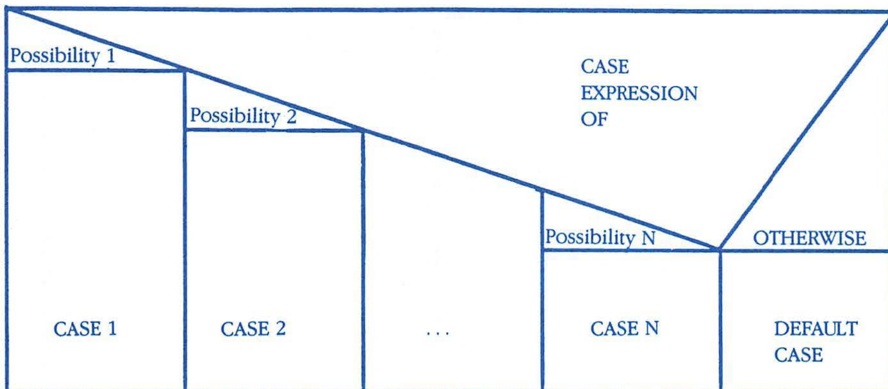


FIGURE 9.12

As indicated in the figure, the **case** structure can be drawn illustrating as many cases as needed to cover the various options in the problem. The default option (**otherwise**) should appear in the right-most column.

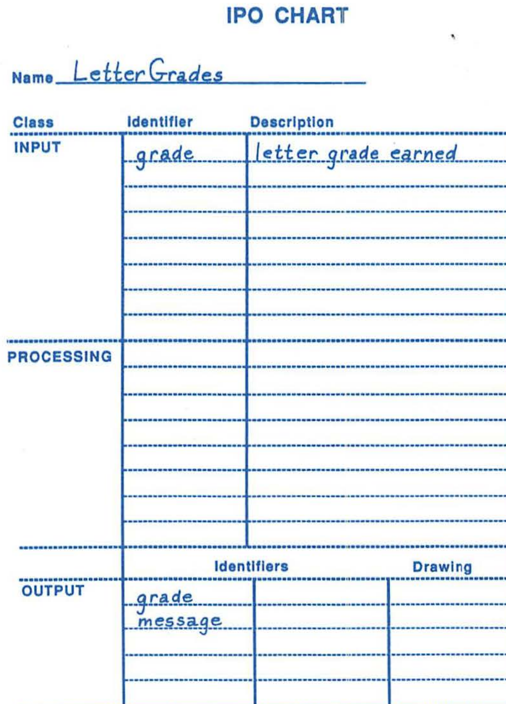
Problem: Write a program that accepts a letter grade *(A, B, C, D,* or *F)* and produces a description of that grade according to the following information:

GRADE	DESCRIPTION
A	Honors
B, C, or *D*	Passing
F	Failing

The IPO chart for the solution is given in Figure 9.13.

A **case** structure is indicated since three distinct options are listed. A fourth option, the default option, should be added to account for any character not in the table (an invalid grade). The N-S chart outlining the solution is given in Figure 9.14.

Figure 9.15 shows a program listing and several sample runs testing the various options, including the **otherwise** option for erroneous data.



IPO CHART

Name LetterGrades

Class	Identifier	Description
INPUT	grade	letter grade earned
PROCESSING		

	Identifiers		Drawing
OUTPUT	grade		
	message		

FIGURE 9.13

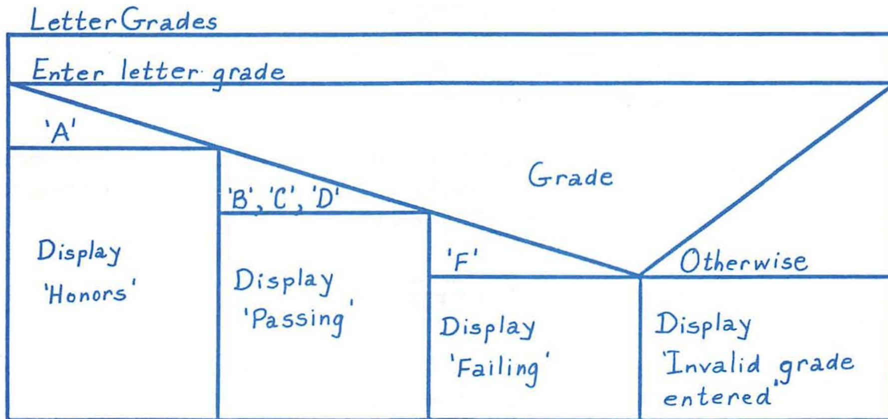


FIGURE 9.14

```
program LetterGrades;
{Accept letter grade and output rating of student's work}
 var
  grade : char; {letter grade earned}
begin
 write('Enter letter grade : ');
 readln(grade);
 write('Rating     :  ');
 case grade of
  'A' :
   writeln('Honors');
  'B', 'C', 'D' :
   writeln('Passing');
  'F' :
   writeln('Failing');
  otherwise
   writeln('Invalid grade entered.')
 end
end.
```

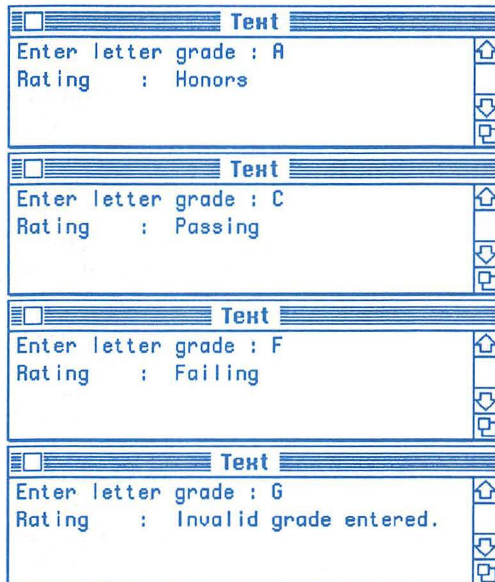


FIGURE 9.15

The message `Passing` is displayed when either *B, C,* or *D* is input for `grade`. The program outputs `Invalid grade entered.` if you attempt to enter grades as lowercase letters. Remember, the computer makes a distinction between upper- and lowercase letters used as data items.

Problem: Write a program to find the total cost of theater tickets given the number of seats required in a desired location. The following table gives the seat location and the unit price for a seat at that location:

LOCATION	UNIT PRICE
Orchestra	$25
Mezzanine	$15
Balcony	$ 8

The solution is divided into three subprograms. Each subprogram is a procedure. The first procedure, `Enter`, accepts the number of seats required and their location in the theater. It employs a list of all the valid choices for the location.

The second procedure, `Cost`, accepts values for `location`, `unitprice`, and `flag` and passes these values to its local variables `place`, `price`, and `error`, respectively. The **case** structure in this procedure determines the price per ticket. The valid values for `place` are 1, 2, or 3. If a value other than those is entered in the first procedure, the identifier `error` is changed to *true* in the second procedure.

The last procedure, `Display`, accepts values for `flag`, `unitprice`, and `amt` and passes them to the local identifiers `flag`, `unitprice`, and `amt`. Although the names chosen for these identifiers are the same, they represent different memory locations. The first three are global, while the last three are local to `Display`. `Display` calculates and outputs the total cost for the theater tickets or, on receiving an invalid location entry, outputs an error message.

Figure 9.16 presents the IPO charts for the problem solution. Each of the procedures has its own IPO chart, as does the main program.

The N-S charts for the complete solution appear in Figure 9.17.

When a program is broken into subprograms, the main program should appear rather simple. Figure 9.18 displays the program listing and several runs.

After `flag` is initialized to *false,* the procedure `Enter` is called, and data for the number of seats and their location are entered.

The `Cost` procedure is then passed a *false* value for the local identifier `error`, and the value of `location` for the local identifier `place`. After the price is determined in `Cost`, values for `flag` and `unitprice` are passed back to the main program and passed to the `Display` procedure. The identifier `flag` should still be *false* unless an improper code was entered for the location.

`Display` outputs the total cost or a message indicating that an incorrect location has been input.

IPO CHART

Main Program | Function | **(Procedure)** (circle one)

Name: Enter

Class	Identifier	Description
INPUT	amt	number of seats required
	location	seat location desired
PROCESSING		

OUTPUT	Identifiers	Drawing
	amt	
	location	

IPO CHART

(Main Program) | Function | Procedure (circle one)

Name: Tickets

Class	Identifier	Description
INPUT		
PROCESSING	flag	identifies illegal data entry
	unitprice	price per ticket
	location	location of theater seats
	amt	number of tickets

OUTPUT	Identifiers	Drawing

IPO CHART

Main Program | Function | **(Procedure)** (circle one)

Name: Cost

Class	Identifier	Description
INPUT	place	theater seat location
PROCESSING	price	price per ticket
	error	identifies illegal data entry

OUTPUT	Identifiers	Drawing
	price	
	error	

IPO CHART

Main Program | Function | **(Procedure)** (circle one)

Name: Display

Class	Identifier	Description
INPUT	flag	identifier of illegal data entry
	unitprice	price per ticket
	amt	number of tickets
PROCESSING	totcost	unitprice x amt

OUTPUT	Identifiers	Drawing
	totcost	
	or	
	errormessage	

FIGURE 9.16

TICKETS

Set invalid data flag to false

Enter number of tickets and location (Enter)

Find price per ticket (Cost)

Determine cost of tickets or display
Invalid data message (Display)

Enter

Enter number of tickets

Enter location for tickets

Cost

Case
Location
of

1	2	3	Otherwise
price = 25.00	price = 15.00	price = 8.00	Set error flag

Display

flag
?

NO	YES
cost = unitcost × number of tickets	Display message that data are invalid
Display cost of tickets	

FIGURE 9.17

```
program Tickets;
{Find the total cost of theater tickets}
 var
  amt,           {number of tickets desired}
  location    {location code for seats desired}
   : integer;
  unitprice   {unit price for tickets}
   : real;
  flag           {indicates invalid location entry}
   : boolean;

 procedure Enter;
  {Enter location and number of tickets}
 begin
  write('Enter the number of seats required:  ');
  readln(amt);
  writeln('    1   orchestra');
  writeln('    2   mezzanine');
  writeln('    3   balcony');
  write('Enter the number preceding the seat location desired:  ');
  readln(location)
 end;

 procedure Cost (place : integer;
          var price : real;
          var error : boolean);
  {Determine cost of tickets chosen}
  {place : location code for seats desired}
  {price : unit price of ticket}
  {error :  indicates invalid location entry}
 begin
  case place of
   1 :
    price := 25.00;
   2 :
    price := 15.00;
   3 :
    price := 8.00;
   otherwise
    error := true
  end
 end;

 procedure Display (flag : boolean;
          unitprice, amt : real);
  {Output results}
  {flag : indicates invalid location entry}
  {unitprice : unit price of tickets}
  {amt : number of tickets desired}
  var
   totcost    {total cost of all tickets}
    : real;
```

FIGURE 9.18 *(continued)*

```
    begin
     if flag then
      writeln('Invalid location entry')
     else
      begin
       totcost := unitprice * amt;
       writeln('Total cost is $', totcost : 5 : 2)
      end
    end;

  begin { main program }
    flag := false;
    Enter;
    Cost(location, unitprice, flag);
    Display(flag, unitprice, amt)
  end.
```

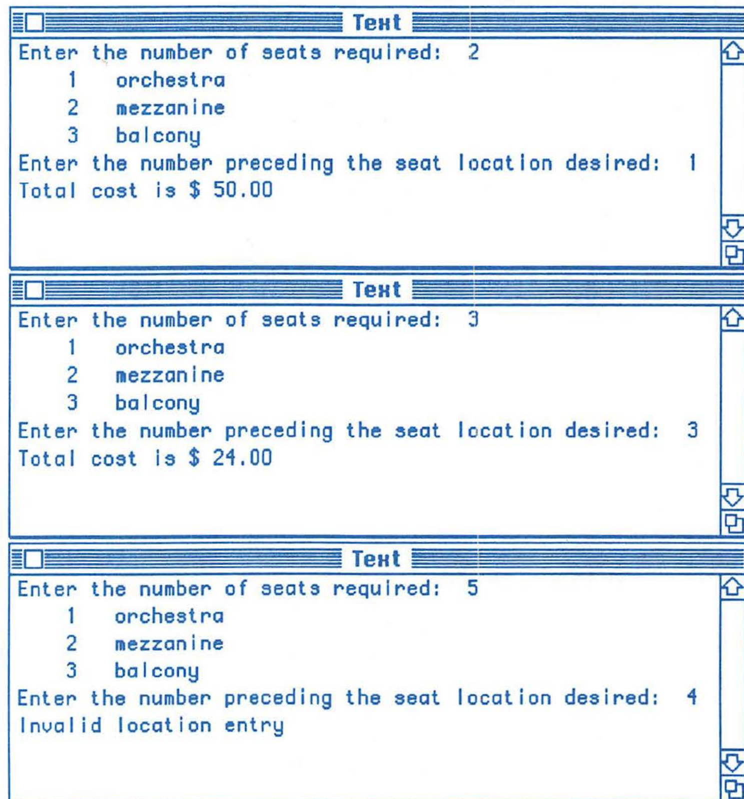


FIGURE 9.18

9.6 TYPICAL PROGRAMMING ERRORS

The following programming errors are typical:

1. Mismatch of operand types in a relational expression.
   Example:
```
const
 x = 34;
 y = 'test';
begin
 if x = y then ...
```
2. Improper operand in logical expression.
   Example:
```
var
 x, y : integer;
begin
 ...
 if x and y > 0 then ...
```
3. Improper use of branching.
   Example:
```
if x > 5 then
 writeln('OK');
writeln('Not OK');
```
4. Omission of the default option in a **case** structure:
   Example:
```
x := -4;
case x of
 1 :
   writeln('one');
 2 :
   writeln('two');
 3 :
   writeln('three')
end;
```

NONPROGRAMMING EXERCISES

1. Classify each of the following Pascal statements as valid or invalid. If the statement is invalid, explain why.
   a.
```
if (a >= 27.5) and (b < 34) then
  x = 4;
```
   b.
```
if 'beta' < 'gamma' then
  writeln('beta');
else
  writeln('gamma');
```

c.
```
if a > b then
  if a > c then
   writeln(a)
  else
   writeln(c)
else
  writeln(b);
```
d.
```
case sqr(test) of
  2 : writeln('2');
  1 : writeln('1')
end;
```
e.
```
case x of
  'a','b' :
   begin
    writeln(a);
    writeln(b)
   end;
  otherwise
   writeln('neither')
end.
```
f.
```
case y of
  >= 90 :
   writeln('good');
  >= 65:
   writeln('fair');
  otherwise
   writeln('poor')
end;
```
g.
```
readln(testscore);
if (testscore < = 100) and (testscore > = 90) then index :=1;
if (testscore < 90) and (testscore > = 65) then index :=2;
if (testscore < 65) and (testscore > = 0) then index := 3;
case index of
 1:
  writeln('Honors');
 2:
  writeln('Pass');
 3:
  writeln('Fail');
 otherwise
  writeln ('Incorrect test score entered')
end;
```

2. Walk through each of the following program sections to determine the output.

a.
```
a := 3;
x := a * 2;
if x <> 6 then
 writeln(x)
else
 a := a + x;
writeln(a);
```

b.
```
a :=3;
x := a * 2;
if x <> 6 then
 writeln(x)
else
 begin
  a := a + x;
  writeln(a)
 end;
```
c.
```
x :=4;
case x + 2 of
 3 :
  writeln('first option');
 4 :
  writeln('second option');
 5, 6, 2 :
  writeln('third option');
 otherwise
  writeln('default option')
end;
```
d.
```
x := 5;
case x + 2 of
 3 :
  writeln('first option');
 4 :
  writeln('second option');
 5, 6, 2 :
  writeln('third option');
 otherwise
  writeln('default option')
end;
```
e.
```
code := 'e';
case code of
 'a', 'b', 'c' :
  writeln('first class');
 'd', 'e' :
  writeln('second class');
 'f' :
  writeln('third class');
 otherwise
  writeln('no class')
end;
```
f.
```
code := 'F';
case code of
 'a', 'b', 'c' :
  writeln('first class');
 'd', 'e' :
  writeln('second class');
 'f' :
  writeln('third class');
 otherwise
  writeln('no class')
end;
```

a. `not a or b`
b. `b and (not c)`
c. `a and b or c`
d. `a and (b or c)`
e. `(a and b) or c`
f. `(not a) or (c and (not b))`
g. `not(not c or (a and c)) or (b and (not b))`

a. Output *hello* when x is 4.
b. If t is positive, find its square root; otherwise, find its square.
c. If t is positive, find its square root; if it is negative, find its square; otherwise output *neither.*
d. If x is odd, double it and output *odd;* otherwise divide it by 2 and output *even.*
e. Test the value of a character identifier c. If c has a value x output *no good;* if c has a value y, output *may be good;* if c has a value z, output *always good;* otherwise output *bad.*

PROGRAMMING EXERCISES

(G) 5. Write a program that displays a question that can be answered by *yes* or *no,* and ask for a response. Output a message indicating whether the response is correct or not.

(G) 6. Write a program to accept as input the amount and type of coins in your pocket, calculate the dollar value, and print the amount and the message "HEAVY METAL" if the amount is at least $2.50, or the message "TRAVELING LIGHT" if the amount is less.

(G) 7. Write a program to input the number of times a baseball player in the Bush League has been at bat and the number of hits the player has made. Calculate the batting average by dividing the number of hits by the number of times at bat and multiplying by 1,000, and express the answer in integer form. If the batting average is 300 or more, print the message "Heavy hitter".

(B) 8. Miser Charge has a $1,000 limit on the amount of credit a purchaser may charge in one store. Write a program to accept as input the amount of a proposed charge, and output a message indicating whether or not the charge exceeds the credit limit.

(G) 9. The coefficients of linear equation $AX + BY + C = 0$ are given. Write a program to calculate the slope of the associated line, and output a message indicating whether the slope is positive or negative.

(B) 10. Ding Dong Bell Telephone Company charges 20 cents for the first five minutes of a local telephone call. For each additional minute, the charge is 7 cents. Write a program to input the name of a caller and the length of the call in minutes. Calculate the charge, and output the caller's name and the charge.

(G) 11. A quadratic equation has the following form: $AX^2 + BX + C = 0$. The "discriminant" is the expression $B^2 - 4AC$. Write a program that accepts as input the three coefficients A, B, and C, A not equal to 0, and determines whether the discriminant is positive, negative, or zero.

(B) 12. The Impress Mail Company accepts orders from its customers for monogrammed innersoles. Each pair of innersoles sells for $3.40. Assume the company starts out with an inventory of 200 pairs of innersoles. Write a program that accepts a number of pairs ordered. If the number of pairs ordered is less than or equal to 200, subtract that number from 200, find and output the cost to the customer, and print the updated inventory. If the number ordered exceeds the number on hand, print a message to the customer indicating that an insufficient supply is presently available.

(G) 13. The Narrow Lane bowling team admits new members if the average score bowled in three try-out games is at least 180. If the average is better than 200, the new member is admitted as a "Star Bowler." Write a program to accept the name of a candidate for the team and his or her scores for the three try-out games, and output a suitable message indicating whether the candidate is accepted and, if so, whether the candidate receives Star Bowler recognition.

(B) 14. Reunited Package Service charges for delivery of packages by weight. The basic charge is $15 for the first 10 pounds. There is an extra charge of $1 for each additional pound. Write a program to accept the weight of a package, and calculate and output the total cost to ship this package using this service.

(G) 15. A professor assigns a letter grade of A for a numeric grade of 90 or higher; B for a grade ranging from 80 through 89; C for 70 through 79; D for 60 through 69; and F for a grade below 60. Write a program to accept, as input, the name of a student and his or her numeric grade, and output the name of the student and the letter grade. Include an appropriate message for each option.

(B) 16. Customers of the Orange Computer Outlet are given a discount of 20 percent on all purchases that exceed $3,000, 15 percent on purchases between $2,000 and $3,000 inclusive, and 10 percent on purchases between $1,500 and $2,000. No other purchases qualify for discount. Write a program to accept a person's name, account number, and total amount of purchases. Output the name, account number, and total net cost after discounting.

(G) 17. Given the lengths of the three sides of a triangle, write a program to determine whether the triangle is equilateral, isosceles, or scalene.

(B) 18. The Mizer Muffler Co. sells mufflers for cars. The mufflers come in six sizes, coded A through F by the company. The basic advertised price of the cheapest muffler is $39.95 installed. As the size increases, the price changes according to the following table:

CODE	*PRICE FACTOR*
A	1.00
B	1.04
C	1.11
D	1.20
E	1.36
F	1.50

The price of a muffler is determined by multiplying the price factor by the price of the cheapest muffler ($39.95). Write a program to accept as input a customer name and the code for the muffler size needed, and output the total cost for the job. Include a message stating that the muffler has a lifetime guarantee.

(G) 19. At Underlin College, students must register according to a schedule based on the first letters of their last names. Students whose last names begin with the letters *A* through *F* register on Monday, *G* through *L* on Tuesday, *M* through *R* on Wednesday, and *S* through *Z* on Thursday. Write a program to print out a registration form including a student's name and ID number and the day on which the student must register.

(B) 20. The Golden Bank charges business customers a minimum of $7.50 per month to maintain a checking account. There is no charge per check if the total number of checks drawn in one month is less than 10. If the number is 10 or more, there is an additional charge of 15 cents per check. If the start-of-the-month balance in the account is at least $1,000, Golden pays interest on the amount at 0.5 percent for the month. However, if the interest earned exceeds the total monthly charges, the bank does not credit the customer with excess interest earned. Write a program to accept a customer account name and number, the balance at the start of the month, and the number of checks drawn during the month; then output the monthly charge (if any).

(B) 21. The Citrus Heaven Grapefruit Company ships cases of grapefruits all over the country. Each case sells for $29.95. The country is divided into four different shipping zones. Shipping charges are determined by the following table:

ZONE	*SHIPPING FACTOR* (%)
1	5.0
2	8.5
3	14.3
4	20.0

The shipping factor is multiplied by the purchase price to determine the shipping costs. Write a program that accepts the name of a customer, the number of cases purchased, and the shipping zone, and that outputs the total cost to the customer.

(B) 22. Consider a problem in which the percentage commission earned by a salesperson is determined by a commission code, as shown in the following table:

SALESPERSON CODE	*COMMISSION FACTOR* (%)
R	18
T	10
S	22
Y	10

The commission is the product of the total sales and the commission factor. Salespeople whose code is X receive half of their total sales as their commission. Write a program that accepts a salesperson's name, total sales, and commission code and outputs the person's name and commission.

(G) 23. In symbolic logic, an implication has the form A ⇒ B. It is interpreted to mean "If A, then B." The implication has the same set of truth values as the statement (not A) or B. The truth value chart is as follows:

A	*B*	*A ⇒ B*	*(NOT A) OR B*
True	True	True	False or True = True
True	False	False	False or False = False
False	True	True	True or True = True
False	False	True	True or False = True

An "argument" is a compound implication. For example, given the statements If A, then B, and A; the conclusion B should follow. This can be stated in argument form as

$$((A \Rightarrow B) \text{ and } A) \Rightarrow B$$

or equivalently,

not(((not A) or B) and A) or B

The argument is called *valid* if all its final truth values are true. Write a program to test the validity of this argument, which is referred to as the *modus ponens argument.*

(B) 24. Given an employee's name, gross monthly salary, and number of dependents, write a program to calculate that employee's net monthly salary, using tax rates from the following table.

NUMBER OF DEPENDENTS	*GROSS MONTHLY SALARY (in dollars)* $0–1600	*$600–1000*	*$1000–2000*	*Over $2000*
0	0.24	0.28	0.31	0.35
1	0.20	0.25	0.28	0.31
>1	0.18	0.22	0.25	0.28

CHAPTER 10

Loops

10.1 INTRODUCTION

Solving problems often requires us to repeat an action. Consider the situation in which a secretary is preparing portfolios for an upcoming conference with 200 participants. Each portfolio contains documentation for five different events. The secretary has to organize the documentation for each participant and place this documentation in a folder—and repeat the process 200 times.

As a second example, consider a stockbroker who has been given a standing order from a client to buy 20 shares of a particular stock every day until the stock increases to a certain value. At that value, all the shares are to be sold. Although the process is repetitive, the stockbroker does not know in advance the exact number of days on which to purchase stock, because there is no way to predict how long it would take to reach the ceiling value, if ever.

These examples typify repetitive actions. The type discussed in the first example ends when a specified number of repetitions is completed, and the type discussed in the second ends when a certain condition is satisfied.

Many programs contain sequences of instructions that must be repeated. These sequences are called *loops*. The computer, incapable of becoming bored or making careless errors, can perform repetitions, saving countless hours of human effort, and produce results that are as accurate as the coding permits.

10.2 LOOPING STRUCTURES

All repetitive structures are similar. They include a condition that controls the number of times the sequence of instructions is to be executed. This section of the book describes the three types of loop structures provided by MacPascal: one that repeats a specified number of times, a second that repeats while a condition is true, and a third that repeats until a condition becomes true.

THE FOR...DO STRUCTURE

The **for...do** structure is designed to handle situations in which you know, in advance, the number of times a sequence of instructions is to be repeated.

FORMAT: **for** `<identifier> := <expr1>` **to** `<expr2>` **do**
`<statement>;`

The **for** statement contains an identifier, called an *index,* with an initial value of <expr1> and a final value <expr2>. <statement> is executed when the index is equal to <expr1>, and once for every succeeding value until that index exceeds <expr2>. The repeated portion is called the *body* of the loop. The body of the loop consists of a single statement or a compound statement enclosed in a **begin-end** set.

The N-S chart that depicts this structure is shown in Figure 10.1.

The entire **for...do** structure is contained in a large box. The body of the loop is displayed as a smaller box contained within the larger one. The index that controls the number of times the loop is executed is defined above the inner box. The inner box is broken up into smaller sections detailing the structure of the body of the loop.

As with all identifiers, the index must be defined in the declaration portion of a Pascal program. An index can be of type **char** or **integer**.

Example:

```
for j := 1 to 3 do
 begin
  write('loop ');
  write('de ')
 end;
writeln('lah');
```

1 is the initial value for the index `j`. The body of the loop consists of two **write** statements that are executed, and the index becomes 2. The **write** statements are executed two more times (for index values of 2 and 3) before the **writeln** statement is executed. The final output appears thus:
`loop de loop de loop de lah`

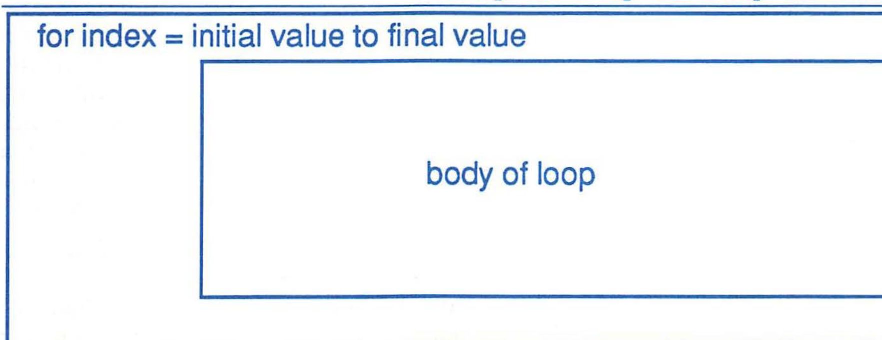


FIGURE 10.1

Replacing **to** to **downto** allows the index to move from a "higher" value to a "lower" one.

```
FORMAT: for <identifier> := <expr1> downto <expr2> do
            <statement>;
```

Example:

```
start := 10;
final := 7;
for i := start downto final do
 writeln('Hello! ');
writeln('That''s some echo! ');
```

Hello! Is displayed four times, once for each value of i. Execution results in the following output:

```
Hello!
Hello!
Hello!
Hello!
That's some echo!
```

MacPascal checks the values in the **for** statement before the loop is executed the first time. If there are inconsistencies, the body of the loop is not executed at all. For example, if the loop begins with the instruction **for n := 6 to 3 do**, the entire loop is ignored, because it is impossible for n to go from 6 to 3 in increasing order.

Although the identifier that serves as the index of a loop can be used in the body of the loop, the value of that identifier cannot be changed within the body or an error message results.

Once the loop is completed, the value of the identifier that is used as the index in the loop has an unpredictable value.

Example:

```
for i := 1 to 3 do
 writeln('This is pass ',i:2);
write('The value of i after ');
writeln('loop completion: ',i:2);
```

The index i is used in the body of the loop as a counter of the number of times the loop is executed. Its value is not altered by the statement in the loop. The value of i after the loop is completed cannot be determined, as can be shown by an execution of the statements in the example.

Problem: Write a program that draws a bull's-eye in the middle of the **Drawing** window. The bull's-eye should have a radius of 100 pixels and contain 10 rings, alternating black and white.

The IPO chart for the solution is shown in Figure 10.2.

The radius of the outermost circle is entered and the outermost circle is drawn before the loop starts. Since the loop is to be executed nine times (making a total of 10 rings), a **for...do** looping structure is used. For every pass through the loop, the circle is inverted and the radius decreased. The outline of the solution is given by the N-S chart of Figure 10.3.

IPO CHART

(Main Program) Function Procedure (circle one)

Name BullsEye

Class	Identifier	Description
INPUT		
PROCESSING	center	x and y coordinates of the center of the bull's-eye
	i	index for loop
	radius	radius of ring

OUTPUT	Identifiers		Drawing
			bull's-eye

FIGURE 10.2

BullsEye

initialize radius
draw solid outer ring
for i = 1 to 9
decrease radius by 10   draw inverted circle

FIGURE 10.3

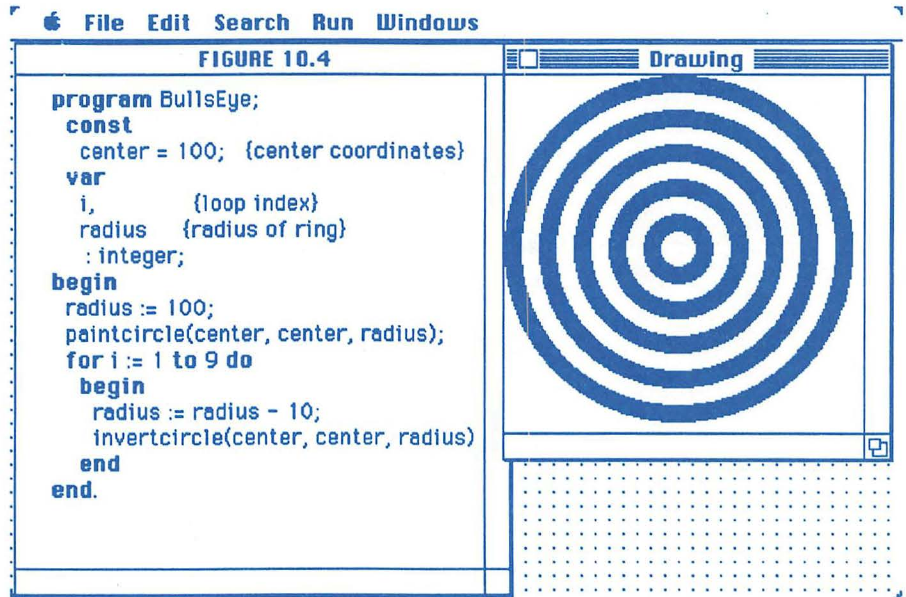


FIGURE 10.4

A complete listing and run of the program are illustrated in Figure 10.4.

The constant **center** is used for both the *x* and *y* coordinates of the center. A more general case would include one constant declared for the *x* coordinate of the center and another for the *y* coordinate.

When the number of repetitions of a sequence of instructions is not known in advance of its execution, two other loop structures are used. Both are controlled by conditions that determine whether or not the loop is repeated. If the condition is evaluated before the body of the loop is executed, a **while...do** structure is used. In this case, the body of the loop may never be executed. If the condition is evaluated after the body of the loop has been executed, a **repeat...until** structure is used, and the body of the loop is executed at least once before testing for termination.

THE WHILE...DO STRUCTURE

FORMAT: **while** `<condition>` **do**
`<statement>`;

As with selection structures, <condition> is a combination of relational operators, Boolean operators, arithmetic operators constants, and identifiers, and it yields a Boolean value. If that value is true, the single statement or compound statement <statement> is executed. If the condition evaluates to false,

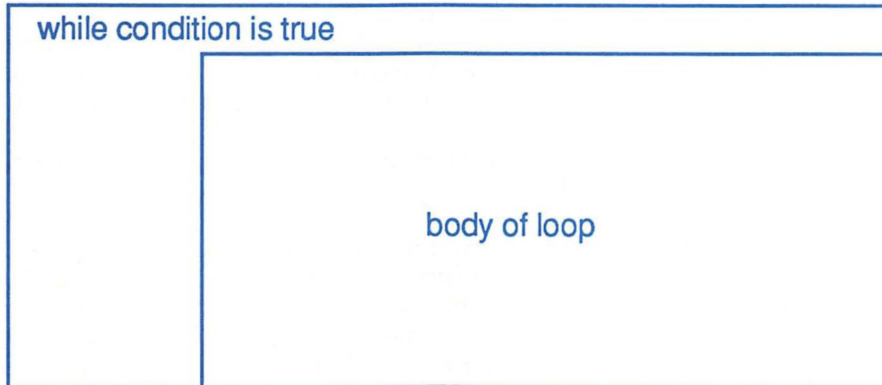


FIGURE 10.5

<statement> is ignored and execution of the program continues with the statement following the entire structure. A semicolon terminates the structure.

The N-S chart for the **while..do** structure is shown in Figure 10.5. Since the test for termination occurs at the beginning of the loop, the line *while condition is true* precedes the box that represents the body of the loop.

while...do structures are often used when repetition is terminated by a predetermined value that you enter while the loop is executing. The predetermined value is called an *endcode* or *sentinel.*

Example:

```
writeln('Enter xxx to end');
write('Enter your name:  ');
readln(name);
while name <> 'xxx' do
 begin
  writeln('Welcome ');
  writeln;
  write('Enter your name:  ');
  readln(name);
 end;
writeln('End of welcome.');
```

The first **writeln** tells you what value to use to end the execution of the loop. The **write** and **readln** statements that follow allow you to input data before entering the loop. The **while** statement tests the value of **name**. As long as **name** is not *xxx,* the welcome message is displayed, the next input requested, and control transferred to the **while** statement, where this input is now tested against the endcode to see whether to end the loop. `End of welcome.` is displayed after *xxx* is entered for **name**.

When using a **while...do** structure for applications of this type, you should always precede the loop with a description of how the loop is to be terminated (what the sentinel value is), and the statements that ask for the entry of the data used in the condition of the **while...do** (`write('Enter your name: ');` `readln(name);`). These statements should be repeated as the last part of the body of the loop so that the newly entered data can be used in the condition of the **while...do** loop.

Problem: Riverview Apartments is planning rent increases for the next year. The percent increase is to be determined by the number of bedrooms in each apartment. Write a program that accepts the apartment number, the number of bedrooms that apartment has, and the current rent. The program should find and display the new rent for the next year, using the following information:

NUMBER OF BEDROOMS	PERCENT INCREASE
1	4%
2	6
3	7

The program should continue to accept data until an endcode of *zzz* is entered for the apartment number.

The IPO chart describing the identifiers used in the solution is given in Figure 10.6.

The solution consists of a **while...do** structure that continues to accept and process data as long as the apartment number is not *zzz*. Inside that loop structure is a **case** structure that provides the percent increase for the different sizes of the apartments. The default option in this structure, **otherwise**, indicates that an improper number of bedrooms has been entered. The N-S chart blueprinting the solution is shown in Figure 10.7.

Figure 10.8 provides a complete program listing and sample runs.

The calculation of the new rent, `nrent`, is determined by multiplying the present rent, `prent`, by 1 plus the percent increase written as a decimal. For example, an increase of 4 percent means the present rent is multiplied by 1.04 to get the new rent. The message `End of processing` indicates that the loop has been terminated.

THE **REPEAT...UNTIL** STRUCTURE

The **repeat...until** structure is used when the loop must be executed at least once. The test for termination appears at the end of the looping structure.

```
FORMAT: repeat
          <statement>
        until <condition>;
```

IPO CHART

(Main Program) Function Procedure (circle one)

Name: Riverview

Class	Identifier	Description
INPUT	apt	apartment number
	nofb	number of bedrooms
	prent	present rent
PROCESSING	endcode	termination code for input
	nrent	new rent

OUTPUT	Identifiers		Drawing
	apt		
	nofb		
	prent		
	nrent		

FIGURE 10.6

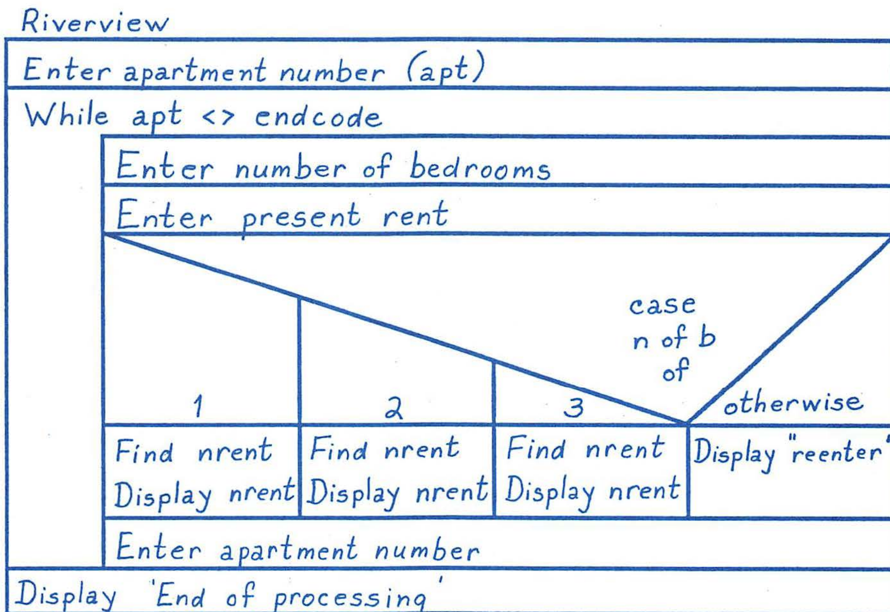


FIGURE 10.7

```
program Riverview;
{Given apartment number, determine new rent based}
{on the number of bedrooms and present rent}
 const
  endcode = 'ZZZ';
 var
  apt          {apartment number}
   : string;
  nofb        {number of bedrooms}
   : integer;
  prent,           {present rent}
  nrent        {new rent}
   : real;
begin
 writeln('Enter ZZZ for apartment number to end processing');
 write('Enter apartment number : ');
 readln(apt);
 while apt <> endcode do
  begin
   write('Enter number of bedrooms : ');
   readln(nofb);
   write('Enter present rent : $');
   readln(prent);
   case nofb of
    1 :
     begin
      nrent := 1.04 * prent;
      writeln('New rent : $', nrent : 7 : 2)
     end;
    2 :
     begin
      nrent := 1.06 * prent;
      writeln('New rent : $', nrent : 7 : 2)
     end;
    3 :
     begin
      nrent := 1.07 * prent;
      writeln('New rent : $', nrent : 7 : 2)
     end;
    otherwise
     begin
      writeln('Number of bedrooms must be 1 or 2 or 3.');
      writeln('Reenter data')
     end
   end; {end of case}
```

FIGURE 10.8 *(continued)*

```
        writeln;
        write('Enter apartment number : ');
        readln(apt)
      end; {of while..do loop}
    writeln;
    writeln('End of processing.')
  end.
```

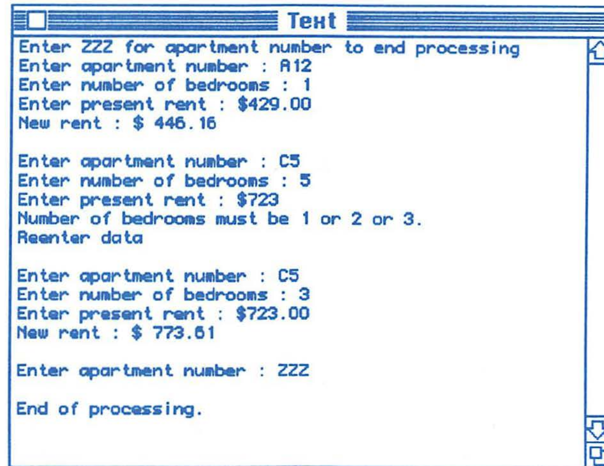


FIGURE 10.8

The keyword **repeat** signals the beginning of the structure, and the keyword **until**, the end. Even if <statement> is compound, *no* **begin-end** set is needed. Just list the statements and separate them by semicolons. The statement preceding **until** does not have to be followed by a semicolon because **until** signifies the end of the structure. <condition> has the same format as in selection and **while...do** structures and is followed by a semicolon to end the **repeat** structure. If <condition> is false, the body of the loop is repeated. If <condition> is true, the loop ends and execution continues with the statement following **until**.

The N-S chart for the **repeat...until** structure is shown in Figure 10.9. Since the test for termination occurs at the end of the loop, the *repeat until, no...condition is true* follows the box representing the body of the loop.

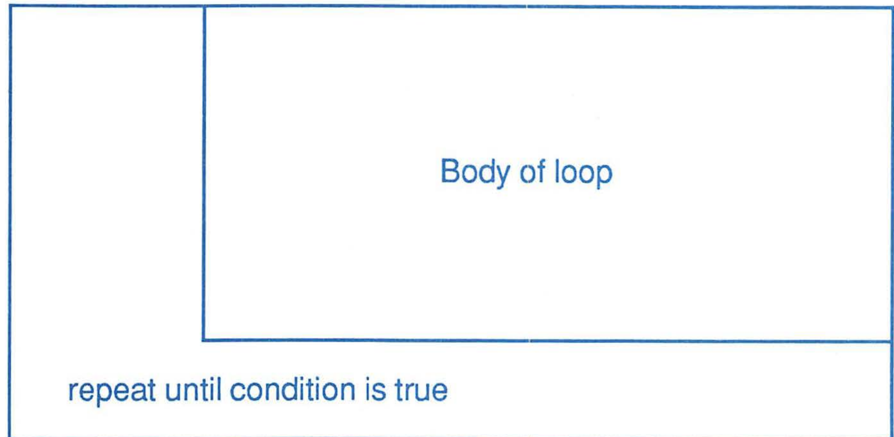


FIGURE 10.9

Example:

```
repeat
  writeln('You look great! ');
  write('Enough (y/n)?   ');
  readln(ans)
until ans = 'y';
writeln;
writeln('What a person! ');
```

The **repeat** introduces the body of the loop. If you choose `n` for a response to the question, the body of the loop is repeated. If `y` is chosen, the loop ends and `What a person!` is output. Notice that the **readln** statement does not end with a semicolon, because it precedes **until**.

Problem: Jack Sprat has just purchased 100 shares of HamHox International at $50 per share. He has given an order to his stockbroker to sell his stock if the stock goes above $80 per share or falls below $20 per share. Write a program that accepts the daily change in the price of the stock and calculates the new price per share. When the stock is sold, have the program output the net profit or loss and a message indicating "profit" or "loss."

Figure 10.10 shows the IPO chart for the solution.

The **repeat...until** structure is an appropriate looping structure for the solution to this problem since the initial price is $50 and the body of the loop must be executed at least once before determining whether to sell or not. The body contains statements that ask for the change and find the new price. A selection structure follows the **repeat** structure and determines whether Jack made a profit or suffered a loss. The N-S chart for the solution is depicted in Figure 10.11.

A complete listing and sample run of the program are shown in Figure 10.12.

The condition following **until** contains a compound logical expression. The logical operator **or** separates two simpler conditions and causes termination of

IPO CHART

Main Program (circled) | Function | Procedure (circle one)

Name *Ups And Downs*

Class	Identifier	Description
INPUT	change	daily stock price change
	ulimit	upper limit for stock selling
	llimit	lower limit for stock selling
	price	purchase price per share
	noofshares	number of shares purchased
PROCESSING	newprice	new daily price per share
	diff	profit or loss
	cost	initial cost of stock

Class	Identifiers		Drawing
OUTPUT	ulimit	newprice	
	llimit	cost	
	price	diff	
	noofshares		
	change		

FIGURE 10.10

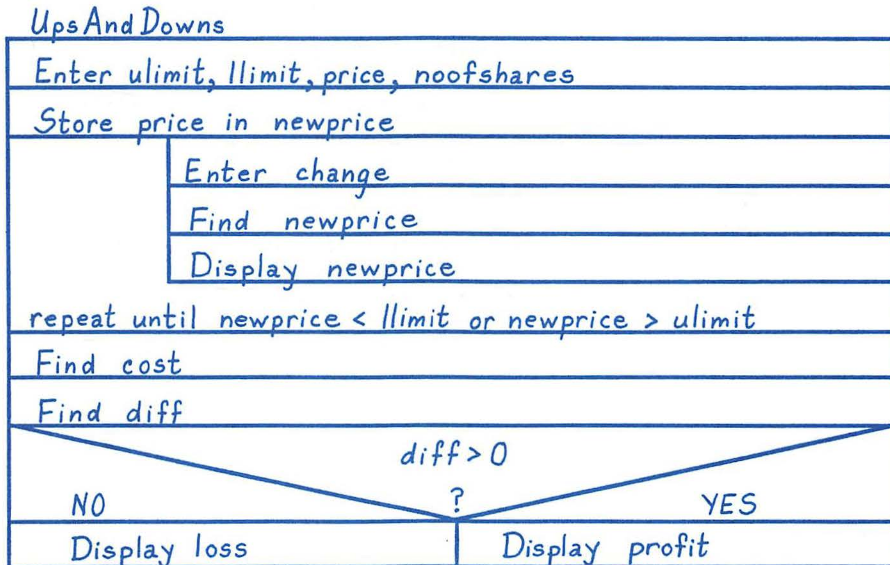


FIGURE 10.11

```
program UpsAndDowns;
{Determines when to sell a stock based on its current price}
 var
  ulimit,                 {upper limit for stock sale}
  llimit,                 {lower limit for stock sale}
  price,                   {purchase price per share}
  noofshares,           {number of shares purchased}
  change,                 {daily stock price change}
  newprice,             {new daily price per share}
  diff,                      {profit or loss}
  cost,                     {initial cost of stock}
   : real;
begin
 writeln('Daily activities of HamHox International Stock');
 write('Enter upper limit for stock sale:  $');
 readln(ulimit);
 write('Enter lower limit for stock sale:  $');
 readln(llimit);
 write('Enter purchase price per share:  $');
 readln(price);
 write('Enter number of shares purchased:  ');
 readln(noofshares);
 newprice := price;
 writeln;
 repeat
  write('Enter today''s change for HamHox International Stock:  ');
  readln(change);
  newprice := newprice + change;
  writeln('New price per share:  $', newprice : 11 : 2)
 until (newprice < llimit) or (newprice > ulimit);
 cost := price * noofshares;
 diff := newprice * noofshares - cost;
 write('Cost of stock : $', cost : 10 : 2);
 if diff > 0 then
  writeln('   Profit earned:  $', diff : 10 : 2)
 else
  writeln('   Loss incurred:  $', diff : 10 : 2)
end.
```

FIGURE 10.12 *(continued)*

```
Text
Daily activities of HamHox International Stock
Enter upper limit for stock sale:  $80.00
Enter lower limit for stock sale:  $20.00
Enter purchase price per share:  $50.00
Enter number of shares purchased:  100

Enter today's change for HamHox International Stock:  +2.50
New price per share:  $       52.50
Enter today's change for HamHox International Stock:  -7.75
New price per share:  $       44.75
Enter today's change for HamHox International Stock:  +40
New price per share:  $       84.75
Cost of stock : $   5000.00   Profit earned:  $    3475.00
```

FIGURE 10.12

the loop if either of these simpler conditions becomes true; that is, if `newprice` is greater than $80 or less than $20.

The body of the loop consists of statements that accept `change`, update `newprice`, and store that value in the location represented by the same identifier, `newprice`, for the next pass through the loop.

The entry of constantly fluctuating changes can cause a never-ending or infinite loop if the price of the stock never goes above or falls below the limits set by Jack. If you have to stop execution of a program caught in an infinite loop, use the mouse to choose **Halt** from the **Pause** menu.

10.3 SUMMING AND COUNTING STATEMENTS

There are two assignment statements that are commonly used with loops for counting and summing.

The general form for a *counting statement* is as follows:

<identifier> := <identifier> + <constant>;

The same constant is added to the <identifier> each time the statement is executed, and this value replaces the previous contents of <identifier>. The same identifier appears on both sides of the assignment operator. For example, `count := count + 1` is an instruction that adds 1 to the identifier `count` and assigns the new value to the identifier `count`. In effect, the value of `count` increases by one each time this statement is processed. If `count` is given a

proper initial value, count can be used to represent the number of times the body of a loop was repeated.

You must always assign an initial value to a counting identifier using an assignment statement. MacPascal does not automatically do this for you.

Of course, counting need not be in steps of 1. For example, if max is selected as a counting identifier and you want to count in increments of 2, a counting statement might read max := max + 2.

Another assignment statement frequently used in loop structures is called a *summing statement.* An instruction of this type keeps a running total of the values of an identifier used in that instruction. The general form for a summing statement is as follows:

<identifier 1> := <identifier 1> + <identifier 2>;

The summing statement differs from the counting statement in that the amount added each time the statement is processed is itself an identifier. <identifier 2> is added to the current contents of <identifier 1>, and the new total replaces the former contents of <identifier 1>. <identifier 1> is called a *summing identifier.*

For example, if you wish to find the sum of four weekly salaries, where total is the summing identifier and the identifier salary is used to represent each week's salary, a suitable summing statement might appear thus:

total := total + salary

Each time this statement is processed, a new value for salary is added to the current value of total and that sum becomes the new value of the identifier total. After four executions of this instruction, total represents the sum of the four weekly salaries.

Example:

```
n := 4;
ans := 0;
while n <= 100 do
 begin
  ans := ans + n;
  n := n + 2
 end;
writeln('Sum is ',ans:5);
```

The counting identifier n is initialized to 4 and increased by 2 each time the loop is executed. The summing identifier ans is initialized to 0 and is increased by the value of n at each pass through the loop. The loop terminates when the value of n exceeds 100, at which point the sum of the even numbers from 4 to 100, ans, is displayed.

The technique used in the previous example can be extended to other operations. For example, you might be interested in finding the product of all the integers from a specified integer down to 1. This is known as "finding the factorial" of a number. For example, 5 factorial, represented mathematically as 5!, is the product of the integers from 5 to 1 ($5 \times 4 \times 3 \times 2 \times 1$). Although the operation is multiplication rather than addition, the technique is similar.

Example:

```
prod := 1;
for k := 5 downto 1 do
 prod := prod * k;
writeln('5! is ',prod:3);
```

The identifier `prod` is initialized to 1. The index of the loop, `k`, is multiplied by the current value of `prod` to produce the next value of `prod`. This process continues until `k` reaches 1, at which point the loop is completed and the answer displayed.

Factorials tend to grow rapidly. You may have to use a long integer type to represent a large factorial.

Problem: Write a program to process monthly transactions in a checking account. Enter the opening balance, the check amounts as negative values, and the deposit amounts as positive values, and output the closing balance. Transaction processing terminates when 0 is used for the amount of the transaction or the balance is less than or equal to 0. If the balance falls below 0, display a message indicating that there are insufficient funds and showing the balance of the account discounting the overdrawn check. Otherwise, just print the balance in the account. The program should also count the number of checks drawn from the account and the number of deposits made to the account.

The IPO chart for the solution is shown in Figure 10.13.

Since the number of monthly transactions varies from month to month, a **while...do** structure is used. The counters for the number of checks and deposits are initialized first. As indicated in the previous discussion, the statement used to enter a transaction must occur immediately preceding the loop and must appear in the last part of the body of the loop. The compound logical condition following **while** can cause termination of the loop in two different ways: if the balance falls below 0 or if 0 is entered for a transaction. The selection structure that follows this **while...do** structure determines which of these two conditions actually terminates the loop and performs an appropriate action based on this decision. The selection structure following the updating of the balance determines whether a transaction is a check or a deposit and updates the appropriate counter. The N-S chart for the solution is illustrated in Figure 10.14.

Figure 10.15 shows a complete listing and a sample run of the program.

The **while** loop condition has two parts. The logical operator **and** connects the two conditions, since you want to end the loop if either of the two parts becomes false.

The first part of the condition, `balance + trans >= 0`, terminates execution of the loop if the sum of the most recently entered transaction and the current balance yields a negative balance. This result signifies that there are insufficient funds remaining in the account to cover the amount of the check. The **then** part of the **if...then...else** statement that follows the loop handles this case.

The second part of the condition in the **while** statement, `trans <> 0`, terminates execution of the loop if you enter 0 for a transaction. The value 0 is used as an endcode to signal that no additional data items are to be entered. The **else** part of the **if...then...else** structure displays the closing balance if this condition caused termination of the loop.

The `balance := balance + trans` statement is an example of a summing statement, since it keeps a running total of the account balance. The identifiers `nocks` and `nodeps` are updated after `trans` is added to the current balance and are examples of counting statements.

IPO CHART

(Main Program) Function Procedure (circle one)

Name: Banking

Class	Identifier	Description
INPUT	balance	amount in checking account
	trans	transaction
PROCESSING	nocks	number of checks
	nodeps	number of deposits

OUTPUT	Identifiers		Drawing
	balance		
	trans		
	nocks		
	nodeps		

FIGURE 10.13

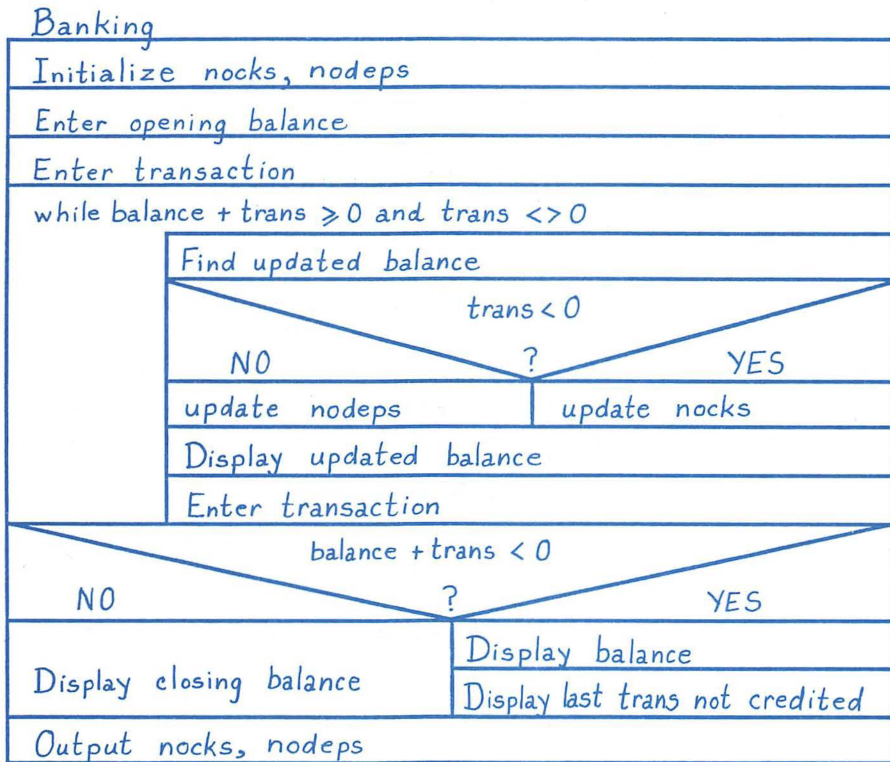
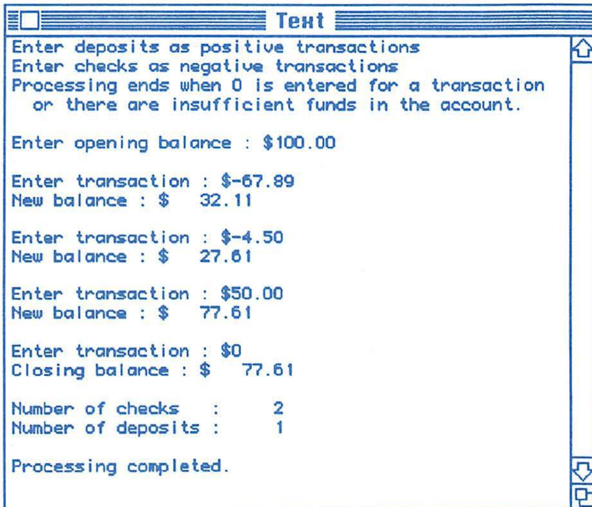


FIGURE 10.14

```
program Banking;
{Monthly checking account transactions}
 var
  balance,           {amount in checking account}
  trans              {transaction}
   : real;
  nocks,               {number of checks}
  nodeps             {number of deposits}
   : integer;
begin
 nocks := 0;
 nodeps := 0;
 writeln('Enter deposits as positive transactions');
 writeln('Enter checks as negative transactions');
 writeln('Processing ends when 0 is entered for a transaction');
 writeln('  or there are insufficient funds in the account.');
 writeln;
 write('Enter opening balance : $');
 readln(balance);
 writeln;
 write('Enter transaction : $');
 readln(trans);
 while (balance + trans >= 0) and (trans <> 0) do
  begin
   balance := balance + trans;
   if trans < 0 then
    nocks := nocks + 1
   else
    nodeps := nodeps + 1;
   writeln('New balance : $', balance : 8 : 2);
   writeln;
   write('Enter transaction : $');
   readln(trans)
  end;
```

FIGURE 10.15 *(continued)*

```
  if balance + trans < 0 then
   begin
    writeln('Insufficient funds available for last transaction');
    writeln('    Balance : $', balance : 8 : 2);
    writeln('    Last transaction (not credited to account) : $', trans : 8 : 2)
   end
  else
   writeln('Closing balance : $', balance : 8 : 2);
  writeln;
  writeln('Number of checks   : ', nocks : 4);
  writeln('Number of deposits : ', nodeps : 4);
  writeln;
  writeln('Processing completed.')
end.
```



```
Enter deposits as positive transactions
Enter checks as negative transactions
Processing ends when 0 is entered for a transaction
  or there are insufficient funds in the account.

Enter opening balance : $100.00

Enter transaction : $-67.89
New balance : $   32.11

Enter transaction : $-4.50
New balance : $   27.61

Enter transaction : $50.00
New balance : $   77.61

Enter transaction : $0
Closing balance : $   77.61

Number of checks   :    2
Number of deposits :    1

Processing completed.
```

FIGURE 10.15

10.4 NESTED LOOPING STRUCTURES

Some programs may contain more than one loop structure. The structures may follow one another in sequence, or one may be contained entirely within another. When the latter case occurs, the loop structures are said to be *nested*.

Examples:

```
for i := 1 to 3 do
 for j := 1 to 2 do
  writeln(i : 4, j : 4);
```

The body of the **for...do** structure with index `i` consists of one statement, another **for...do** structure with index `j`. The inner loop is completely executed for each pass through the outer loop. For example, `j` goes from 1 to 2 when `i` is 1, `j` goes from 1 to 2 when `i` is 2, and so on. The output generated by these statements is:

```
1   1
1   2
2   1
2   2
3   1
3   2
```

```
k := 1;
repeat
 for i := 1 to k do
  writeln('inner');
 writeln;
 k := k + 1
until k = 4;
```

The **for** loop is completed for each pass through the **repeat** loop. The number of repetitions in the **for** loop is controlled by **k**, and the value of **k** is changed each time the **repeat** loop is traversed. The resulting output appears as:

```
inner

inner
inner

inner
inner
inner
```

Problem: Write a program to find all the prime numbers between 3 and a value entered by the user. A prime number is an integer that is evenly divisible (remainder is 0) by only 1 and itself. For example, 37 is a prime since 1 and 37 are the only divisors of 37 that produce a remainder of 0, and 36 is not a prime, because 2, 3, 4, and 6 are additional divisors that produce a remainder of 0.

The IPO chart is shown in Figure 10.16.

After the upper limit for testing is entered, the solution must test all the odd numbers from 3 up to and including that limit. Since all even numbers are divisible by 2, none of them can be prime and you do not have to test them. A **repeat...until** structure is used to cycle through the numbers to be tested.

A general approach in determining if an integer is prime or not uses all the odd integers from 3 up to that integer as divisors. If one of these divisions produces a 0 remainder, the given integer is not a prime and no further testing need occur. Since at least one divisor must be tested, a **repeat...until** structure is used to test all possible divisors of an integer. This second **repeat...until** structure, which is nested in the first, can end if a divisor has been found that gives a remainder of 0 or if you've tested all the integers and found that none give a remainder of 0. The selection structure following the inner **repeat...until** struc-

IPO CHART

(Main Program) Function Procedure (circle one)

Name Prime

Class	Identifier	Description
INPUT	limit	upper limit for testing primes
PROCESSING	index	counter to cycle through primes
	tv	test value tested for prime
	tlimit	limit for testing divisors
	flag	indicates whether divisor has been found or not

Class	Identifiers	Drawing
OUTPUT	limit	
	tv	

FIGURE 10.16

ture indicates that an integer is prime if the reason that loop terminated is that all possible divisors have been tested and none produces a remainder of 0.

The N-S chart outlining the solution is given in Figure 10.17.

A complete program listing and a sample run are illustrated in Figure 10.18.

The Boolean `flag` is initialized to *false* inside the outer **repeat...until** loop. Its value is changed to *true* if a divisor has been found that gives a remainder of 0. The top limit, `tlimit,` is the truncated value of the square root of the upper limit entered from the keyboard. When looking at the possible divisors of an integer, only divisors up to the square root of that number have to be tested.

The identifier `index` is initialized to 3 and increased by 2 during each pass through the inner loop, since only odd integer divisors are tested.

The selection structure following the inner **repeat...until** contains `not flag.` If the value of `flag` is *false* (you've tested all the divisors and none give a remainder of 0), `not flag` is true and causes the **then** part of the **if...then** structure to be executed.

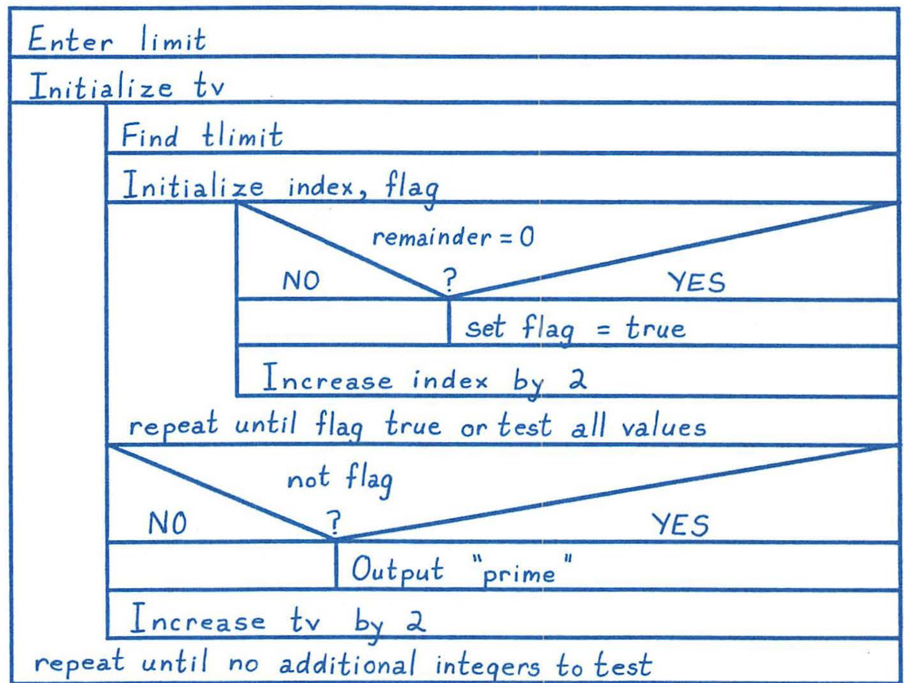


FIGURE 10.17

```
program Prime;
{Determine the prime numbers from 3 to a given limit}
 var
  limit,              {largest integer to be tested for prime}
  index,              {cycles through odd divisors from 3 to limit}
  tv,                   {value of divisor of test integer}
  tlimit            {largest divisor to be tested}
   : integer;
  flag                {signals divisor with no remainder has been found}
   : Boolean;
begin
 write('Enter upper limit for integers to be tested:  ');
 readln(limit);
 tv := 3;
 repeat
  tlimit := trunc(sqrt(tv));
  index := 3;
  flag := false;
  repeat
   if tv mod index = 0 then
    flag := true;
   index := index + 2
  until (flag = true) or (index > tlimit);
  if not flag then
   writeln(tv : 4, ' is a prime number');
  tv := tv + 2
 until (tv > limit)
end.
```

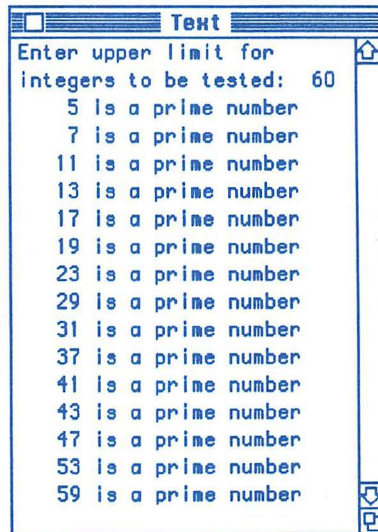


FIGURE 10.18

Problem: Write a program that accepts a student's name and scores on five tests. Find an average numeric grade for each student, and determine and output the rating corresponding to the average grade according to the following information:

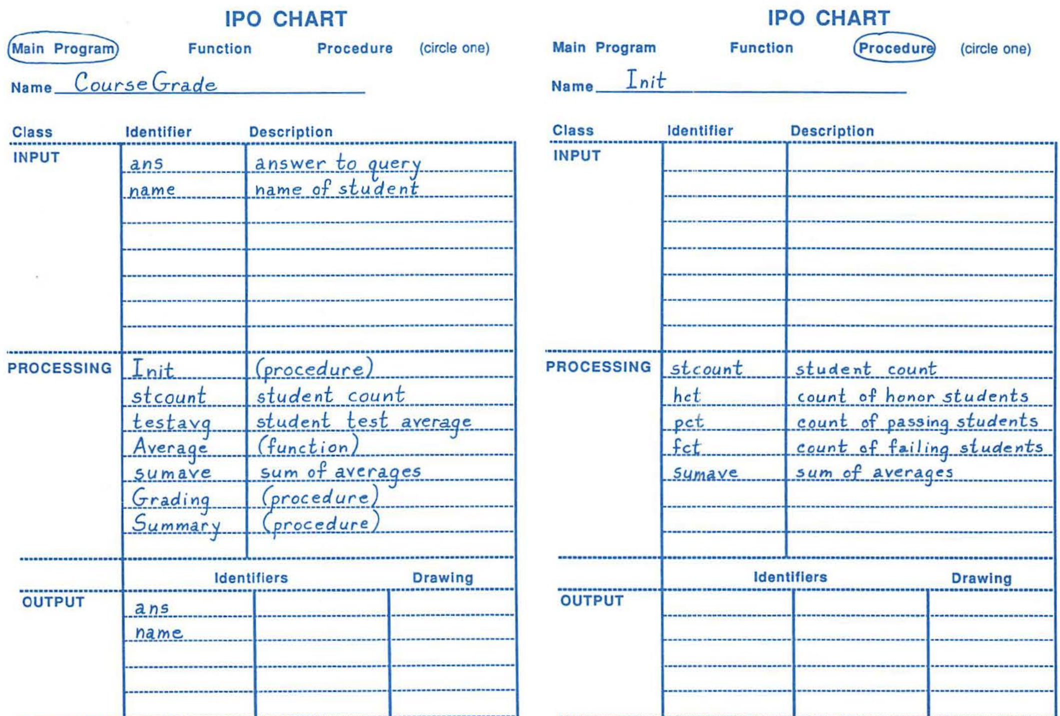
AVERAGE	RATING
85 or above	Honors
70 – 84.9	Pass
below 70	Fail

The program should accommodate a class of unspecified size and should determine and output the number of students receiving each letter grade and the average numeric grade for the entire class. In addition, the program should not allow any grade above 100 or below 0 to be entered as a test score.

To make the solution easier to design and implement, it is divided into four subprograms: `Init` (initializes counters and sums), `Average` (finds the average of the student's test scores), `Grading` (determines rating and updates counts), and `Summary` (outputs the results).

Figure 10.19 provides IPO charts for the identifiers in this program.

The first step is to initialize all counters and sums. This is done by the procedure `Init`. Since the number of students in the class is unknown in



IPO CHART

(Main Program) Function Procedure (circle one)

Name CourseGrade

Class	Identifier	Description
INPUT	ans	answer to query
	name	name of student
PROCESSING	Init	(procedure)
	stcount	student count
	testavg	student test average
	Average	(function)
	sumave	sum of averages
	Grading	(procedure)
	Summary	(procedure)

Class	Identifiers	Drawing
OUTPUT	ans	
	name	

IPO CHART

Main Program Function (Procedure) (circle one)

Name Init

Class	Identifier	Description
INPUT		
PROCESSING	stcount	student count
	hct	count of honor students
	pct	count of passing students
	fct	count of failing students
	sumave	sum of averages

Class	Identifiers	Drawing
OUTPUT		

FIGURE 10.19 *(continued)*

IPO CHART

Main Program ~~ **(Function)** ~~ Procedure (circle one)

Name: Average

Class	Identifier	Description
INPUT	testscore	test score for a student
PROCESSING	index	loop index
	testsum	sum of test scores
	Average	average of test scores

OUTPUT	Identifiers		Drawing
	testscore		

IPO CHART

Main Program ~~ Function ~~ **(Procedure)** (circle one)

Name: Grading

Class	Identifier	Description
INPUT	stavg	test average
PROCESSING	letter	letter grade
	hon	count of honor students
	pass	count of passing students
	fail	count of failing students

OUTPUT	Identifiers		Drawing
	letter		

FIGURE 10.19

IPO CHART

Main Program ~~ Function ~~ **(Procedure)** (circle one)

Name: Summary

Class	Identifier	Description
INPUT	coursesum	sum of averages
	students	student count
	honors	count of honor students
	pass	count of passing students
	fail	count of failing students
PROCESSING	courseavg	average of all students

OUTPUT	Identifiers		Drawing
	courseavg		
	honors		
	pass		
	fail		

advance, a **while...do** structure is appropriate. This loop ends when the response to the question asking if processing is to continue is no.

The body of the **while...do** structure first asks for student name and updates the count of students. Next, control is transferred to the subprogram `Average`, where test scores are entered and a numeric average is calculated.

The subprogram `Average` contains a **for...do** structure that reads in the five student test scores. Nested in this structure is a **repeat...until** structure that rejects any test score that does not fall within the allowable range. This common programming practice should be employed when the possibility of entering erroneous data exists.

After returning to the main program and updating the sum of the student averages, control is transferred to the `Grading` subprogram, which determines and outputs the rating that corresponds to the course average. It also updates the count for each rating. Control is returned to the main program, where the request for continuation is posed.

The last subprogram, `Summary`, outputs the results when the **while...do** structure is completed.

The N-S charts for the solution are provided in Figure 10.20.

Figure 10.21 provides a list and run.

At this point, it should be clear that there are many ways to combine selection structures and loop structures. They can be nested or sequenced; and even if nested, there may be many levels of nesting. Loop structures provide great flexibility in the design of a solution to a problem.

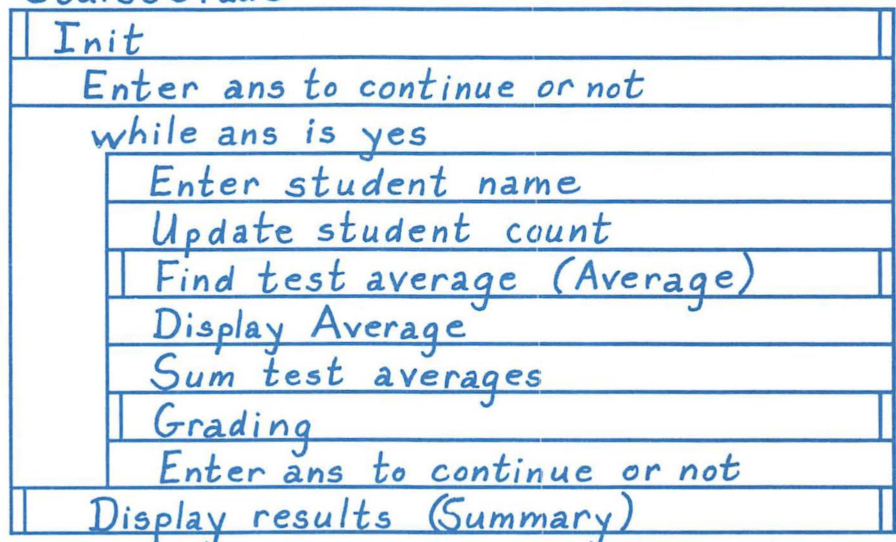


FIGURE 10.20 *(continued)*

Init

Set student count to 0
Set honors, pass and fail counts to 0
Set sum of averages to 0

Average

Initialize sum for tests
for index = 1 to 5
    Enter test score
    repeat until test score in valid range
    Add test score to sum
Find average of test scores

Grading

student avg ≥ 85 ?
NO:
    student avg ≥ 70 ?
    NO:
        Update failcount
        letter = "fail"
    YES:
        Update passcount
        letter = "pass"
YES:
    Update honors count
    letter = "Honors"
Output letter

Summary

Find course average
Output course average
Output honors, pass, fail count

FIGURE 10.20

```
program CourseGrade;
{Determines student grade based on average of 5 test scores}
 var
  name,            {name of student}
  ans              {answer to query}
   : string;
  stcount,          {student count}
  hct,                 {count of honor students}
  pct,                 {count of passing student}
  fct                 { count of failing students}
   : integer;
  testavg,            {student test average}
  sumave            {sum of averages}
   : real;

 procedure Init;
  {Initializes the sums and the counts}
 begin
  stcount := 0;
  hct := 0;
  pct := 0;
  fct := 0;
  sumave := 0.0;
 end;

 function Average : real;
  {Finds the average of a student's five test scores}
  var
   index,              {loop index}
   testscore,       {testscore}
   testsum        {sum of the scores}
    : integer;
 begin
  testsum := 0;
  for index := 1 to 5 do
   begin
    repeat
     write('      Enter test score #', index : 1, '  : ');
     readln(testscore)
    until (testscore >= 0) and (testscore <= 100);
    testsum := testsum + testscore;
   end;
  Average := testsum / 5
 end;
```

FIGURE 10.21 *(continued)*

```
procedure Grading (stavg : real;
         var hon, pass, fail : integer);
 {Determine grade for average and update counts}
 {stavg : test average}
 {hon : count of honor students}
 {pass : count of passing students}
 {fail : count of failing students}
 var
  letter     {letter grade received}
   : string;
begin
 if stavg >= 85.0 then
  begin
   hon := hon + 1;
   letter := 'Honors'
  end
 else if stavg >= 70.0 then
  begin
   pass := pass + 1;
   letter := 'Pass'
  end
 else
  begin
   fail := fail + 1;
   letter := 'Fail'
  end;
 writeln('      Grade : ', letter : 6)
end;

procedure Summary (coursesum : real;
         students, honors, pass, fail : integer);
 {Outputs summary data and average for course}
 {coursesum : sum of averages}
 {students : student count}
 {honors : count of honor students}
 {pass : count of passing students}
 {fail : count of failing students}
 var
  courseavg {average of all students}
   : real;
begin
 courseavg := coursesum / students;
 writeln;
 writeln('Summary of Course Results');
 writeln('Course average : ', courseavg : 7 : 2);
 writeln('Honors  :  ', honors : 4);
 writeln('Passing : ', pass : 4);
 writeln('Failing  :', fail : 4)
end;
```

FIGURE 10.21 *(continued)*

```
begin{*main program*}
 Init;
 write('Do you wish to enter test scores for a student ? (y/n)');
 readln(ans);
 while ans = 'y' do
  begin
   writeln;
   write('  Student name : ');
   readln(name);
   stcount := stcount + 1;
   testavg := Average;
   writeln('      Average : ', testavg : 6 : 2);
   sumave := sumave + testavg;
   Grading(testavg, hct, pct, fct);
   writeln;
   write('Do you wish to enter test scores for another student? (y/n)');
   readln(ans)
  end;
 Summary(sumave, stcount, hct, pct, fct)
end.{*main program*}
```

```
Text

  Student name : Julius Caesar
     Enter test score #1  : 76
     Enter test score #2  : 72
     Enter test score #3  : 88
     Enter test score #4  : 104
     Enter test score #4  : 94
     Enter test score #5  : 78
     Average :  81.60
      Grade :   Pass

Do you wish to enter test scores for another student? (y/n)n

Summary of Course Results
Course average :   81.60
Honors  :      0
Passing :      1
Failing  :     0
```

FIGURE 10.21

10.5 TYPICAL PROGRAMMING ERRORS

Typical errors a programmer might make on the Pascal code discussed in this chapter are as follows:

1. Creating never-ending or infinite loops.

   Example:
```
x:= 1;
repeat
 x := x + 1
until x < 0;
writeln(x);
```
2. Changing the index of a **for...do** loop inside the body of the loop.

   Example:
```
for m := 4 downto 1 do
 m := m - 1;
```
3. Not using a **begin-end** pair to enclose the body of a loop in a **while...do** or a **for...do** structure.

   Example:
```
while x < 6 do
 writeln('This is in the loop.');
writeln('Also in the loop.');
```
4. Not initializing summing and counting identifiers.
5. Including the initialization of a summing or counting identifier inside a loop.

   Example:
```
while b <> 1 do
 begin
  count := 0;
  readln(x);
  b := x * y;
  count := count + 1
 end;
```

NONPROGRAMMING EXERCISES

1. Classify each of the following Pascal statements as valid or invalid. If the statement is invalid, explain why.

   a. `for xbar := 5 do;`

   b.
```
repeat
  writeln('I'm Ok.');
  writeln('You're Ok.');
  x := x + 1
until x > 5;
```

c.
```
while (x > 0) and (y < 0) do
  begin
   writeln(x : 5, y : 5);
   x := x + 1;
   y := y - 1
  end;
```
d.
```
for j := 1 to 6 do
  begin
   writeln(j);
   j := j + 1
  end;
```
e.
```
x := 1;
while (x <> 0) do
  writeln('Hello! ');
```
f.
```
repeat
  x := y * 5 + sqr(y)
until;
```
g.
```
for x := 1 to 6 do
  begin
   sum := 0;
   count := 1;
   sum := sum + count;
   count := count +1
  end;
```
h.
```
while not(ans = 'yes') do
  readln(ans);
```
i.
```
repeat
  x := x + 2
  writeln(x)
until x := 10;
```
j.
```
for k := 10 to 5 do
  begin
   j := k + 5;
   writeln(j)
  end;
```

2. Write the Pascal equivalent of each of the following statements.
   a. Display the message "repeat" 10 times.
   b. Keep a running total of values for the identifier *x*.
   c. Display the countdown for a space shuttle launching, starting at 10 and ending at 1.
   d. Start *y* at 2 and keep doubling *y* as long as its value is less than 500.
   e. Enter values for name, and stop when a sentinel of *xxx* is entered.
   f. Use the identifier *d* to count by ones.
   g. Draw a rectangle and immediately erase it 20 times.
   h. Continue to read values for an identifier until a value of 0 is entered. The first value may be 0.
   i. Continue to read values for an identifier until a value of 0 is entered. The first value cannot be 0.
   j. Subtract the contents of location *y* from total to create a new value for total. Let *y* go from 2 to 20 by 2's and do the subtraction for each value of *y*.

3. Show the exact output for each of the following groups of Pascal statements.

a.
```
for i := 1 to 10 do
  begin
   j := i * 2;
   writeln(j : 4)
  end;
```

b.
```
k := 4;
while k > 1 do
  begin
   k := k - 1;
   writeln(k : 3)
  end;
```

c.
```
p := 1;
repeat
  p := p * 2;
  writeln(p : 6)
until p > 12;
```

d.
```
for a := 5 downto 2 do
begin
  b := a * 2;
  while b < 10 do
 begin
  writeln(b : 4);
  b := b + 2
 end
end;
```

e.
```
s := 0;
c := 0;
writeln('c' : 4, 's' : 4);
repeat
 c := c + 2;
 s := s + c;
 writeln(c : 4, s : 4)
until s > 20;
```

f.
```
x := 1;
repeat
 for i := 1 to x do
  write(i : 4);
 x := x + 2
until x > 5;
```

g.
```
a := 3;
b := 8;
expo := 1;
for i := 1 to b do
 expo := expo * a;
writeln(expo : 5);
```

PROGRAMMING EXERCISES

(G) 4. Write a program to output the message *one,* output *two,* output *one* twice, output *two* twice, then *one* and *two* three times, and so on, until *two* appears five times in sequence.

(G) 5. Write a program to find the sum of the numbers 2, 5, 8, . . . , 311.

(G) 6. Write a program that accepts a string and outputs the characters of that string on separate lines.

(G) 7. Write a program to find the sum of k terms of the series

$$1/2 + 2/3 + 3/4 + 4/5 + 5/6 + \dots,$$

where the value of k is to be input by the user.

(B) 8. Write a program to draw a line graph that charts the activity of a stock over a five-day period. Design a graph whose y-axis represents stock prices and whose x-axis represents the days of the week. Accept the price of the stock on each of the five days, and display the corresponding points on this graph. Connect the points representing the five prices.

(G) 9. Write a program using the **div** operator to accept as input a positive integer, and output a message indicating whether or not that integer is prime.

(G) 10. Given the function $y = 2x^2 + 5x - 4$, output a table of values (x,y) that satisfy the function. Select x values from 1 through 4 in steps of 0.2.

(B) 11. Write a program to produce a weekly profile for the five employees of Popda Corp. Input the name of each employee, the number of hours worked in the week, and the hourly wage. Output the gross weekly wage. In addition, calculate and output (a) the average number of hours worked for the week and (b) the average gross weekly wage for all employees.

(G) 12. Write a program that simulates the toss of a coin and counts and displays the number of heads, the number of tails, the percentage of heads, and the percentage of tails. The program should be capable of performing three different trials of 100, 1,000, and 10,000 tosses, respectively.

(B) 13. The Jobs 4U Employment Agency has been offered two plans of payment by the HAL Corporation in exchange for finding executive personnel for its management program. The first plan consists of the payment of a flat fee of $100 for each of the first 30 days the executive works. In the second plan, Jobs 4U is paid $0.01 the first day an executive works, $0.02 the second day, $0.04 the third, and so on, with each day's payment doubling the payment of the previous day, up to a maximum of 30 days. Write a program to determine and output the total payment under each plan. Also print a message indicating which of the plans to accept.

(B) 14. Use the information given in Exercise 13 to write a program to find the day number on which the payment from Plan 2 exceeds the payment from Plan 1.

(B) 15. The Bijou Theater charges $4.50 admission for adults, $3.00 for senior citizens, and $2.00 for children. Write a program to accept a count for each type of admission per day for three days. Output the total receipts for each day, the total receipts for the three-day period, the total admissions by type for the three-day period, and the average receipts per day over the three-day period.

(G) 16. Write a program to simulate the movement of time in hours and minutes on a 24-hour clock.

(G) 17. A palindrome is a word that reads the same forward as backward. For example, the word *noon* reads the same whether read from left to right or right to left. Write a program that accepts a word and displays a message indicating whether that word is a palindrome or not. *Hint:* Remember those string functions!

(B) 18. Write a program to process customer accounts for Lacy's Department Store. Input for each customer account includes the customer name, the unpaid balance in the account, and the number of days that account is outstanding. If the account is outstanding less than 30 days, a finance charge of 1.5 percent is added. If the account is outstanding for 30 days or more, a late payment fee of $5 is charged in addition to the finance charge. Output the customer name and new balance, and for those accounts outstanding 30 or more days, append a special message. Terminate the processing when *ZZZZZ* is entered for a customer name.

(G) 19. Write a program to produce a profile of an incoming first-year college student. Input includes a five-digit student ID number, a verbal aptitude test grade, and a mathematical aptitude test grade. Both tests are scored on a scale from 200 through 800. Output the average verbal score and the average mathematical score for the group. In addition, a separate message should be printed for any student for whom the sum of the two aptitude scores exceeds 1,200. Terminate the program with the endcode ID number 99999 and include checks for the entry of erroneous data.

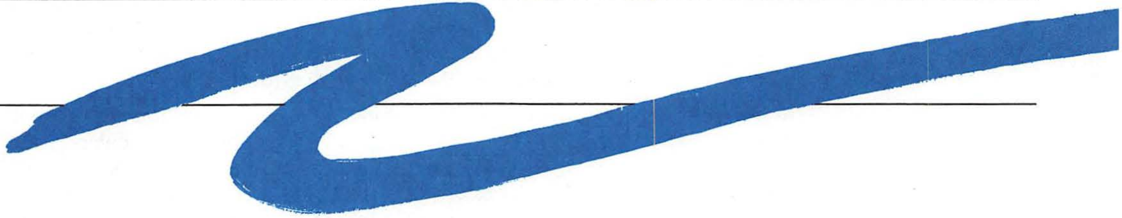
(B) 20. Maxie's Taxi Service operates eight taxis. Write a program to process the accounts for a business day. For each taxi, input the total number of trips for the day, the total mileage covered, and the total cash receipts. Output the totals for the day, the average cash receipt per taxi, the average mileage expended per trip per taxi, and the corresponding averages for the fleet of eight taxis.

(B) 21. When a sum of money, called the *principal,* or *p,* is deposited in an interest-bearing account and left on deposit for n interest periods, at a rate of interest per period i, the amount after n periods is given by the formula $a^b = p(1 + i)^n$. Write a program to show the growth of an initial deposit of $5,000 over a five-year period when interest is compounded quarterly (four times each year) at the rate of 2 percent per quarterly period. Output the results, showing the new amount on deposit at the end of each period. *Hint:* Although there is no exponential operator in Pascal, there are several ways to determine a: (1) create a subprogram that multiplies a by itself b times, and (2) create an expression that contains the **exp** and **ln** functions. These functions are not discussed in the text, but they are similar to other numeric functions. The expression `exp(b * ln(a))` is equivalent to a^b.

(B) 22. The Ace Auto Parts Company requires a weekly inventory report. Accept as input the part number, the quantity on hand at the beginning of the week, the number of new parts received during the week, and the number of parts sold during the week. Output the quantity on hand at the end of the week for each of the parts. Input should end when -1 is entered for a part number. For all parts whose inventory at the end of the week falls below 50, print a reorder message.

(G) 23. Write a program to simulate the dice game of Over/Under. A pair of dice is tossed and the player bets on whether the total shown on the dice is under 7, over 7, or 7. If the player's guess is wrong, he or she loses the amount of the bet. If the player's guess is correct on over or under that player doubles his or her money. If the player guesses correctly on 7, the player triples his or her money. The game should stake the player to a bankroll of $100 and should end at the player's discretion, or when the player's bankroll is gone. At no time should a bet be accepted that exceeds the amount of the player's current bankroll.

CHAPTER 11



Fundamental Data Structures

11.1 INTRODUCTION

Many computer applications require the manipulation of large quantities of data, which often necessitates distinct storage locations for individual data items. A practical example presented in Chapter 10 discussed portfolios that were being prepared for 200 conference attendees. Now let's imagine that the organizers of the conference wish to access information about each of these attendees. If that information is stored using computing facilities, the names of the participants alone would require 200 different locations, each with a unique identifier. Then those names are accessible at any time during the conference for billing, attendance listing, or any other conference-related activity. Additionally, identifiers have to be selected for the storage of other information pertaining to conference attendees, such as mailing addresses, telephone numbers, business affiliations, and the like.

The storage and maintenance of thousands of different identifiers becomes a monumental task. Loop structures cannot be used for the processing of 200 names, addresses, charges, and so on, because each data item would require a distinct identifier. MacPascal provides a method for the handling of large amounts of related data that must be kept in memory at the same time. This method makes use of two structured data types: *arrays* and *records*.

Arrays and records are composed of other data types and allow you to treat these other data types as a group as well as on an individual basis. Both arrays and records must therefore be defined in terms of these other data types.

11.2 THE ARRAY DATA TYPE

An array structure allows the storage of related data items of a single type in the same area of computer memory using a single identifier, rather than employing a different identifier for each value. That single identifier describes an entire

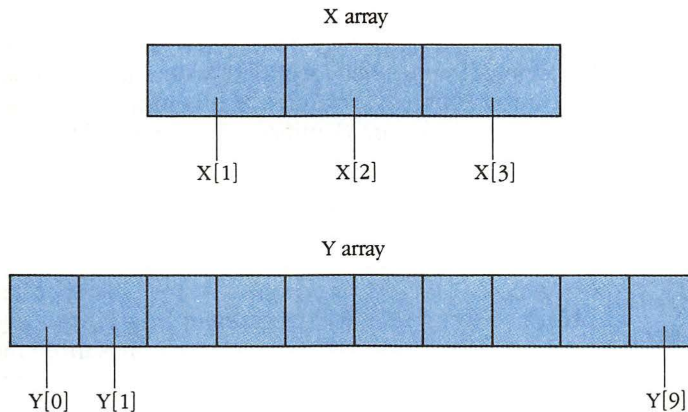


FIGURE 11.1

area of storage, and a reference marks the specific position in the area occupied by a particular item.

The simplest array structure is called a *one-dimensional array, vector,* or *list.* It is so named because all the data values can be thought of as being arranged in storage in a single column or single row. A pictorial representation of the internal storage for one-dimensional arrays is presented in Figure 11.1

The array name acts as a "family name," and the position acts as the "first name" of the specific data item. The position is called the *index* or *subscript* and can be an integer or character. It appears in square brackets ([ ]) and follows the array name. In this manner, each data item stored in the array structure can be uniquely referenced using (1) an identifier declared to be of the array type and (2) the subscript. For example, if the array identifier is `family`, `family[2]` represents the element in Position 2 of array `family`.

As with other data types, array types must be defined in the declaration section of a program. An array declaration statement indicates the number of storage locations reserved for the array, as well as the type of data to be stored in these locations. Although an array can be defined in either the **var** or **type** statement, it is better to define it in the **type** section, since only this definition of an array allows the array to be passed between subprograms.

FORMAT:
```
type
    <type identifier> = array[<index>] of <data type>;
var
    <identifier> : <type identifier>;
```

Variable identifiers declared to be <type identifier> are arrays of data items of <data type>. <index> may be composed of a subrange of integers or characters that represent the valid positions for elements in the array.

The subrange may be defined in a **type** statement, in which case the name of that range is enclosed in the square brackets. A subrange of integers or

characters also may be directly specified within the square brackets by entering the lowest element in a range, followed by two periods and the highest element in the range. All values between the lowest and the highest, inclusive, are allowable for array subscripts. <index> also indicates to the computer how much storage is needed for an identifier of this type.

Examples:

`type`
` alpha = array[1..3] of real;`
 	
`var`
` x : alpha;`
	The **type** statement defines **alpha** to be a data type that has three memory locations where real values can be stored. The identifier **x**, declared to be of type **alpha**, actually defines the three identifiers **x[1]**, **x[2]**, and **x[3]**.

`type`
` digit = 0..9;`
` beta=array[digit] of string;`
 	
`var`
` y : beta;`
	The type **digit** is a subrange of the type **integer**. The index in the declaration of **beta** reserves 10 memory locations for the array **y** that can be referenced by the identifiers **y[0]** through **y[9]**. Each of these 10 locations can store a data item of type **string**.

If reference is made to a location that is not in the range of values defined in the **type** statement, an error message results.

Once an array structure has been declared, specific data values may be stored in that structure in the same manner in which data are stored using any identifier.

Example:

`day[4] := 'Tuesday';`	The value **Tuesday** is stored in the array element **day[4]**.

Although it is possible to fill an entire array with data by using a large number of assignment statements, that process can become tedious. Entering data by using looping structures is much easier. The index of a looping structure serves as the subscript for an array element—and that's the key!

Examples:

`for i := 1 to 3 do` ` x[i] := i * 2;`	The index of the loop, `i`, is also used as the subscript for elements of the array `x`. When `i` is 1, the value 2 is stored in location `x[1]`, when `i` is 2, 4 is stored in `x[2]`, and so on.

`count := 1;`
`write('Enter name:  ');`
`readln(name [count]);`
`while name[count]<>'xxx'do`
` begin`
`  count := count + 1;`
`  write('Enter name:  ');`
`  readln(name[count])`
` end;`
`count:=count-1;`
`writeln (count,' items');`
	The identifier `count` acts as the subscript for array elements. Since the value of `count` is incremented by 1 after each pass through the array, each time a name is entered it is stored in a different element of the array. For example, the first name is stored in `name[1]`, the second in `name[2]`, and so on, until `xxx` is entered.

Problem: Write a program that accepts five integers and stores them in an array. Find their average. Display the array elements in a row and their average on the following row.

The description portion of the IPO chart should have some indication that an identifier represents an array.

By this time you should have become familiar with the use of IPO and N-S charts. Although you should continue to use these aids in designing your programs, they do not appear in the remainder of this chapter, so that we can concentrate on the programming structures themselves. Complete program listings and sample runs are still shown and described in detail.

Figure 11.2 shows a complete listing and sample run of the solution to the average problem.

The program is divided into three subprograms: a procedure for data entry, a function to find the average of the values, and a procedure to display the results.

The entry procedure reads data from the keyboard into the `value` array. The type `list` declared in the main program for `a` is also used for `value` in the procedure declaration. If the array were not declared in a **type** statement, it could not be passed as a parameter to a function or procedure. The index of the loop, `i`, is a subscript and locates the proper place in the array to store the entered data. The contents of `value` are passed to the main program and stored in the array `a`.

The function `Mean` that finds the average is passed the entire array called `a`. The parameter following the declaration of `Mean` shows the local name for this array `(data)`. In this function, the use of the index of the loop as an array subscript allows the computer to cycle through all the elements of the array adding each one, in turn, to the summing identifier `sum`.

The third subprogram, **Display**, is passed the contents of array **a** and **average** and outputs the contents of the array **value** by including a **write** statement inside a loop. Since the index of the loop serves as a subscript for the array, all the values of the array are displayed.

The advantage of passing entire arrays between subprograms is illustrated in Figure 11.3, in which a subprogram is referenced by different statements in the main program.

```
program ArrayAverage;
{Find the average of the elements in a 5-element array}
 type
  list = array[1..5] of integer;
 var
  a        {array of integers}
   : list;
  average {average of values}
   : real;

 procedure Entry (var value : list);
  {Enter data into the array}
  {value : array of integers}
  var
   i       {loop index}
    : integer;
 begin
  for i := 1 to 5 do
   begin
    write('Enter value #', i : 2, '    ');
    readln(value[i])
   end
 end;

 function Mean (data : list) : real;
  {Determines the average of the values in the array}
  {data : array of integers}
  var
   sum,          {sum of the values}
   x             {index of loop}
    : integer;
 begin
  sum := 0;
  for x := 1 to 5 do
   sum := sum + data[x];
  mean := sum / 5
 end;
```

FIGURE 11.2 *(continued)*

```
procedure Display (value : list;
        avg : real);
 {Displays the data stored in the array}
 {value : array of integers}
 {avg : average of array elements}
 var
  y         {index of loop}
   : integer;
begin
 for y := 1 to 5 do
  write(value[y] : 6);
 writeln;
 writeln('Average:    ', avg : 6 : 2);
end;

begin {*main program*}
 writeln('Array a:');
 Entry(a);
 writeln;
 average := Mean(a);
 write('Array a:   ');
 Display(a, average);
end.
```

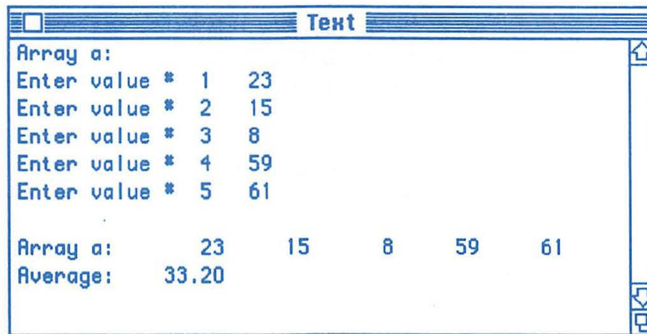


FIGURE 11.2

```
program ArrayAverage2;
{Find the averages of the elements in two 5-element arrays}
 type
  list = array[1..5] of integer;
 var
  a,         {first array of integers}
  b        {second array of integers}
   : list;
  average {average of values}
   : real;

 procedure Entry (var value : list);
  {Enter data into the array}
  {value : array of integers}
  var
   i     {loop index}
    : integer;
 begin
  for i := 1 to 5 do
   begin
    write('Enter value #', i : 2, '    ');
    readln(value[i])
   end
 end;

 function Mean (data : list) : real;
  {Determine the average of the values in the array}
  {data : array of integers}
  var
   sum,          {sum of the values}
   x            {index of loop}
    : integer;
 begin
  sum := 0;
  for x := 1 to 5 do
   sum := sum + data[x];
  mean := sum / 5
 end;
```

FIGURE 11.3 *(continued)*

```
  procedure Display (value : list;
           avg : real);
   {Displays the data stored in the array}
   {value : array of integers}
   {avg : average of array elements}
   var
    y       {index of loop}
     : integer;
  begin
   for y := 1 to 5 do
    write(value[y] : 6);
   writeln;
   writeln('Average:      ', avg : 6 : 2);
  end;

begin {*main program*}
 writeln('Array a:');
 Entry(a);
 writeln;
 average := Mean(a);
 write('Array a:   ');
 Display(a, average);
 writeln;
 writeln('Array b:   ');
 Entry(b);
 writeln;
 average := Mean(b);
 write('Array b:   ');
 Display(b, average)
end.
```

```
Text
Array a:
Enter value #  1   34
Enter value #  2   56
Enter value #  3   4
Enter value #  4   23
Enter value #  5   78

Array a:        34      56       4      23      78
Average:     39.00

Array b:
Enter value #  1   92
Enter value #  2   47
Enter value #  3   12
Enter value #  4   8
Enter value #  5   42

Array b:        92      47      12       8      42
Average:     40.20
```

FIGURE 11.3

The identifiers `a` and `b` are defined to be five-element arrays of integers (data type `list`) and are filled with data by the entry procedure, which is referenced in the `Entry(a);` and `Entry(b);` statements. The **`Mean`** and **`Display`** subprograms are also referenced twice, and the data elements and their average are displayed for both arrays.

Problem: Write a program that accepts the number of times a die is rolled, simulates the rolls, and counts the number of times each of the outcomes 1 through 6 occurs (the frequency of occurrence). Divide each of those counts by the total number of rolls and multiply by 100 to determine the percent frequency for each outcome. Output, in three labeled columns, the face value of a die, the number of occurrences of each value (frequency), and the percent frequency of each.

A complete listing and sample run of the solution is given in Figure 11.4.

The program is divided into subprograms that initialize the six frequency counters, roll and count the frequencies of occurrence, determine the percent frequencies, and display the results.

The first procedure, `Initialize`, sets each memory location in the array count to 0.

The array `freq` is used in the second subprogram, `Find`. This time, the array elements act as counters. The computer chooses a value from 1 to 6, and this value, in addition to representing the roll of a die, is also the subscript of one of the counters. For example, if **x** is 4, the counting statement, in this instance `freq[4] := freq[4] + 1;`, adds 1 to the count in `freq[4]`.

Once the die has been rolled and the number of rolls is complete, control passes to the third subprogram, `Percent`. In this procedure, the `freq` array helps to determine values for the elements of the `pct` array. Again, this calculation takes place in a loop.

The `Display` procedure outputs a table of the results of the rolls. The **writeln** statement that produces the heading for the three columns contains field descriptors that establish field widths for the display of the results. This statement comes before the loop structure and ensures that the heading is displayed once.

The **writeln** statement that displays the contents of the arrays `freq` and `pct` is inside the **for...do** loop and is executed six times, accessing a different array element with each pass through the loop. Field descriptors in this **writeln** statement output the information under the headings produced prior to the execution of the loop.

```
program DieOutcomeFrequency;
{Determine the frequency of occurrence of each outcome}
{for a given number of rolls of a die}
 type
  ilist = array[1..6] of integer;
  rlist = array[1..6] of real;
 var
  limit      {number of throws of a die}
   : integer;
  count      {array of numbers of occurrence for each outcome}
   : ilist;
  pctoftotal     {array of percentage of occurrence for each outcome}
   : rlist;

 procedure Initialize;
  {Set all counters to 0}
  var
   i     {loop index}
    : integer;
 begin
  for i := 1 to 6 do
   count[i] := 0
 end;

 procedure Find (limit : integer;
          var freq : ilist);
  {Count the number of occurrences of each outcome}
  {limit : number of trials}
  {freq : array of numbers of occurrence for each outcome}
  var
   j,           {loop index}
   x        {random integer from 1 to 6}
    : integer;
 begin
  for j := 1 to limit do
   begin
    x := abs(random mod 6 + 1);
    freq[x] := freq[x] + 1
   end
 end;
```

FIGURE 11.4 *(continued)*

```
procedure Percent (freq : ilist;
        max : integer;
        var pct : rlist);
 {Find the percentage of occurrence for each outcome}
 {freq : array of numbers of occurrence for each outcome}
 {max : number of throws of a die}
 {pct : array of percentage of occurrence for each outcome}
 var
  k     {loop index}
   : integer;
begin
 for k := 1 to 6 do
  pct[k] := freq[k] / max * 100
end;

procedure Display (freq : ilist;
        pct : rlist);
 {Output results}
 {freq : array of numbers of occurrence for each outcome}
 {pct : array of percentage of occurrence for each outcome}
 var
  face      {face value of a throw of a die}
   : integer;
begin
 writeln('FACE' : 13, 'FREQUENCY' : 13, 'PERCENT' : 11);
 for face := 1 to 6 do
  writeln(face : 10, freq[face] : 10, pct[face] : 14 : 2)
end;

begin
 Initialize;
 write('Enter number of trials:  ');
 readln(limit);
 Find(limit, count);
 Percent(count, limit, pctoftotal);
 Display(count, pctoftotal)
end.
```

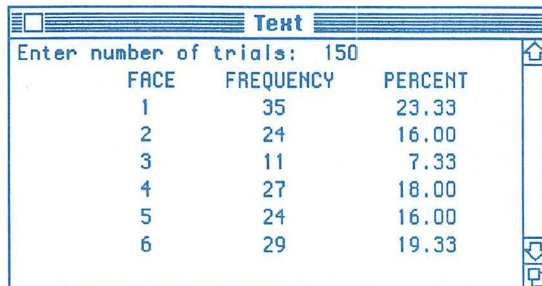


FIGURE 11.4

Problem: Write a program to graph the function $y = x^2$ where the values of the x coordinates range from -8 to 8 in increments of 1.

A complete listing of the program solution and the output in the **Drawing** window are shown in Figure 11.5.

```
program LineGraph;
{Graph the function y = x²}
 const
  xtran = 100;     {horizontal origin translation}
  ytran = 100;     {vertical origin translation}
  xscale = 10;     {scale for x axis}
  yscale = 1;       {scale for y axis}
 type
  point = array[-8..8] of integer;
 var
  x,               {actual x coordinates for graph}
  y             {actual y coordinates for graph}
   : point;
  xdraw,         {drawing x coordinates}
  ydraw        {drawing y coordinates}
   : point;

 procedure ActualCoordinates;
  {Determines coordinates of points on graph}
  var
   i      {loop index}
    : integer;
 begin
  for i := -8 to 8 do
   begin
    x[i] := i;
    y[i] := sqr(x[i])
   end
 end;


 procedure DrawingCoordinates;
  {Determines plotting coordinates for points on graph}
  var
   i      {loop index}
    : integer;
 begin
  for i := -8 to 8 do
   begin
    xdraw[i] := x[i] * xscale + xtran;
    ydraw[i] := -y[i] * yscale + ytran
   end
 end;
```

FIGURE 11.5 *(continued)*

```
procedure DrawAxes;
 {Draws axes and hash marks}
 var
  k : integer;
begin
 drawline(10, 100, 190, 100);
 k := 20;
 repeat
  drawline(k, 98, k, 102);
  k := k + 10;
 until k > 189;
 drawline(100, 10, 100, 190);
 k := 20;
 repeat
  drawline(98, k, 102, k);
  k := k + 10
 until k > 189
end;

procedure LabelAxes;
 {Put scale values on axes}
 var
  i,           {cycles through x values}
  j,           {cycles through y values}
  v        {y axis scale values}
   : integer;
begin
 i := -8;
 repeat
  moveto(xdraw[i] - 7, 115);
  writedraw(x[i] : 1);
  i := i + 4
 until i > 8;
 j := 40;
 repeat
  moveto(110, j + 5);
  v := -j * yscale + ytran;
  writedraw(v : 2);
  j := j + 40
 until j > 160
end;


procedure DrawGraph;
 {Draws function on coordinate system}
 var
  j      {loop index}
   : integer;
```

FIGURE 11.5 *(continued)*

```
  begin
   moveto(xdraw[-8], ydraw[-8]);
   for j := -7 to 8 do
    lineto(xdraw[j], ydraw[j]);
  end;

begin
 ActualCoordinates;
 DrawingCoordinates;
 DrawAxes;
 LabelAxes;
 DrawGraph
end.
```

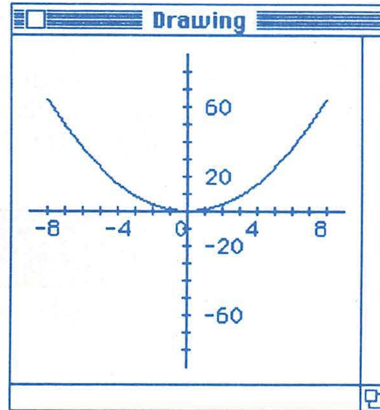


FIGURE 11.5

The solution contains five procedures: `ActualCoordinates` (determines the coordinates of points satisfying the equation), `Drawing-Coordinates` (transforms the actual coordinates into the coordinates of the **Drawing** window), `DrawAxes` (draws the x and y axes with hash marks indicating scale values); `LabelAxes` (labels the x and y axes with the scale values chosen); and `DrawGraph` (plots the points and connects them).

After declaring an appropriate subscript range (-8 to 8) for the arrays storing the x and y values for the function, the values are determined and stored as array elements in the procedure `ActualCoordinates`.

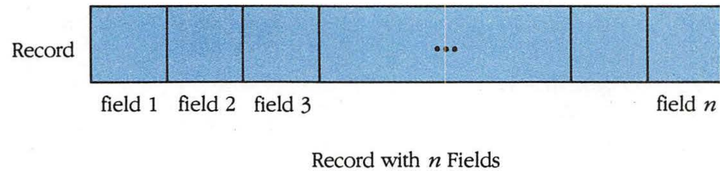
`DrawingCoordinates` converts the x and y array elements to pixel locations (`xdraw` and `ydraw` array elements) in the **Drawing** screen by first multiplying each by a scale factor and adding a translation parameter. The scale factor is determined by how many pixels comprise a unit on an axis. The scale factor for the x axis (`xscale`) is 10, so 10 pixels are equivalent to one unit. Similarly, `yscale` is 1, so 1 pixel on the y axis is equivalent to one unit. The translation parameters (`xtran` and `ytran`) move the origin of the graph from the upper left corner to another position. Since both `xtran` and `ytran` are 100, the origin moves 100 pixels to the right of the upper left corner and 100 below.

`DrawAxes` displays the lines that serve as the x and y axes and uses **repeat...until** loops to put hash marks on these axes to represent particular values. Selected hash marks are labeled in `LabelAxes`. It is not necessary to label all hash marks.

The last procedure, `DrawGraph`, plots the first or leftmost point in the **Drawing** window using the `xdraw` and `ydraw` values for that point. The **lineto** procedure in the loop that follows connects each preceding point to the one that follows.

11.3 THE RECORD DATA TYPE

A record structure allows related data items of different types to be referenced using a single identifier. The data items are arranged in fields in the record structure. Figure 11.6 depicts the structure of a record containing *n* fields.



Record with *n* Fields

FIGURE 11.6

The **record** type is given an identifier, as is each field in the record.

FORMAT:
```
type
  <type identifier> = record
    <field identifier> : <data type>;
    <field identifier> : <data type>;
       ...
    <field identifier> : <data type>
  end;
var
  <identifier> : <type identifier>;
```

Variable identifiers declared to be of type <type identifier> consist of several fields. Each field is defined by specifying <field identifier> and the data type for that identifier. The **end** statement closes the record definition.

When records are defined, the field names listed in the **type** declaration must be used in conjunction with the identifier declared in the **var** section defined to be of that **record** type in order to access the data in a field.

Example:

```
type
 alpha = record
   name : string;
   age : integer
 end;

var
 census : alpha;
```

The record of type `alpha` has two fields: `name`, of type **string**; and `age`, of type **integer**. The identifier `census`, declared in the **var** statement, is defined to have the same structure.

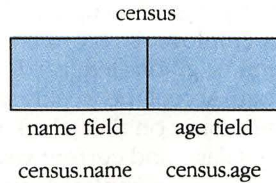


FIGURE 11.7

Figure 11.7 shows the structure of the record **census** of type **alpha**.

To access the information in the **name** field, the identifier **census.name** is used. To access information in the **age** field, the identifier **census.age** is used. In general, the field of a record can be accessed by appending the field name to the record identifier and separating them by a period.

Example:

```
census.name := 'Tom Jones';
census.age := 39;
writeln(census.name:15);
```

The value **Tom Jones** is stored in the **name** field of the record **census**. The value 39 is stored in the **age** field of the same record. The **writeln** statement displays the contents of the **name** field.

Instead of continually writing the record name before each field name, MacPascal supplies the **with** statement to facilitate the handling of records with many fields.

FORMAT:
```
with <record identifier> do
    <statement>;
```

Any reference to a field of <record identifier> in the **with** statement does not have to be preceded by <record identifier> or a period; just use the field name. If <statement> is a compound statement, it must be enclosed in a **begin-end** set.

Example:

```
with census do
 begin
  name := 'Tom Jones';
  age := 39;
  writeln(name:15)
 end;
writeln(census.age:3);
```

The record identifier **census** is not used as a prefix to access information in the **name** and **age** fields of that record since the references to these fields are contained within the **with** structure. The last **writeln** statement must use the prefix **census** since it is not contained in the **with** structure.

Problem: Write a program to update the payroll record for an employee of ABC Company. Each employee's record consists of the employee's name, a commission classification or grade (letter *a, b, c,* or *d*), a weekly base salary, and a year-to-date salary total. Accept the weekly sales for an employee, find the gross salary (base plus commission on the sales), update the year-to-date total, and output the name, gross salary, and current year-to-date figure.

Figure 11.8 gives a complete listing and a sample run of the solution.

The type declaration defines a record type, `salary`, with four fields: `name`, `grade`, `base`, and `ytd` (year-to-date). Since `empl` is declared to be of type `salary`, any reference to the fields of `empl` must be preceded by `empl.`

```
program Payroll;
{Update payroll data and output results for an employee}
 type
  salary = record
    name : string;
    grade : char;
    base : real;
    ytd : real
   end;
 var
  empl      {record for the employee}
   : salary;
  pay,                    {weekly pay for the employee}
  sales     {weekly sales by the employee}
   : real;

 procedure EnterRecord;
  {Enter data for individual employee record}
 begin
  with empl do
   begin
    write('Enter employee name:  ');
    readln(name);
    repeat
     write('Enter grade:  ');
     readln(grade)
    until ((grade = 'a') or (grade = 'b') or (grade = 'c') or (grade = 'd'));
    write('Enter base salary:  $');
    readln(base);
    write('Enter year-to-date earnings:  $');
    readln(ytd)
   end
 end;
```

FIGURE 11.8 *(continued)*

```
procedure EnterSales;
 {Enter weekly sales by each employee}
begin
 write('Enter weekly sales for ', empl.name, ':  $');
 readln(sales)
end;

procedure Update;
 {Update employee record}
 var
  pct,             {commission percentage}
  comm    {amount of commission}
   : real;
begin
 with empl do
  begin
   case grade of
    'a' :
     pct := 0.18;
    'b' :
     pct := 0.13;
    'c' :
     pct := 0.10;
    otherwise
     pct := 0.06
   end;
   comm := sales * pct;
   pay := base + comm;
   ytd := ytd + pay
  end
end;
procedure Display;
 {Output employee record information}
begin
 with empl do
  writeln(name : 15, pay : 15 : 2, ytd : 15 : 2);
end;

begin {  main program  }
 EnterRecord;
 EnterSales;
 Update;
 writeln;
 page;
 writeln('Payroll Summary');
 writeln;
 writeln('Employee' : 15, 'This Pay' : 15, 'Year-To-Date' : 15);
 writeln('---------------------------------------------');
 Display
end.
```

FIGURE 11.8

```
Text
Payroll Summary

        Employee      This Pay   Year-To-Date
---------------------------------------------
           Smith      25090.00       29055.60
```

FIGURE 11.8

The program structure consists of four procedures: one to enter the record data (`EnterRecord`), another to enter the sales data (`EnterSales`), a third to find the weekly salary and update the year-to-date total in the record (`Update`), and a last one to display a summary of the results (`Display`). To simplify the understanding of the concept of records, all identifiers are global.

The data entry is divided into two subprograms, because it is likely that the employee records come from a file while the weekly sales are entered from the keyboard. If such an employee file exists, only the `EnterRecord` procedure has to be modified.

The first procedure, `EnterRecord`, contains a **with** structure, so field names are not preceded by `empl`. A **repeat...until** loop is included to ensure that only valid employee's grades are accepted. Since no **with** statement is included in the second procedure, `empl.name` must be used in the **write** statement.

Array and record structures can be combined to form an array of records or records with array fields. These combined structures are very useful when storing many records.

Examples:

```
type
 salary = record
   name : string;
   grade : char;
   base : real;
   ytd : real
  end;
 force=array[1..3] of salary;
var
empl : force;
```

After the record `salary` is defined, `force` is defined to be an array of records of type `salary`. The identifier `empl` represents an array of records. Figure 11.9 shows the structure of `empl`.

The array elements are `empl[1]`, `empl[2]`, and `empl[3]`. Each has 4 fields, and each field name must be preceded by the record identifier for a particular element in the array. For example, `empl[2].grade` represents the `grade` field of the record `empl[2]`.

```
type
 room = record
  id : string;
  dim : array[1..3] of real
 end;

var
 abode : room;
```

The record type `room` has two fields: the first, `id`, represents the name of a room; and the second, `dim`, is an array of three reals. The identifier `abode` is of type `room`, and its structure is shown in Figure 11.10.

To access the data in the second element of the second field of `abode`, use the identifier `abode.dim[2]`.

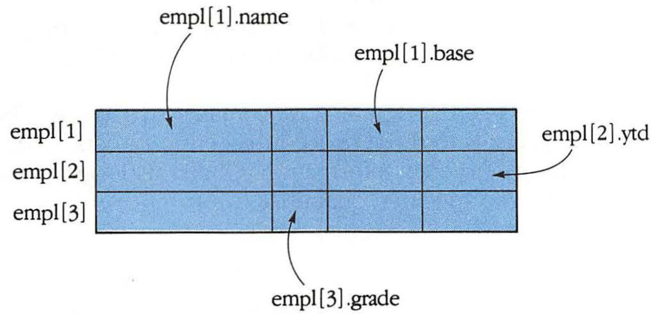


FIGURE 11.9

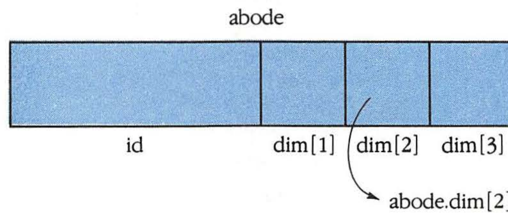


FIGURE 11.10

Problem: Write a program to update the payroll records for three employees of the ABC Company. Each employee's record consists of the employee's name, a commission classification or grade (letter *a, b, c,* or *d*), a weekly base salary, and a year-to-date salary total. Accept the weekly sales for the employees, find the gross salaries (base salaries plus commissions on sales), update the year-to-date totals, and output the names, gross salaries, and current year-to-date figures.

The solution to this problem requires minor modifications of the solution to the same problem with one employee shown in Figure 11.8. Figure 11.11 lists the program and a sample run for the three employees.

The declaration statements create an array of records. The size of the array is determined by the identifier **max** appearing in the constant declaration portion of the program. The structure of each record in this array is the same as the record defined in the preceding example. Arrays **pay** and **sales** are also defined in this section. The identifiers **i** and **k** are also declared for indices for the loops used in the program.

The only changes in the procedures are that **empl[i]**, **pay[i]**, and **sales[i]** replace **empl**, **pay**, and **sales**, respectively, in the first three procedures. A similar replacement occurs in **Display** where the index of the loop is **k** rather than **i**.

The main program now contains two loops. The body of the first loop contains calls to three procedures: entry of data, determination of salaries, and updating of record information. Once this loop is completed, the second loop displays the specified quantities.

```
program Payroll;
{Update employee payroll data and produce payroll report}
 const
  max = 3;        {number of employees}
 type
  salary = record
    name : string;
    grade : char;
    base : real;
    ytd : real
   end;
  salesforce = array[1..max] of salary;
  check = array[1..max] of real;

 var
  empl      {array of records for each employee}
   : salesforce;
  pay,            {array of weekly pays for employees}
  sales      {array of weekly sales by each employee}
   : check;
  i, k          {loop indices}
   : integer;
```

FIGURE 11.11 *(continued)*

```
procedure EnterRecord;
 {Enter data for individual employee record}
begin
 with empl[i] do
  begin
   write('Enter employee name:  ');
   readln(name);
   repeat
    write('Enter grade:  ');
    readln(grade)
   until ((grade = 'a') or (grade = 'b') or (grade = 'c') or (grade = 'd'));
   write('Enter base salary:  $');
   readln(base);
   write('Enter year-to-date earnings:  $');
   readln(ytd)
  end
end;

procedure EnterSales;
 {Enter weekly sales by each employee}
begin
 write('Enter weekly sales for ', empl[i].name, ':  $');
 readln(sales[i])
end;

procedure Update;
 {Update employee record}
 var
  pct,           {commission percentage}
  comm    {amount of commission}
   : real;
begin
 with empl[i] do
  begin
   case grade of
    'a' :
     pct := 0.18;
    'b' :
     pct := 0.13;
    'c' :
     pct := 0.10;
    otherwise
     pct := 0.06
    end;
    comm := sales[i] * pct;
    pay[i] := base + comm;
    ytd := ytd + pay[i]
   end
 end;
```

FIGURE 11.11 *(continued)*

```
procedure Display;
 {Output employee record information}
begin
 with empl[k] do
  writeln(name : 15, pay[k] : 15 : 2, ytd : 15 : 2);
end;

begin { main program }
 for i := 1 to max do
  begin
   EnterRecord;
   EnterSales;
   Update;
   writeln
  end;
 page;
 writeln('Payroll Summary');
 writeln;
 writeln('Employee' : 15, 'This Pay' : 15, 'Year-To-Date' : 15);
 writeln('---------------------------------------------');
 for k := 1 to max do
  Display
end.
```

```
Text
Payroll Summary

       Employee       This Pay   Year-To-Date
---------------------------------------------
          Green         118.00         218.00
          White         226.00         426.00
         Phelps         330.00         630.00
```

FIGURE 11.11

Although the solution is designed for three employees, it can be extended to handle any number of employees by changing the upper limit (**max**) on the loops in the program and the declaration of the sizes. Since the upper limit is defined in the **const** declaration portion of the program, only this declaration statement would have to be changed to accommodate additional employees.

It may also be more reasonable to use a **while...do** or **repeat...until** structure to process the employee listing, in which case counters and sentinel values would have to be declared.

11.4 OPERATIONS ON DATA STRUCTURES

There are many applications for arrays and records in programming. Some of the more frequently used applications involve searching through the data for a specific value, sorting data according to given criteria, or combining two existing lists to form one. The following sections highlight the special techniques of programming logic used to implement these applications.

SEARCHING

A typical processing procedure is called a *search.* This application arises when you are looking for an element that has a specific value, or "target value." In this case, the search stops once a match is found. You can also search for several elements that might match a given target value. Even if a match is found, you have to continue to search through the entire structure for additional matches.

Many methods can be employed to search through a list of items. The simplest one to understand and implement is called a *linear search.* It involves testing each item in a list, sequentially, from the first item to the last item. We are using arrays to illustrate the search procedure.

Example:

```
ct := 0;
found := false;
repeat
 ct := ct + 1;
 if A[ct] = target then
  found := true
until (found) or (ct = max);
if found then
 writeln('Match at element ',ct)
else
 writeln('No match');
```

The identifier `ct` is used to cycle through the array `A`. The Boolean `found` is used to stop the search if a match is found (if the target value equals an array value). The **repeat...until** structure ends if a match is found or if all the elements in the array have been tested (`ct = max`, the maximum number of items in the array). The selection structure following the loop determines which condition causes the loop to terminate.

SORTING

It is sometimes necessary to reorganize data in a specified order. For example, numeric values may be placed in size order, that is, in ascending order or descending order. String data are frequently reorganized in alphabetical order. The operation that accomplishes this ordering is called *sorting.* There are many algorithms that are used to sort data. One of the easier methods to understand

and implement is a technique known as the *bubble sort.*

To illustrate this technique, assume that the data in an array are to be reorganized so that the values are in ascending order. Assume further that the data have all been loaded in an array. A Boolean data value is used to terminate the sort. Such a value is called a *flag* and is set equal to *true* at the beginning of the sort. The array is searched to locate any pair of adjacent elements that are not in ascending order. Whenever such a pair is located, two things are done: first, the flag is set equal to *false,* and second, the values of the two adjacent elements are interchanged within the array. When a pass through the array is completed, the flag is tested. If the flag remains *true,* it indicates that no interchanges were made, and the values in the array are in the required order. However, if the flag has been changed to *false,* at least one interchange has occurred. In that event, the flag is reset to *true,* and the entire array search is repeated.

This procedure, though comprehensive, is not the most efficient method for sorting because it always requires at least one extra search through the array. However, it is straightforward and not difficult to program. Figure 11.12 illustrates a walkthrough of the bubble sort process with a five-element array. The array contains the following data:

```
        Array
  5   3   8   0   7
```

In this figure, *X* implies that an exchange of adjacent elements is required, and *NX* implies that no such exchange is required.

Any sorting technique can be used to alphabetize a list of strings. If the array is sorted in ascending order and is declared to contain string data, the sort arranges the list in alphabetical order.

	Search #1	Search #2	Search #3	Search #4
Initial FLAG Condition	TRUE	TRUE	TRUE	TRUE
	5 X 3 8 0 7	3 NX 5 0 7 8	3 X 0 5 7 8	0 X 3 5 7 8
	3 5 NX 8 0 7	3 5 X 0 7 8	0 3 NX 5 7 8	0 3 NX 5 7 8
	3 5 8 X 0 7	3 0 5 NX 7 8	0 3 5 NX 7 8	0 3 5 NX 7 8
	3 5 0 8 X 7	3 0 5 7 NX 8	0 3 5 7 NX 8	0 3 5 7 NX 8
	3 5 0 7 8	3 0 5 7 8	0 3 5 7 8	0 3 5 7 8
Final FLAG Condition	FALSE	FALSE	FALSE	TRUE

FIGURE 11.12

Example:

```
repeat
 order := true;
 for i := 1 to max - 1 do
   if B[i] > B[i + 1] then
     begin
       temp := B[i];
       B[i] := B[i + 1];
       B[i+1] := temp;
       order := false
     end
until order;
```

The **repeat...until** structure cycles through the entire array until no exchanges are required. The flag `order` is *true* until an exchange is made. If a complete pass is made without any exchange, the list is sorted and the process ends. If an element is greater than the next element, the values are interchanged, the flag is set to *false,* and the sort continues.

MERGING

Merging is a process in which the data in two structures can be combined into one. You must also be concerned with the ordering of the data values within the structures and with the elimination of any duplicates in the merge. If two structures are already sorted in the same order, merging these lists is simplified. Figure 11.13 illustrates the process that merges two arrays of different sizes (10 and 5 elements), each arranged in ascending order, into one master array. The array that results from the merging is also arranged in ascending order with the duplicates removed.

Let A and B be the given arrays and C the merged array. Three counters, representing pointers, are used to keep track of the positions of related elements in the three arrays. Comparisons start with the elements at the beginning of each of the arrays (all subscripts are initialized at 1).

The first comparison matches the first element in the A array with the first element in the B array (see Figure 11.13b). The smaller of these two elements is placed in the first position of the C array. The pointers of the array that contains the smaller value and of the C array are moved to the next element. The pointer of the array containing the larger value is left unchanged (Figures 11.13c and 11.13d). If the values of the two elements compared are the same (Figure 11.13e), the value is put in the next available position in the C array and then all three pointers are moved to the next elements to be considered (Figure 11.13f). The comparisons continue until one of the pointers (in this case, the pointer to the B array) goes beyond the last element. At that point, the comparison loop terminates and the procedure continues in one of two directions. Each direction involves completing the C array with the elements still remaining in the original array whose pointer did not go beyond the last element in the comparison loop (in this case, the A array). The final merged array appears in Figure 11.13g.

The two empty spaces at the end of the merged C array are due to the elimination of the duplicates 3 and 8 in the merging process.

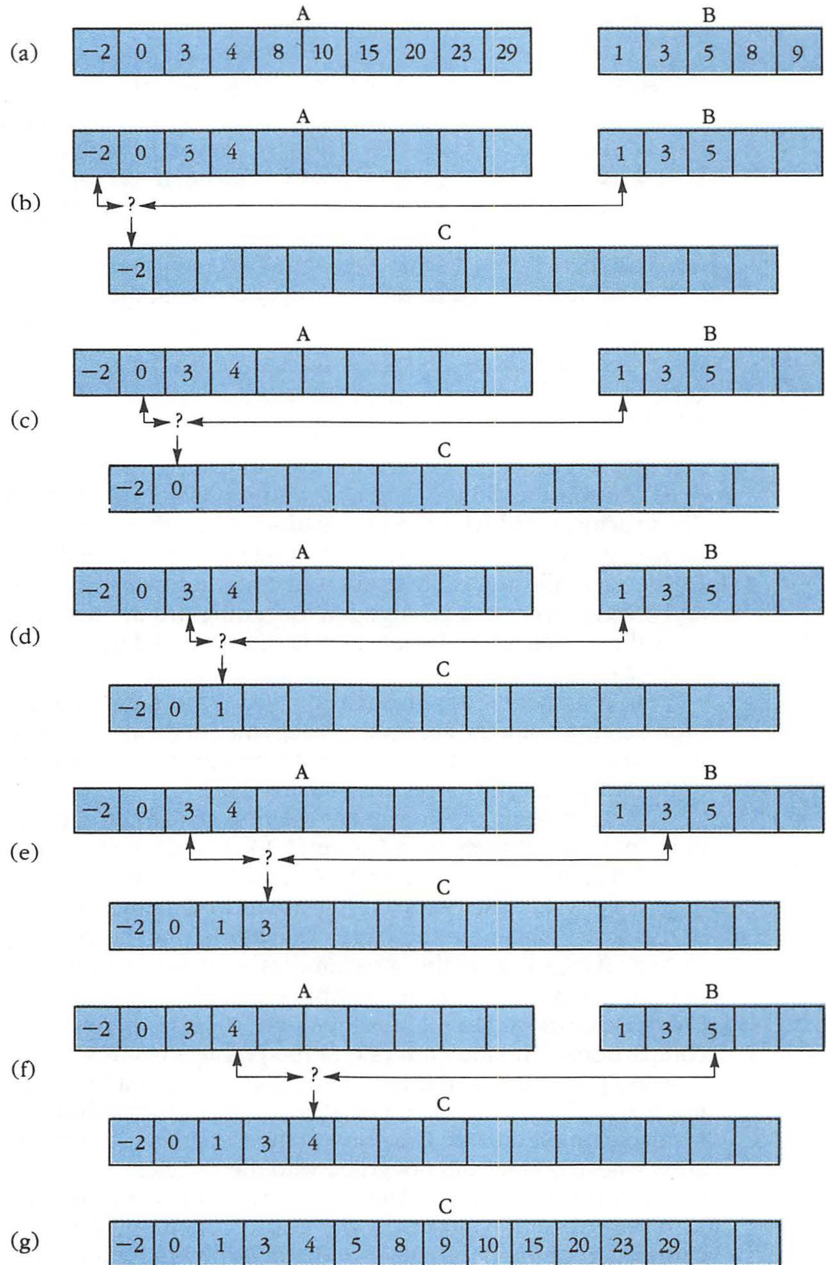


FIGURE 11.13

Example:

```
i := 1;
j := 1;
k := 1;
repeat
 if A[i] < B[j] then
  begin
   C[k] := A[i];
   i := i + 1
  end
 else if A[i] > B[j] then
  begin
   C[k] := B[j];
   j := j + 1
  end
 else
  begin
   C[k] := A[i];
   i := i + 1;
   j := j + 1
  end;
 k := k + 1
until (i > 10) or (j > 5);
if i > 10 then
 for index := j to 5 do
  begin
   C[k] := B[j];
   k := k +1
  end
else
 for index := i to 10 do
  begin
   C[k] := A[i];
   k := k + 1
  end
```

The counters i, j, and k for the A, B, and C arrays, respectively, keep track of the positions in each array of the elements being compared (for A and B) and the destination position in the merged array C. The **repeat...until** structure compares the elements in A and B and selects the smaller to be placed in C. The counter for the array containing the chosen element is updated. If two elements are equal, the one from the A array is arbitrarily selected and both counters are updated. In any case, the counter for array C is updated. The comparison process continues until the counter for A or B exceeds its maximum value. At this point, the values in the remaining array are added to the end of the merged array. The declaration section of the program must contain values that define the maximum size of each.

Problem: Lynx, a chain of department stores, has a list of records each containing a four-digit account number and a customer name. Each store has its own list. Lynx would like to combine the lists from two of its branches into one master list and be able to search the merged list for the name corresponding to a given account number. Write a program that enters the data for two lists, sorts the lists, merges them into a master list with duplicates eliminated, and is capable of finding the name of a customer corresponding to an account number.

Figure 11.14 gives a complete listing and sample run of the solution to the Lynx problem.

```
program MailList;
{Searching, sorting and merging arrays of records}
{on customers of the Lynx Department Store}
 const
  max = 3;
  mermax = 6;
 type
  person = record
    acctno : integer;
    name : string
   end;
  customer = array[1..mermax] of person;

 var
  A,            {array of customer information from one store}
  B,            {array of customer information from a second store}
  C         {array containing merged list}
   : customer;
  key        {account number for a sought-after customer}
   : integer;
  count      {number of duplicates in merged array}
   : integer;

procedure Entry (var X : customer;
         size : integer);
 {Enter data into array of records}
 {X : array of customer information}
 {size : number of customers to process}
 var
  i     {loop index}
   : integer;
begin
 writeln('Enter records for customer list');
 for i := 1 to size do
  begin
   write(i : 3, '    Account Number  : ');
   readln(X[i].acctno);
   write('       Name             :  ');
   readln(X[i].name)
  end
end;
```

FIGURE 11.14 *(continued)*

```
procedure Search (D : customer;
        target : integer);
 {Find customer with given account number}
 {D : array of customer information}
 {target : account number of sought-after customer}
 var
  i      {loop index}
   : integer;
  flag      {indicates when match has been found}
   : boolean;
begin
 i := 0;
 flag := false;
 repeat
  i := i + 1;
  if D[i].acctno = target then
   flag := true
 until (flag) or (i = mermax);
 if flag then
  writeln('Customer : ', D[i].name : 20)
 else
  writeln('Inaccurate account number')
end;

procedure Sort (var D : customer);
 {Sort array in ascending order according to account numbers}
 {D : array of customer information}
 var
  i      {loop index}
   : integer;
  X      {temporary location for switching information in records}
   : person;
  nochange       {indicates when a switch has been made}
   : boolean;
begin
 repeat
  nochange := true;
  for i := 1 to max - 1 do
   begin
    if D[i].acctno < D[i + 1].acctno then
     begin
      X := D[i];
      D[i] := D[i + 1];
      D[i + 1] := X;
      nochange := false
     end
   end
 until nochange
end;
```

FIGURE 11.14 *(continued)*

```
procedure Merge (var noofdup : integer);
 {Merge two arrays of identical size and eliminate duplicates}
 {noofdup : number of duplicates in merged array}
 var
  i, j, k, index       {loop indices}
   : integer;
begin
 i := 1;
 j := 1;
 k := 1;
 noofdup := 0;
 sort(A);
 sort(B);
 repeat
  if A[i].acctno > B[j].acctno then
   begin
    C[k] := A[i];
    i := i + 1;
   end
  else if A[i].acctno < B[j].acctno then
   begin
    C[k] := B[j];
    j := j + 1;
   end
  else {duplicates}
   begin
    C[k] := A[i];
    i := i + 1;
    j := j + 1;
    noofdup := noofdup + 1
   end;
  k := k + 1
 until (i > 3) or (j > 3);
 if i > 3 then
  for index := j to 3 do
   begin
    C[k] := B[j];
    k := k + 1
   end
 else
   for index := i to 3 do
    begin
     C[k] := A[i];
     k := k + 1
    end
 end;
```

FIGURE 11.14 *(continued)*

```
procedure Display (D : customer;
          size : integer);
  {Outputs information in merged array}
  {D :  array of customer information}
  {size : number of elements in array}
  var
   i      {loop index}
    : integer;
 begin
  for i := 1 to size do
   writeln(D[i].acctno : 12, D[i].name : 15)
 end;


begin(*main*)
 Entry(A, max);
 writeln('Customer list A as entered');
 Display(A, max);
 writeln;
 Entry(B, max);
 writeln('Customer list B as entered');
 Display(B, max);
 page;
 writeln('Merged list of customers from A and from B with duplicates eliminated');
 Merge(count);
 Display(C, mermax - count);
 writeln;
 write(' Enter account number for search in merged list:');
 readln(key);
 Search(C, key)
end.
```

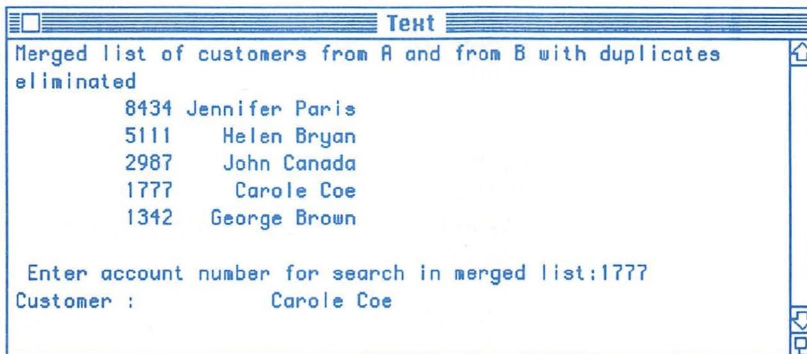


FIGURE 11.14

The program is composed of five procedures: `Entry` (loads data into two lists), `Search` (looks for an account number in the merged array), `Sort` (sorts the two original arrays in descending order), `Merge` (combines arrays `A` and `B` into one array, `C`), and `Display` (shows the merged list). The procedure `Sort` precedes the procedure `Merge` in the program listing since `Merge` calls `Sort` and `sort` must be defined before it is used.

Rather than using one-dimensional arrays for searching, sorting, and merging, as in the three previous examples shown in the text, the solution employs an array of records where the target value for the search is one of the fields in a record.

11.5 TYPICAL PROGRAMMING ERRORS

The following are typical programming stumbling-blocks.

1. Using out-of-range array elements.

   Example:
```
type
 list : array[1..10] of integer;
var
 a : list;
 b, c : integer;
begin
 b := 11;
 c := a[b];
```
2. Not enclosing subscripts in square brackets.

3. Not furnishing the complete identifier for the field of a record data type.

   Example:
```
type
 state = record
   name : string;
   pop : integer
  end;
var
 a : state;
begin
 readln(name);
```
4. Not predeclaring arrays in the type statement if the arrays are to appear as parameters in a subprogram.

   Example:
```
procedure test(a : array[1..3] of
integer);
```

NONPROGRAMMING EXERCISES

1. Classify each of the following Pascal statements as valid or invalid. If the statement is invalid, explain why.

a.
```
type
  a = array of integer;
```
b.
```
type
  b = array['a'..'d'] of string;
```
c.
```
type
  structure = record
    a : real;
    b, d : integer;
    c : char
   end;
```
d.
```
type
  outline = record
    heading, body : string;
    class : char
   end;
  termpapers = array[1..100] of outline;
```
e.
```
with q do
  field1 := field1 + 1;
```
f.
```
if family[i].son <> 'dependent' then
  dep := dep + 1;
```
g.
```
test{i} := test{j} + grade;
```
h.
```
repeat
  sum := sum + grade[name[i]];
  i := i + 1
until i > 10;
```
i.
```
n := 1;
while n < 10 do
  begin
   with r[n] do
    begin
     readln(field1);
     total := total + field1
    end;
   n := n + 1
  end;
```
j.
```
with r[j] do
  while j < 5 do
   begin
    a[j] := field2 * factor;
    j := j + 1
   end;
```

2. Given a one-dimensional array T with the following values—

T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]
9	−3	1	0	5	−5	12	−8	25

—evaluate each of the following:

a. `T[2] + 2*T[6]`
b. `T[ T[3] ] - T[2]`
c. `T[5] + T[6] + (T[9]-1) * T[4]`
d. `T[7]/(T[ 2*T[3] ])`
e. `(T[1] div T[5]) * (T[9]/T[6])`

3. Write the Pascal equivalent of each of the following statements.
   a. Define a record to consist of a client's last name and an amount owed by that client.
   b. Define an array of records, consisting of 100 elements, that stores the names and ages of all U.S. senators.
   c. Use the array created in Exercise 3b to determine the average age of a U.S. senator.
   d. Use the array created in Exercise 3b to determine if any senator is 31 years old. Do not use a **with** statement.
   e. Use the array created in Exercise 3b to determine if any senator is 31 years old. Use a **with** statement.
   f. Use the array created in Exercise 3b to count the number of senators who are 65 years old or older.
   g. Define an array of records containing medical information on a patient. Each record should consist of a patient's name, blood type, pulse rate, and blood pressure.
   h. Sum the contents of corresponding elements (elements with the same subscripts) in arrays A and B and store the result in an array C element with the same subscript.
4. Show the exact output for each of the following groups of Pascal statements.

a.
```
for k := 1 to 6 do
  begin
    a[k] := 2*k - 1;
    writeln(a[k])
  end;
```
b.
```
i := 1;
while i <= 6 do
  begin
    a[i] := 2*i - 1;
    writeln(a[i]);
    i := i + 1
  end;
```
c.
```
j := 1;
repeat
  a[j] := 2*j - 1;
  writeln(a[j]);
  j := j + 1
until j > 6;
```
d.
```
a.name := 'Thomas';
a.height := 72;
with a do
  writeln(name,' is ',height:2,' inches tall');
```
e.
```
a.name := 'Thomas';
a.height := 72;
with a do
  if height > 70 then
```

```
      writeln(name,' needs long size')
    else
      writeln(name,' needs regular size');
f. for i := 1 to 4 do
     a[i] := i;
   j := 1;
   while j < 5 do
     begin
       a[j] := a[j] * 3;
       j := j + 1
     end;
   k := 1;
   repeat
     writeln(a[k]);
     k := k + 1
   until k > 4;
```

PROGRAMMING EXERCISES

(G) 5. Write a program to fill and output a 10-element one-dimensional array with 0's.

(G) 6. Write a program to fill and output a five-element array of records with two fields: college name and number of students enrolled at that college.

(B) 7. Write a program to create an inventory record containing the name of an item, the number of items currently in stock, and the unit price of the item. Suppose the unit price must be increased by 7.5 percent. Update the record and display it.

(B) 8. Write a program to create an array of five inventory records. Use the information in the preceding exercise to output the updated records.

(B) 9. Write a program using the information in the preceding exercise, but create a fourth field containing the inventory value of each item. Output the entire array after adjusting for the 7.5 percent unit price increase, and show the value of the entire inventory.

(G) 10. Write a program that accepts a one-dimensional array called `beta` with 10 numeric elements as input and find the largest number in the array and the subscript identifying its location.

(G) 11. Write a program that calculates a table of values and draws a graph of the function $y = 2x - 5$. The values for the x coordinates should range from -20 to 20 in increments of 2.

(B) 12. Himhe Chocolates wants to find out how the public feels about a new candy bar it is testing. Write a program that asks each of 10 respondents if they would buy the candy bar. Store the 10 yes or no answers in an array. Determine and output the number of yes and number of no answers.

(B) 13. Hardware City is running a sweepstakes for its customers. Write a program that loads an array with the names of the five finalists in the contest. Have the computer choose a random number from 1 to 5, where the random number chosen represents the position of the winner in the given array. Write an appropriate letter to the winner, using that person's name as found in the array.

(B) 14. Firmware City is conducting a contest. The winner must guess the exact number of integrated circuit chips contained in a coffee cup in the store. If nobody guesses correctly, the prize goes to charity. Write a program using an array of records to store a contestant's name and guess. Input the winning number and output an appropriate message to the winner or the charity.

(G) 15. South Haven, Connecticut, is conducting its own survey on the number of children in each family. Write a program to store, in an array of records, the surname and the number of children in each of 10 families. Output a summary of the results, the total number of children in South Haven, and draw a bar graph depicting the number of no-children families, one-child families, and two-children families. Assume that no family in South Haven has more than two children.

(G) 16. Write a program to input a 20-element, one-dimensional array T, and calculate and output the sum of the elements stored in even-subscripted positions and the product of the elements stored in odd-subscripted positions.

(B) 17. Gumbel's Department Store maintains inventory control by storing sales and inventory information in an array of records. The following is an example:

ITEM NO.	*NO. SOLD*	*COST PRICE*	*SALE PRICE*	*PROFIT*
458	0	$17.50	$23.99	0
217	0	9.95	14.95	0
874	0	11.75	13.95	0
121	0	21.00	29.99	0
553	0	15.00	21.75	0
348	0	13.50	17.95	0

Write a program that creates an array of records to store the information shown and that updates the array by inputting the number of items sold in Field 2 and computing the profit in Field 5. Output the entire filled array in tabular form with appropriate headings.

(G) 18. Write a program that generates 40 random integers between 1 and 100 and stores these integers in two 20-element arrays X and Y. Create a new array, Z, by merging the arrays X and Y.

(G) 19. Write a program to assist Professor Ima Sue Smart in processing her grades. Assume that her class has 10 students and each has taken three exams. Enter the following data in an array of records.

NAME	*EXAM 1*	*EXAM 2*	*EXAM 3*
Jones	75	83	78
Wallace	91	88	75
Gomez	82	88	80
Smith	90	72	78
Caruso	77	84	68
Black	81	65	87
Kojak	68	93	85
Chen	73	82	70
Goldberg	86	71	73
O'Grady	62	76	85

Calculate and print the class average score for each exam. The program should also be capable of searching the name array for an input name and calculating the corresponding average of the three exam scores for that student.

(G) 20. Using the data given in the preceding exercise, create a new field containing the average exam score for each student. Use a sorting routine to print the list in descending order according to the average score for the three exams for a student; that is, the record with the highest average should appear first.

(G) 21. Statistics for a baseball team are listed in the following array form:

PLAYER	*AT BATS*	*NO. HITS*	*NO. HOMERS*	*AVERAGE*
Mitts	258	76	10	
Wall	328	108	8	
Cage	311	97	5	
Whiffo	291	101	9	
Mound	287	88	12	
Batts	307	95	3	
Aoute	318	107	7	
Runner	270	83	0	
Bunter	321	103	10	

Write a program that loads an array of records with the preceding information, computes the batting average for each player by dividing the number of hits by the number of at bats, and stores the results in the fifth column of the array. Averages should be rounded to three decimal places before being stored. Output the entire array appropriately labeled.

(G) 22. Use the information in the preceding exercise and, after the averages have been calculated, output the table with the data arranged in descending order according to batting average. Output the array a second time with the data arranged in descending order of the number of homers.

(B) 23. Write an accounts payable program for Alpha Ltd. The company has its ledgers set up by vendor and by product. The company uses three vendors and the same two products (mortars and pestles) for each vendor. Use an array of records with three fields (vendor name, amount of money paid to the vendor for mortars, and amount of money paid to the vendor for pestles) to store the data. Update the data in the array by entering the vendor name, a product (mortar or pestle), and the amount paid to that vendor for that product. Add the amount paid to the amount currently stored in the array. After updating the entire array, output the amount owed to each of the vendors with appropriate headings.

(B) 24. Dana's Dating Service provides a questionnaire for its applicants. The questionnaire is composed of four yes-no questions. Responses are stored in the form of *Y* for yes and *N* for no. Create an array of records for male respondents with the first field representing the applicant's name and the next four fields that applicant's responses, in order. Fill a six-record array with sample data. Assume that a female applicant desires to be matched for a date. Enter the name of the female applicant and her responses to the four questions and determine a suitable dating partner (the one or ones with the most matches).

PART THREE

ADVANCED PROGRAMMING CONCEPTS

CHAPTER 12

Modular Programming

12.1 INTRODUCTION

In programming applications, complex solutions are the rule rather than the exception. Whether the problems relate to education, science, business, or other fields, program solutions often require hundreds of lines of code. The responsibility for the writing, testing, and debugging of such lengthy programs can become an overwhelming task for a programmer.

An efficient approach is to divide the entire problem into smaller units and assign each to a different programmer in what is known as a *programming team.* Each member of the team can then attempt to solve a portion of the problem, write the appropriate code, test the program, and debug it. At the conclusion of this process, each member of the team has a working program to solve a particular aspect of the problem. Each of these separate programs, or modules, can then be united under the single logic design of the original program.

As you have seen in previous chapters, programs are composed of functions and procedures that are tied together by a main program. Pascal is particularly suited to a modular approach for the design of solutions of complex problems.

For example, assume that Popstop Distributors wishes to computerize its end-of-the-month sales reports. The company distributes two regular sodas and two diet sodas. Each salesperson is required to write a monthly sales report listing the number of cases of each soda that was sold in the given month and compare that to the number of cases of the same soda sold for the corresponding month of the previous year. In addition, the salesperson is to obtain the difference in the number of cases sold monthly in both years and express the same as a percentage increase or a percentage decrease. Moreover, a separate accounting for regular and diet sodas, listing the total number of cases sold for each soda and their comparisons, is required.

This problem can be divided into four separate subprograms. These include the entry of monthly data and the display of a monthly summary for all sodas,

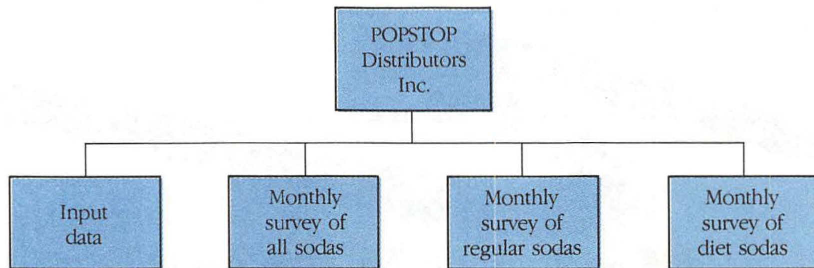


FIGURE 12.1

regular sodas, and diet sodas. The use of the modular design approach to the solution of such a problem helps the programmer focus on the four individual parts. Figure 12.1 shows a diagram for the solution of the problem using the modular approach.

The four parts are shown in separate boxes, all connected to the box representing the entire job.

The benefit of modularization increases with the complexity of the problem to be solved. This chapter introduces a modularization technique that can be used in the design of program solutions to practical problems. The intention is to provide you with an understanding of, and an appreciation for, this approach to problem solving.

12.2 THE THEORY OF MODULAR PROGRAM DESIGN

A lot has been written about programming style and the enhancement of the readability of programs. If a program is easy to read, it can be debugged more readily. Further, it can help to serve as a model for a programmer who is constructing a program solution to a similar problem. Finally, a program that can be easily understood can be modified when the various parameters of the initial problem change. Modifications to programs are often made after a significant amount of time has elapsed, and it is not unusual for the modifications to be made by a programmer who did not write the original program. Clarity in design and implementation makes the job of the modifying programmer much simpler. The result is a more efficient operation and a minimization of the management costs involved in system maintenance.

Although you've been introduced to many of the principles of good programming practice in previous chapters, you should now begin to understand and appreciate how these techniques fit together to form a unified approach for the solution of problems. We have used top-down design, structured programming, and documentation in the text and are now ready to present a formal discussion of these principles.

TOP-DOWN DESIGN

The top-down design technique can be used at any programming level. This process helps you move from the initial statement of the problem to the identification of the specific tasks required to solve it. Individual tasks are broken down into separate subtasks and these subtasks individually programmed and linked together to form a solution to the original problem.

In top-down design, the initial problem is broken down into logical, easy-to-understand steps, and each of these steps can be redefined into simpler ones, producing a series of elementary steps. Each stage of the decomposition shows how these steps are interrelated. The final refinement leads to the production of program code.

The IPO chart, which was introduced earlier in the text, is an example of an aid to top-down design in the solution of a problem. In order to construct an IPO chart, you identify the data given in the problem statement, determine what results are called for, and decide how to form the bridge that leads from the problem statement to the solution. This progression is done for each subprogram in the main program.

The N-S chart also clarifies the structure of the solution to a complex problem. The N-S chart for the main program shows the relationship among all the subprograms. Each subprogram has its own N-S chart. The way in which the subprograms are connected and how they are related is shown in a structure chart similar to the one shown in Figure 12.1.

For example, consider the job of preparing a payroll for a small business. The first step in the analysis of the task might lead to the following division: input of data, calculations of the pay amounts, and output of the results. The initial decomposition of the problem might take the form shown in Figure 12.2.

A refinement of each of these primary steps might begin with a determination of the sources of data needed to solve the problem. If the payroll is prepared on a weekly basis, it is likely that the number of hours worked in that week by an employee is entered from a summary of the daily figures accumulated on time cards, or perhaps from the time cards themselves. On the other hand, individual employee data of the year-to-date total for earnings, deductions, and

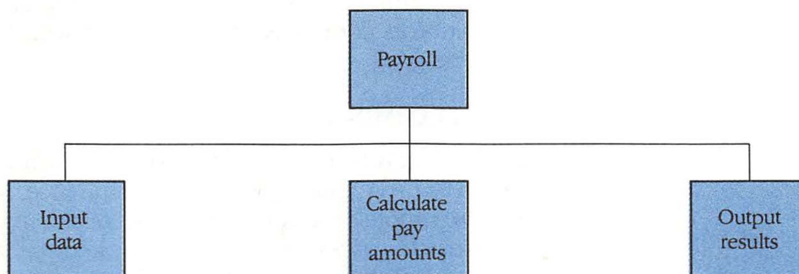


FIGURE 12.2

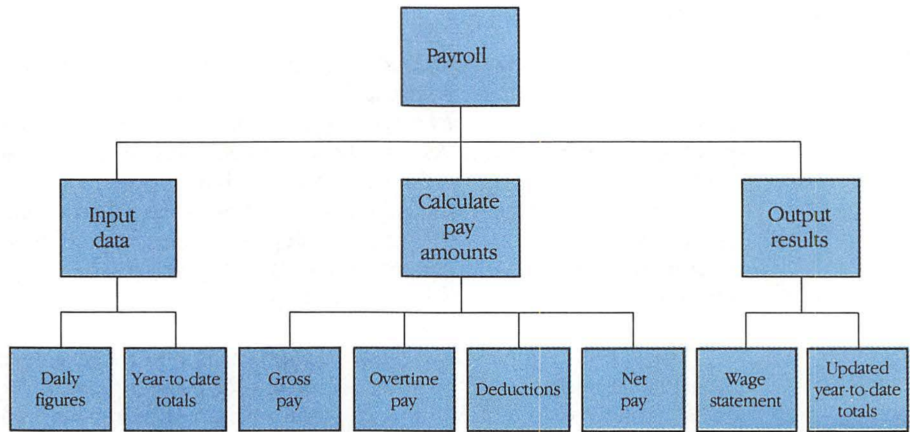


FIGURE 12.3

net payments are likely to be stored in a file. Hence, the input portion of the solution should be able to accommodate input from both a file and the keyboard.

A further refinement of the calculation step might indicate the steps required to find the individual quantities for the pay amounts. These quantities could include gross wages for the pay period; overtime pay, if any; amounts of specific deductions for taxes, FICA, pension payments, and health insurance payments; and net pay.

The output portion of the initial analysis might be decomposed into modules that produce a wage statement listing gross figures, deductions, and net figures; the printing of payroll checks; and an update of the year-to-date figures. The modules could be used for preparing a payroll report and for updating a file so that it is ready for the processing of similar data during the next pay period.

A second-level decomposition in this approach appears in Figure 12.3.

The objective of the top-down design strategy is the decomposition of a problem into small components, each of which can be programmed readily, and the integration of those individual components into a single logical framework. Once the decomposition is completed, you are faced with the task of writing code to accomplish the goal.

STRUCTURED PROGRAMMING

Once the solution has been broken up into modules, each module should be coded using three basic programming structures: sequence, selection, and repetition. No matter how simple or how complicated a problem is, a well-designed solution can be constructed as a combination of these three structures.

A collection of statements that are executed in order, from beginning to end, is called a *sequence structure.*

Chapter 9, "Choices," deals with *selection structures* in which decisions are made. The result of a decision directs program control to one of a number of branches.

Chapter 10, "Loops," treats those processes in which a repetition of program instructions is required to solve a problem. The statements comprising a loop form a *repetition structure*.

It is common for a program not only to employ all three structures but also to nest some structures within others. The N-S charts are designed to graphically represent problem solution employing the three structures. By taking this perspective in writing programs, you can learn to develop analytical skills that ultimately help in solving complex problems.

DOCUMENTATION

One technique that can enhance program readability is the use of remarks or comments throughout the program. Any device that adds an explanation or otherwise reinforces the understanding of a program is referred to as *documentation*. Documentation within a program helps a programmer understand why certain processing steps occur at specific points. If the program is a structured, well-designed program, this type of documentation provides additional clarity.

You have already observed that MacPascal permits comments to be interspersed throughout the program by the use of braces. A complete series of comments may be used to give a full explanation of what is happening in the program or to summarize an IPO chart in the program. When writing a program, you may find it useful to assume that another programmer may be reading the program and modifying it at some time in the future. Explanations and directions that might otherwise be communicated to the second person via conversation should be organized into a series of brief comments included in the program.

The techniques just described are ways of including documentation within a program listing. Output demands the same attention to clarity and completeness. Output text and data should be organized and presented in a format that is not only readable but that describes how that output is to be interpreted. Values should be identified, tabular data should include row and/or column captions, and text output should be displayed in paragraph form. The ordinary standards of good writing style should prevail.

12.3 USER-FRIENDLY PROGRAMS

As computers assume an ever-increasing role in modern society, the number of inexperienced users continues to grow. These users interact with programs and software packages of various types. In most cases, these individuals are totally unfamiliar with the logic of a program itself. This type of user is only interested in applying a particular program to his or her specific problem. To respond to this growing audience of novice users, a programming approach termed *"user-friendly"* has been developed.

User-friendly programs are designed to reduce the anxiety and possible confusion that can arise when the end-user is not a programmer. To accomplish this, a user-friendly program sets up a dialog that provides the user with the

opportunity to make responses and elect choices that in turn trigger certain actions in the program. Accordingly, the program is designed to ask simple questions of the user, to which the user can respond in a conversational manner.

CREATION OF A PROGRAM MENU

A program menu displays the various tasks that the program can perform. The very term *menu* suggests a listing from which selections may be made. The solution of a complex problem by modularization leads quite naturally to the creation of a program menu. The menu offers a user the opportunity to select the specific portion of the large program that is desired at a particular time.

Consider the example of Popstop Distributors, introduced at the beginning of this chapter. A reorganized structure depicting the tasks to be performed in that problem is shown in Figure 12.4.

The menu shown in Figure 12.5 lists the options that are available in the program.

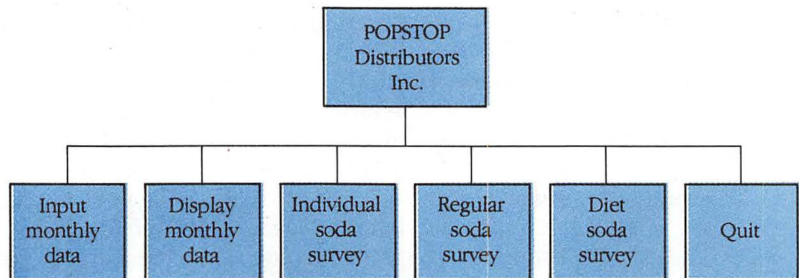


FIGURE 12.4

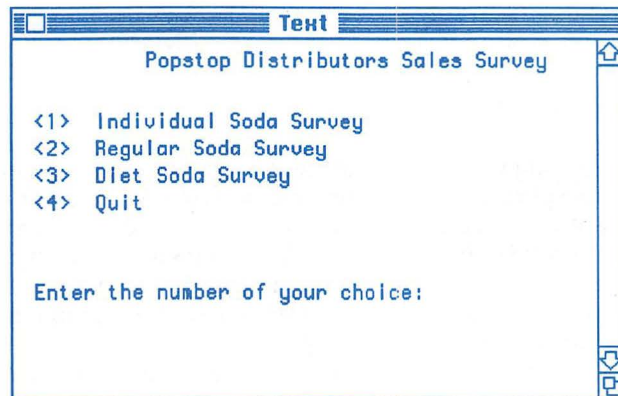


FIGURE 12.5

The menu offers four options listed with corresponding number choices. Choice 4, the `Quit` option, provides the user with a way to terminate the program. Although the user may have no knowledge of the actual program instructions, he or she can nevertheless use the program by making appropriate selections from this menu.

SUBPROGRAM DIALOGUE

Each selection of a menu choice activates a subprogram. Since a subprogram may only be executed through the specific selection of a menu choice by the user, the main program must include a means for selecting the execution of a choice. In other words, there must be a dialogue between the main program and the subprogram. When a particular option is selected by the user, the module containing the program code must be summoned, or called into action by the main program. Once the execution of the particular subprogram is complete, control in the program returns to the main program at the point from which the call was made to permit other menu selections to be chosen. The procedures and functions described in previous chapters are ideal for this technique. Subprograms are always complete and self-contained; distinguishable as separately listed statements from the main program; subject to a summon or call from the main program; and capable of returning control to the main program at the point from which the call was made.

12.4 PRACTICAL APPLICATIONS

The techniques described in the previous sections are now applied to specific problems so that you can see how these techniques fit together in the solution of the problem. We are returning to the use of IPO and N-S charts to give you an idea of what a complete solution design to a program should look like.

Problem: The Reliable Automobile Insurance Company conducts a random sampling of 10 customers to determine the average number of accidents per customer during the past year. Write a program containing modules that (1) find the average number of accidents per driver per year, (2) find the median number of accidents per driver per year, (3) find the mode number of accidents per driver per year, and (4) draw a bar graph depicting the number of drivers (*y*-axis) for each category for number of accidents. The categories are zero accidents per year, one accident per year, two accidents per year, three accidents per year, and four or more accidents per year.

The structure chart showing the modules and their relationships is depicted in Figure 12.6.

On the first level of the structure chart are six subprograms represented by the boxes labeled `Entry`, `Average`, `Median`, `Mode`, `BarGraph`, and `Quit`. The first five of these subprograms are further divided on the second level into one or more subprograms. One of the subprograms on this level,

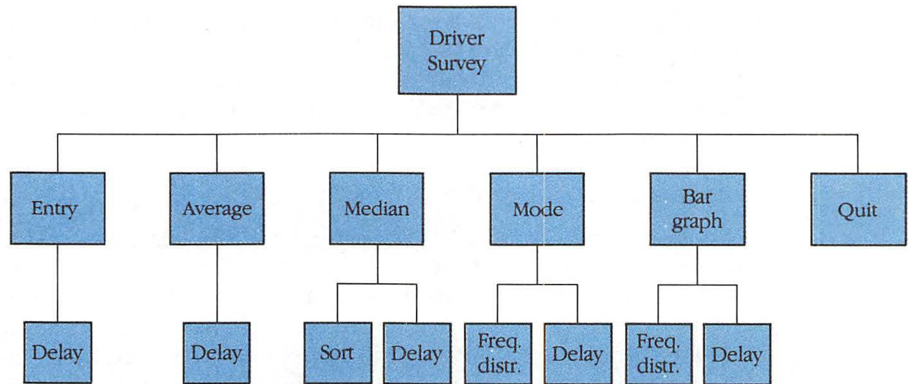


FIGURE 12.6

`Delay`, is called by each of the subprograms, and the other subprogram, `FrequencyDistribution`, is called by only two of them.

At this point you may ask, "With so many subprograms to write, which one do I write first?" A good starting point is to begin with the main program in order to establish the global identifiers and organize the calls to the subprograms according to the structure chart. Then each subprogram can be completed and added to the program in the same order as the main program procedure calls.

The IPO chart for the main program and the accompanying N-S chart are shown in Figure 12.7.

The main program and the global declaration section are shown in Figure 12.8.

To conserve storage space, the identifiers `auto` and `freq` have been declared in the main program and used in the subprograms. This eliminates the need to pass any parameters to the subprograms. If limited memory is not a concern, local identifiers for these values can be declared in each module that needs them. This adds flexibility to the program, since modules become independent of the identifiers declared in the main program.

The identifier `auto` is of type `motor` and therefore defined to be a 10-element array of integers. It is used to store the numbers of accidents for the 10 drivers in the sample. The identifier `freq` is of type `count`, which is defined to be a five-element array of integers.

The initial statements in the main program direct a user to organize the screen to obtain a good view of text and graphic output. Although the program continues regardless of whether these directions were followed, it is helpful to prepare the screen for proper program display.

In programs that have many parts, it is often necessary for the program to pause between segments to permit a user to see results or prepare for data entry. The `Delay` module, shown in Figure 12.9, serves this purpose.

This module merely halts the program until a user is ready to continue. After pressing any key (although a Return is requested), the program returns to the main program, where the **Text** window is cleared and the menu is displayed.

IPO CHART

Main Program Function Procedure (circle one)

Name DriverSurvey

Class	Identifier	Description
INPUT	auto	array of integers representing number of accidents per driver per year
	choice	option number selection
PROCESSING	freq	array of integer counts for number of accidents per driver per year

OUTPUT	Identifiers		Drawing
	choice		

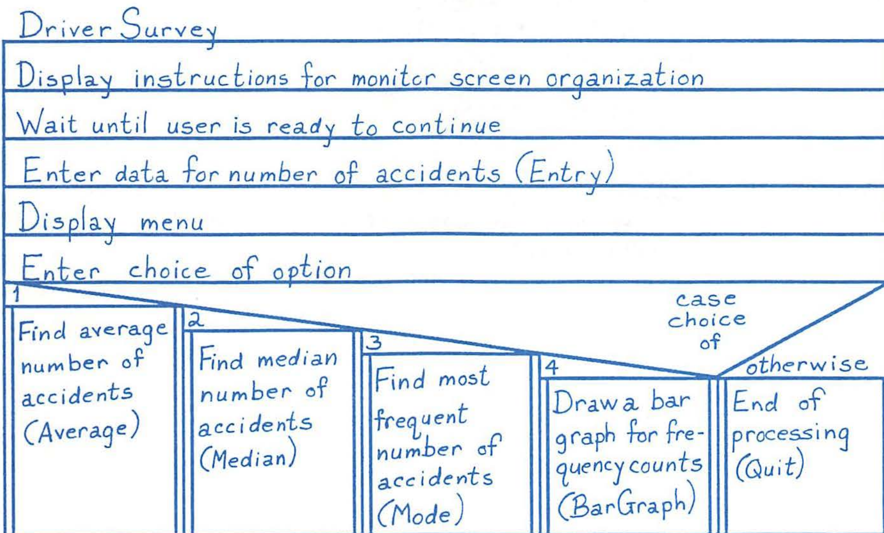


FIGURE 12.7

```
program DriverSurvey;
{Determine the average, median, mode}
{of data entered on accidents per driver}
{and displays data as a bar graph}
 type
  motor = array[1..10] of integer;
  count = array[0..4] of integer;
 var
  choice   {menu selection}
   : integer;
  auto      {array of the number of accidents for a motorist}
   : motor;
  freq      {array of count of number of drivers with 0 to 4 accidents}
   : count;
begin {*main program*}
 writeln('Put the Text window on');
 writeln('left half of the screen.');
 writeln;
 writeln('Put the Drawing window on');
 writeln('the right half of the screen.');
 writeln;
 Delay;
 Entry;
 repeat
  page;
  writeln('          Driver Survey');
  writeln('        Sample Size : 10');
  writeln;
  writeln('Number of accidents per driver');
  writeln;
  writeln('<1>  Average');
  writeln('<2>  Median ');
  writeln('<3>  Mode');
  writeln('<4>  Bar Graph');
  writeln('<5>  Quit');
  writeln;
  writeln;
  repeat
   write('Enter the number of your choice: ');
   readln(choice)
  until (choice <= 5) and (choice >= 1);
  case choice of
   1 :
    Average;
   2 :
    Median;
   3 :
    Mode;
   4 :
    BarGraph;
   otherwise
    Quit
  end
 until choice = 5
end.
```

FIGURE 12.8

IPO CHART

Main Program Function (Procedure) (circle one)

Name Delay

Class	Identifier	Description
INPUT	ans	any keystroke
PROCESSING		

	Identifiers		Drawing
OUTPUT	ans		

Delay

Request keystroke to continue
Enter keystroke

```
procedure Delay;
 {Permits user to view screen display and continue at own option}
 var
  ans    {continuation response}
   : string;
begin
 writeln;
 write('Press Return to continue.');
 readln(ans)
end;
```

FIGURE 12.9

The purpose of the main program is to produce a menu from which a user selects an option on the menu list. The **Text** window is cleared before the menu is displayed. The last option on a menu is the `Quit` option, which is the only "legitimate" way to end program execution. Each menu option is preceded by a number, and a user is requested to enter one of these numbers. An error check on that number is performed immediately, using a **repeat...until** structure, and the program does not progress to the proper subprogram unless a valid value is entered.

Once a valid option number is obtained, a **case** structure, using that number, directs program execution to the subprogram performing the task selected. The statements that produce the menu, test the option number entered, and select the next program activity are nested in a large **repeat...until** structure. This structure permits a user to continue to make choices from the same menu and view different activities until the `Quit` option is selected.

The data for the 10 drivers must be in memory before any statistical analysis can be performed. The `Entry` IPO and N-S charts, code, and run are shown in Figure 12.10. This module is called before any selection from the menu is offered. The request for data entry appears in the **Text** window. The `Entry` module clears the **Text** window, requests data, and calls the `Delay` module so that a user is able to view the data entered. The `Delay` module returns to the `Entry` module when the user enters a keystroke, and the `Entry` module then returns to the main program.

The first option in the menu is `Average`. This subprogram sums the data in the `auto` array and divides the sum by 10 to produce the mean number of accidents per driver. The module begins with the **page** statement to clear the **Text** window and ends with the `Delay` module before returning to the main program. The main program then displays the menu again and waits for the user's choice. The `Average` IPO and N-S charts, subprogram, and run are displayed in Figure 12.11.

The second option in the menu is `Median`. This module finds the value in the middle of a sorted list. A median is known as another measure of central tendency for a sample of numeric data and may be more representative of the "average" of the sample than is the mean, which may be distorted by extreme values. Figure 12.12 displays the `Median` and `Sort` IPO charts, N-S charts, modules, and run.

The sample data must be in ascending or descending order before the median of the sample can be selected. Therefore the `Median` module calls the `Sort` module, which organizes the data in the `auto` array in ascending order. The sort routine is similar to the one presented in Chapter 11.

The `Median` subprogram clears the **Text** window, calls `Sort`, determines and displays the median of the set of 10 values before calling `Delay` and returning to the main program. Since the set of data is known to have an even number of values, the arithmetic mean of the two middle values (fifth and sixth) is the median value for the set. If the set were to contain an odd number of values, the median value is the value exactly in the middle of the ordered set. For example, if a set contains 11 values the sixth value would be the median.

IPO CHART

Main Program Function (Procedure) (circle one)

Name Entry

Class	Identifier	Description
INPUT	auto	array for number of accidents
PROCESSING	i	loop index

OUTPUT	Identifiers		Drawing
	auto		

Entry

Clear Screen
for i = 1 to 10
Request number of accidents
Store in array
Wait until user wishes to continue (Delay)

FIGURE 12.10 *(continued)*

```
procedure Entry;
 {Enters data into the array for processing}
 var
  i :                    {loop index}
  integer;
begin
 page;
 writeln('Enter number of accidents for 10 drivers');
 writeln;
 for i := 1 to 10 do
  begin
   write('# ' : 2, i : 3, ' Number of accidents : ');
   readln(auto[i])
  end;
 Delay
end;
```

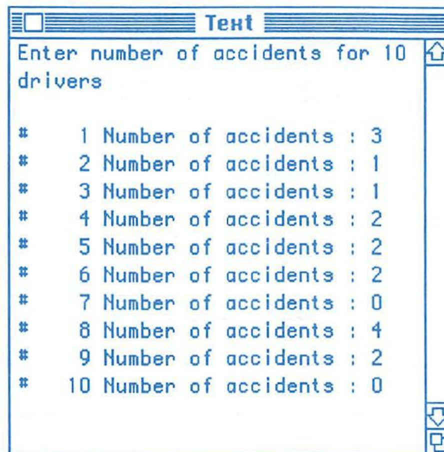


FIGURE 12.10

IPO CHART

Main Program | Function | (Procedure) (circle one)

Name Average

Class	Identifier	Description
INPUT	auto	array of number of accidents per driver
PROCESSING	mean	arithmetic average number of accidents per driver in survey
	sum	sum of all accidents
	i	loop index

OUTPUT	Identifiers		Drawing
	mean		

Average

Clear screen
Initialize sum to zero
for i = 1 to 10
Add array element to sum
mean = sum/10
Display mean number of accidents
Wait until user wishes to continue (Delay)

FIGURE 12.11 *(continued)*

```
procedure Average;
 {Finds the average number of accidents per driver}
 var
  sum,           {total of accidents}
  i              {loop index}
   : integer;
  mean        {average number of accidents}
   : real;
begin
 page;
 writeln('Average');
 writeln;
 sum := 0;
 for i := 1 to 10 do
  sum := sum + auto[i];
 mean := sum / 10;
 writeln('The average number of accidents');
 writeln('per driver per year');
 writeln('====>  ', mean : 5 : 2, '   <====');
 writeln;
 Delay
end;
```

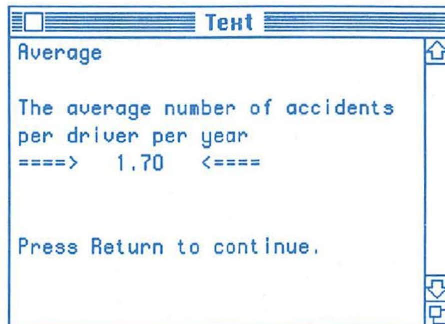


FIGURE 12.11

IPO CHART

Main Program Function (Procedure) (circle one)

Name Sort

Class	Identifier	Description
INPUT	auto	array of number of accidents
PROCESSING	i	loop index
	flag	determined by interchange during a loop pass
	temp	temporary storage location used during element interchange

OUTPUT	Identifiers	Drawing
	auto	

Sort

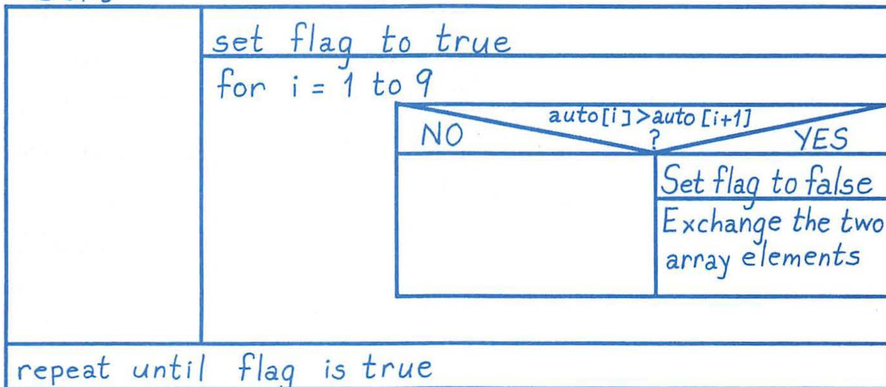


FIGURE 12.12 *(continued)*

IPO CHART

Main Program Function (Procedure) (circle one)

Name Median

Class	Identifier	Description
INPUT	auto	sorted array of number of accidents
PROCESSING	i	loop index
	middle	value in "middle" of sorted list

Class	Identifiers	Drawing
OUTPUT	middle	

Median

Median
Clears screen
Organizes data in array in ascending order (Sort)
Middle value = average of 2 middle values
Display middle value
Wait until user wishes to continue (Delay)

FIGURE 12.12 *(continued)*

```
procedure Sort;
 {Arranges the records in ascending order }
 {according to the number of accidents}
 var
  i       {loop index}
   : integer;
  flag     {indicates whether switch has been made}
   : boolean;
  temp     {temporary location for switching contents}
   : integer;
begin
 repeat
  flag := true;
  for i := 1 to 9 do
   if auto[i] > auto[i + 1] then
    begin
     flag := false;
     temp := auto[i];
     auto[i] := auto[i + 1];

     auto[i + 1] := temp
    end
 until (flag)
end;

procedure Median;
 {Determines the value in the middle of }
 {a sorted list of number of accidents}
 var
  middle     {median number of accidents per driver per year}
   : real;
begin
 page;
 writeln('Median');
 writeln;
 Sort;
 middle := (auto[5] + auto[6]) / 2;
 writeln('The median number of accidents');
 writeln('per driver per year');
 writeln('====>  ', middle : 5 : 2, '   <====');
 writeln;
 Delay
end;
```

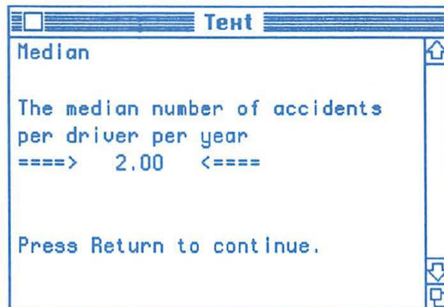


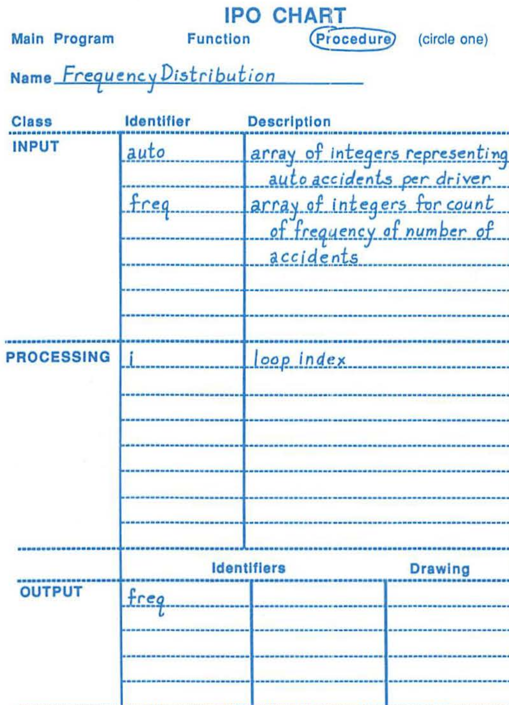
FIGURE 12.12

The third option in the menu is `Mode`. The mode of a set of values is the value that appears most frequently in the set. A set of numbers may have no mode (for example, 1, 5, 4, 8, 0, 98) or may have more than one mode (for example, 1, 5, 1, 8, 8, 98, where both 1 and 8 are the modes).

In order to determine the most frequent number of accidents in the data set, `FrequencyDistribution` is called to count the number of occurrences of each value in `auto`. This frequency count is defined in the main program as the array `freq` with subscripts ranging from 0 to 4. The subscripts represent the numbers of accidents, 0, 1, 2, and 3; 4 represents all the values 4 and larger. For example, if a value in the `auto` array is 2, the value of `freq[2]` is increased by one; if another value is 8, the value of `freq[4]` is increased by one.

Once the frequency array is completely determined, the `Mode` module selects the largest value in the array and finds all the array elements that match the largest value. The subscripts of those array elements then represent the number of accidents—the mode(s).

The IPO and N-S charts, listings, and run for these modules are shown in Figure 12.13.


IPO CHART

Main Program | Function | (Procedure) | (circle one)

Name FrequencyDistribution

Class	Identifier	Description
INPUT	auto	array of integers representing auto accidents per driver
	freq	array of integers for count of frequency of number of accidents
PROCESSING	i	loop index

	Identifiers	Drawing
OUTPUT	freq	

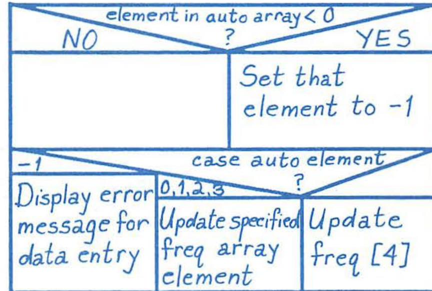
FIGURE 12.13 *(continued)*

Frequency Distribution

for i = 0 to 4

Set freq array elements to zero

for i = 1 to 10



IPO CHART

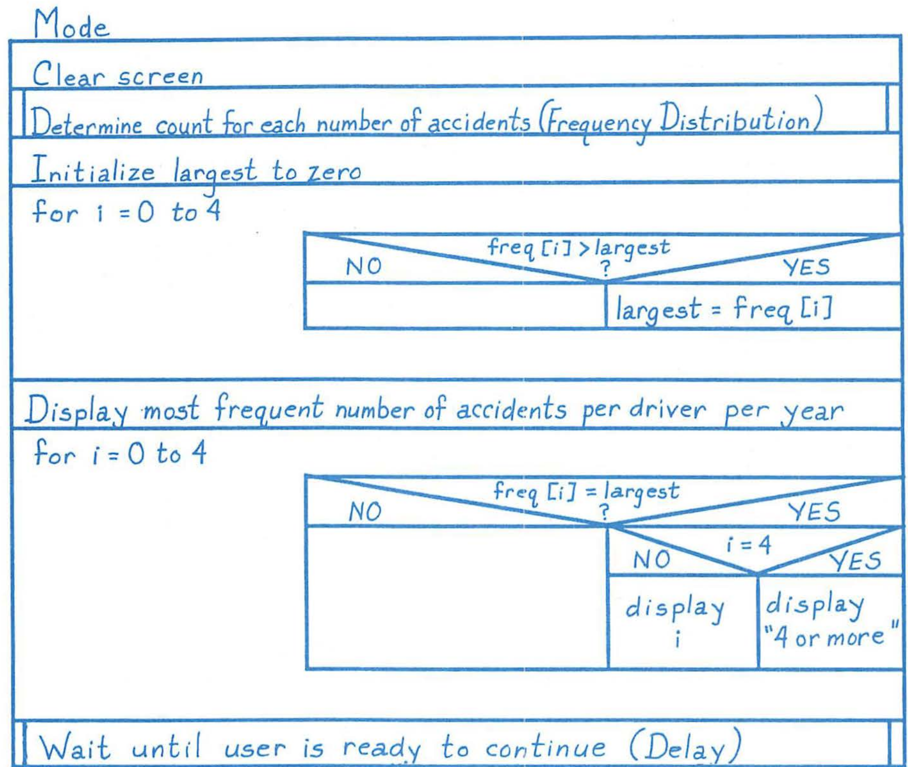
Main Program **Function** **(Procedure)** (circle one)

Name Mode

Class	Identifier	Description
INPUT	freq	frequency array
PROCESSING	i	loop index
	largest	largest value in freq array

	Identifiers	Drawing
OUTPUT	i	
	largest	

FIGURE 12.13 *(continued)*




```
procedure FrequencyDistribution;
 {Finds the frequencies of number of accidents per driver}
 var
  i      {loop index}
   : integer;
begin
 for i := 0 to 4 do
  freq[i] := 0;
 for i := 1 to 10 do
  begin
   if auto[i] < 0 then
    auto[i] := -1;
   case auto[i] of
    -1 :
     writeln('Data entry error : negative number of accidents in ', i : 2, ' th record.');
    0, 1, 2, 3 :
     freq[auto[i]] := freq[auto[i]] + 1;
    otherwise
     freq[4] := freq[4] + 1
   end
  end
end;
```

FIGURE 12.13 *(continued)*

```
procedure Mode;
{Finds the largest frequency}
  var
    i,           {loop index}
    largest     {most frequent number of accidents}
     : integer;
begin
  page;
  writeln('Mode');
  writeln;
  FrequencyDistribution;
  largest := 0;
  for i := 0 to 4 do
    if freq[i] > largest then
      largest := freq[i];
  writeln('Most frequent number of accidents');
  writeln('per driver per year');
  for i := 0 to 4 do
  {if mode is not unique}
    if freq[i] = largest then
      if i = 4 then
        writeln('====>  4 or more  <====')
      else
        writeln('====>  ', i : 1, '  <====  ');
  writeln;
  Delay
end;
```

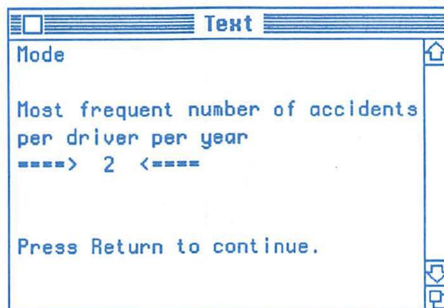


FIGURE 12.13

`FrequencyDistribution` is a separate subprogram, because it is also called in the `BarGraph` module. The module initializes the `freq` array to zeros, checks for erroneous data (negative values), and determines the counts for numbers of accidents.

Another option offered in the menu is `BarGraph`. The IPO and N-S charts, listing and run are shown in Figure 12.14. This subprogram clears the **Text** window and calls `FrequencyDistribution`. The values determined by that module are used to draw a bar graph in the **Drawing** window. The programming that constructs this graph is similar to that used in Figure 7.23.

The final option in the menu is the `Quit` option. The IPO and N-S charts, listing, and run are shown in Figure 12.15.

IPO CHART

Main Program Function (Procedure) (circle one)

Name: Bar Graph

Class	Identifier	Description
INPUT		
PROCESSING	i	loop index
	j	loop index
	y	
	v	horizontal coordinate
	x	horizontal coordinate
	top	top of rectangle drawn for bar graph

	Identifiers		Drawing
OUTPUT			Bargraph

FIGURE 12.14 *(continued)*

BarGraph

Clear Screen
Determine frequency counts (FrequencyDistribution)
Draw vertical and horizontal scales and graph titles
for j = 1 to 10
Find actual vertical coordinate
Find vertical drawing coordinate
Draw line
Reposition pen
Display j
for i = 0 to 4
Find horizontal drawing coordinate
Reposition pen
Display i
Display title
for i = 0 to 4
Find horizontal drawing coordinate
Find top boundary of bar
Paint a rectangle
Wait until user is ready to continue (Delay)

FIGURE 12.14 *(continued)*

```
procedure BarGraph;
 var
  i, j,            {loop indices}
  y,            {actual y coordinate}
  v,            {drawing y coordinate}
  x,            {drawing x coordinate}
  top      {top boundary of bar}
   : integer;
begin
 FrequencyDistribution;
 page;
 writeln('Bar Graph');
 writeln;
 writeln;
 writeln('=============>');
 moveto(10, 30);
 writedraw('frequency ');
 drawline(40, 35, 40, 200);
 drawline(30, 190, 150, 190);
 for j := 1 to 10 do
  begin
   y := 15 * j;
   v := 190 - y;
   drawline(38, v, 42, v);
   moveto(20, v + 5);
   writedraw(j : 2)
  end;
 for i := 0 to 4 do
  begin
   x := 55 + 20 * i;
   moveto(x, 210);
   writedraw(i : 1)
  end;
 writedraw('+');
 moveto(40, 225);
 writedraw('number of accidents');
 for i := 0 to 4 do
  begin
   x := 55 + 20 * i;
   top := 190 - freq[i] * 15;
   paintrect(top, x, 190, x + 10)
  end;
 writeln;
 Delay
end;
```

FIGURE 12.14

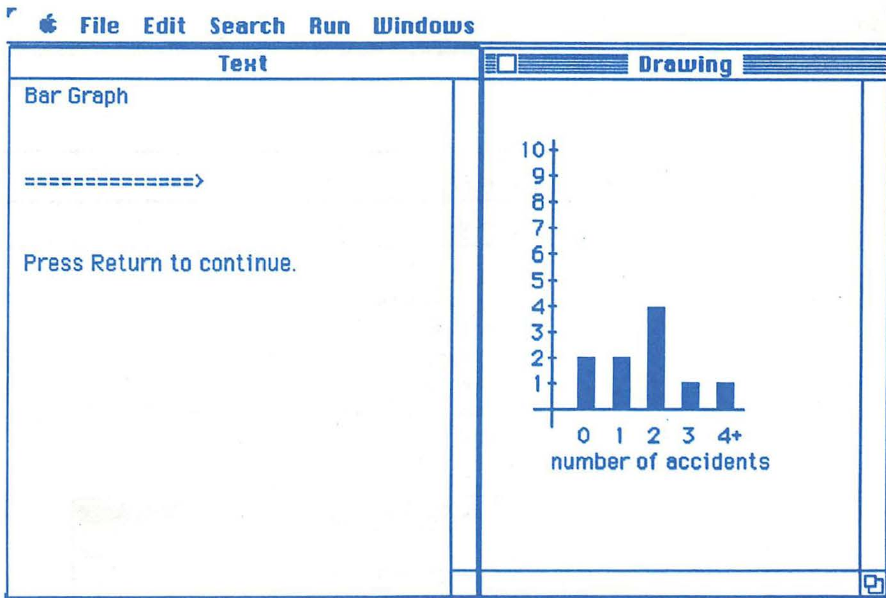


FIGURE 12.14 *(continued)*

IPO CHART

Main Program Function (Procedure) (circle one)

Name *Quit*

Class	Identifier	Description
INPUT		
PROCESSING		

	Identifiers		Drawing
OUTPUT	*message*		

FIGURE 12.15 *(continued)*

Quit
Clear screen
Display message

```
procedure Quit;
begin
 page;
 writeln;
 writeln(' Processing complete.')
end;
```

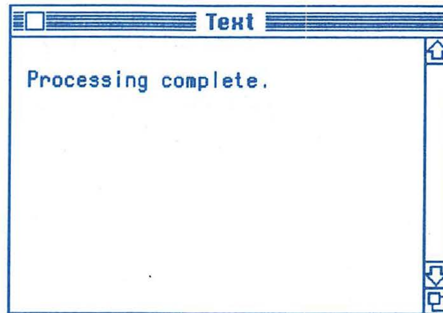


FIGURE 12.15

Problem: President Fizz of Popstop Distributors wants a monthly sales survey showing the number of cases of each of four different types of sodas (two regular and two diet) sold last year, cases sold this year, the difference between the months, and the percent difference. In addition, she wants separate monthly sales surveys done for the regular sodas and for the diet sodas.

The first level of the structure chart presented in Figure 12.16 includes a module for data entry. Although this step is not specified in the problem, it is necessary to have data in memory before work can be done on it.

`Display` is not specified in the problem statement. When a large amount of data is to be entered, it is good programming practice to display the data to make sure that they are entered correctly. It can also give a user a chance to make a quick visual check on the accuracy of the answers produced by the program. In some applications, data are stored in an external file and entered automatically as the program begins.

The three sales survey modules, on the first level, each call `Delay` so that a user can view output before continuing with the program. The `Quit` module is the final module listed.

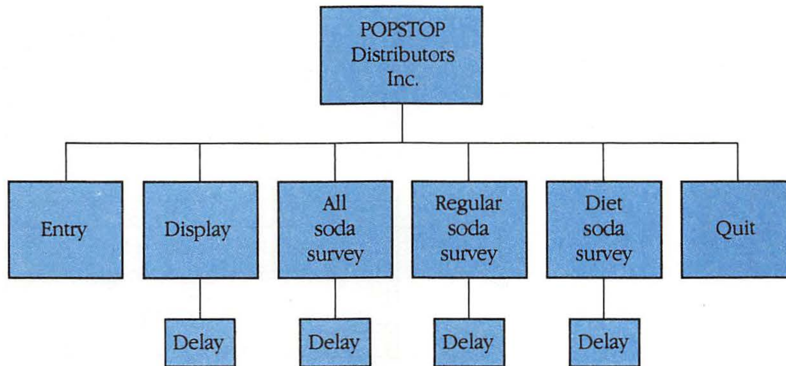


FIGURE 12.16

The IPO and N-S charts for the main program appear in Figure 12.17.

The main program listing and declaration section are shown in Figure 12.18.

The data for this problem are organized into a four-record array. Each of the four records, defined as type `soda`, represents a different soda sold by the company. The first field of the record holds the name of the soda. The second field consists of the monthly sales organized as a 12-element array in which each element represents the number of cases sold in one month. This example shows the use of an array as a field in a record. Since `drinks` is defined to be an array of records and each record contains an array, we have, in effect, an array of arrays.

Data for last year are organized in the four-record array of type `drinks` and defined as `year85`, and data for the present year are in another four-record array of type `drinks` identified as `year86`. The `year85` array is completely filled with data, since the year is finished. However, the elements in the monthly sales array for the records in `year86` are filled in as the present year progresses and remain blank until each month is complete.

The main program begins with a call to `Entry`, shown in Figure 12.19, followed by `Display`, shown in Figure 12.20.

The `Entry` and `Display` modules are called by the main program before a menu is presented to the user.

As in the preceding problem, the menu is nested in a **repeat...until** structure. The menu offers the user three different ways of looking at the same data. Each menu selection is tied to an option in the case structure that follows. The first three options reference the same subprogram, `SodaSurvey`. Each option sends a different set of parameters so that the same subprogram can produce a soda survey for all the sodas, for only the regular sodas, or for only the diet sodas. Figure 12.21 displays the `SodaSurvey` module and three runs produced by the selection of Options 1, 2, and 3 from the menu.

The parameters sent to the `SodaSurvey` module include (1) a string to include in the title of the survey and (2) the initial and final values for the loop

IPO CHART

Main Program   Function   Procedure   (circle one)

Name ______________________

Class	Identifier	Description
INPUT	year85	array of 4 sodas for completed year
	year86	array of 4 sodas for present year
	choice	option number selection
	month	number of present month
PROCESSING	initial	initial value for loop
	final	final value for loop
	i	loop index
	title	identifies soda survey for display

	Identifiers	Drawing
OUTPUT		

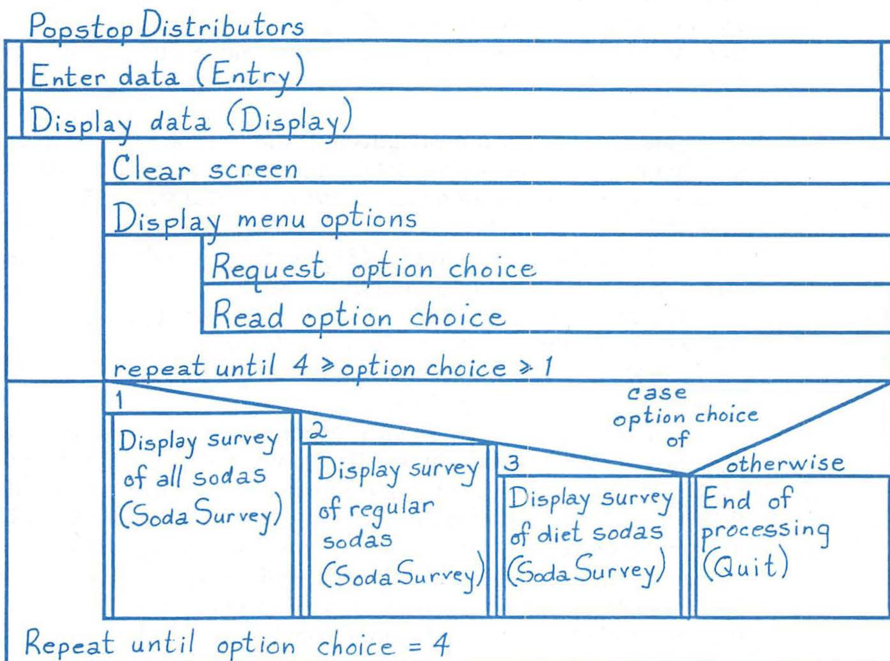


FIGURE 12.17

```
program PopstopDistributors;
{Sales survey for regular and diet soft drinks}
 type
  soda = record
    name : string;
    sales : array[1..12] of integer
   end;
  drinks = array[1..4] of soda;
 var
  year85,               {1985 sales}
  year86           {1986 sales}
   : drinks;
  choice,            {menu selection}
  month      {number of last completed month}
   : integer;


   begin
    Entry;
    Display;
    repeat
     page;
     writein('         Popstop Distributors Sales Survey');
     writeln;
     writeln('<1>  Individual Soda Survey');
     writeln('<2>  Regular Soda Survey');
     writeln('<3>  Diet Soda Survey');
     writeln('<4>  Quit');
     writeln;
     writeln;
     repeat
      write('Enter the number of your choice:  ');
      readln(choice)
     until (choice <= 4) and (choice >= 1);
     case choice of
      1 :
       SodaSurvey('All', 1, 4);
      2 :
       SodaSurvey('Regular', 1, 2);
      3 :
       SodaSurvey('Diet', 3, 4);
      otherwise
       Quit
     end
    until choice = 4
   end.
```

FIGURE 12.18

IPO CHART

Main Program Function (Procedure) (circle one)

Name Entry

Class	Identifier	Description
INPUT	year85	array of soda sales in 1985
	year86	array of soda sales in 1986
PROCESSING	i	loop index for months
	j	loop index for soda sales array

	Identifiers		Drawing
OUTPUT	year 85		
	year 86		

FIGURE 12.19 *(continued)*

that is used to access the data from the record array. The initial and final values determine whether the survey displayed includes all the sodas in the array, only regular sodas, or only the diet sodas.

When Option 4, `Quit,` is selected from the menu, the program ends. Figure 12.22 lists the `Quit` module and its run.

The special advantages of a program that has been written in modular form become apparent when the specifications for the original problem are changed. Let us assume that Popstop Distributors wishes to expand the monthly sales reports to include year-to-date summaries. If a year-to-date summary is required after the third month of the current year has been completed, the corresponding report would summarize the number of cases of each soda sold for the first three months of the current year and would compare that figure with the first three months of the previous year. In keeping with the form of the monthly soda survey reports, the percent difference between the two year-to-date sums is also calculated.

Because the original problem was designed in modular form, a modification of this kind is a simple task. The three steps in the modification are as follows.

Entry

- Request soda names and sales for 1985
- for j = 1 to 4
  - Request name for each soda
  - Enter name
  - for i = 1 to 12
    - Request sales for each month
    - Enter sales
- Wait until user is ready to continue (Delay)
- Clear screen
- Request number of last completed month this year
- Enter month number
- Clear screen
- Request monthly sales for each month completed
- for j = 1 to 4
  - Copy name for each soda
  - Display name
  - for i = 1 to monthnumber
    - Request Sales
    - Read monthly sales

FIGURE 12.19 *(continued)*

```
procedure Entry;
 {Enter data}
 var
  i, j      {loop indices}
   : integer;
begin
 writeln('Enter soda name and sales for 1985');
 for j := 1 to 4 do
  begin
   write('Enter name of soda # ', j : 2, ' ');
   readln(year85[j].name);
   for i := 1 to 12 do
    begin
     write('Month #', i : 3, '  ');
     readln(year85[j].sales[i])
    end
  end;
 Delay;
 page;
 write('Enter number of last completed month : ');
 readln(month);
 page;
 writeln('Enter monthly sales for each completed month this year');
 for j := 1 to 4 do
  begin
   year86[j].name := year85[j].name;
   writeln(year86[j].name);
   for i := 1 to month do
    begin
     write('      Month #', i : 4, '   ');
     readln(year86[j].sales[i])
    end
  end
end;
```

FIGURE 12.19

First, a year-to-date module must be designed to tally and display the additional report requirement. This module is placed before the main program (see Figure 12.23).

Second, a new option for this summary is added to the menu (see Figure 12.24).

Third, a case for this new option is added to the **case** structure in the main program (see Figure 12.25).

A run of the year-to-date module is illustrated in Figure 12.26, and the entire program is listed in Figure 12.27.

IPO CHART

Main Program Function (Procedure) (circle one)

Name Display

Class	Identifier	Description
INPUT	year 85	soda sales array – 1985
	year 86	soda sales array – 1986
PROCESSING	i	loop index for months
	j	loop index for soda sales array

OUTPUT	Identifiers		Drawing
	year 85		
	year 86		

FIGURE 12.20 *(continued)*

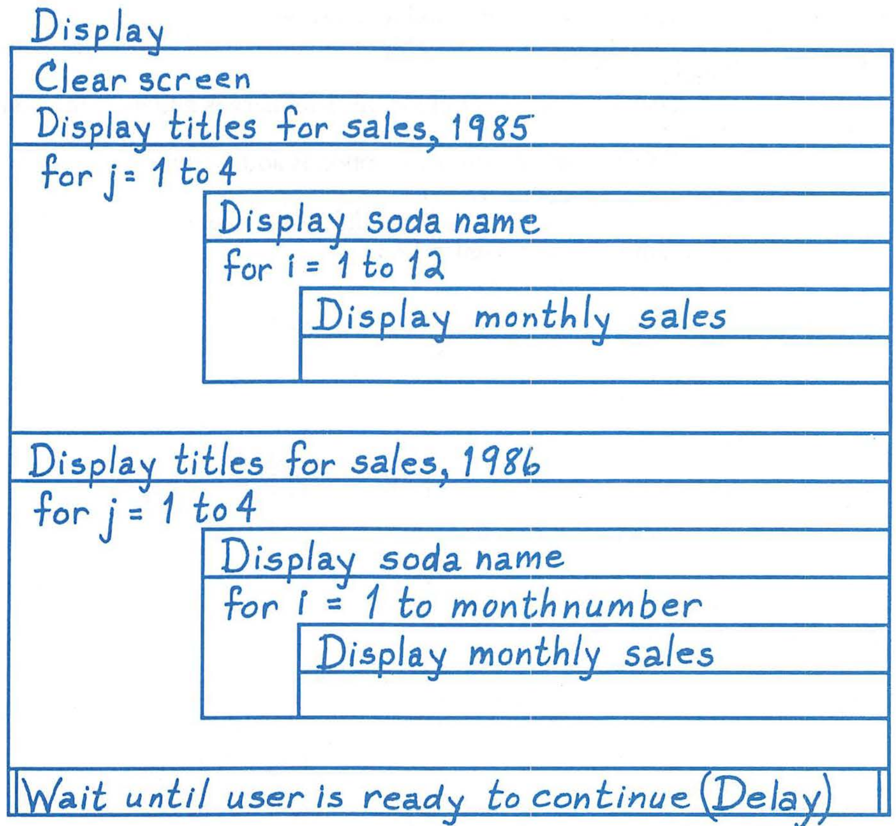


FIGURE 12.20 *(continued)*

```
procedure Display;
 {Displays input data}
 var
  i, j      {loop indices}
   : integer;
begin
 page;
 writeln('Input Data for 1985', 'Month' : 30);
 write('Soda Name' : 10);
 writeln('Jan' : 6, 'Feb' : 5, 'Mar' : 5, 'Apr' : 5, 'May' : 5, 'Jne' : 5, 'Jly' : 5, 'Aug' : 5, 'Sep' : 5,
      'Oct' : 5, 'Nov' : 5, 'Dec' : 5);
 writeln;
 for j := 1 to 4 do
  begin
   write(year85[j].name : 10);
   for i := 1 to 12 do
    write(year85[j].sales[i] : 4);
   writeln;
  end;
 writeln;
 writeln;
 writeln('Input Data for 1986');
 for j := 1 to 4 do
  begin
   write(year86[j].name : 10);
   for i := 1 to month do
    write(year86[j].sales[i] : 4);
   writeln;
  end;
 Delay
end;
```

FIGURE 12.20

SodaSurvey

Clear screen

Initialize sum85, sum86, sumchange

Display titles for survey

for j = initial value to final value

- display soda name, sales 1985, sales 1986 for month i
- change = sales 86 -- sales 85
- perchange = change / sales 85 x 100
- display change, perchange
- sum 85 = sum 85 + sales 85
- sum 86 = sum 86 + sales 86
- sumchange = sumchange + change

sumperchange = sumchange / sum 85 x 100

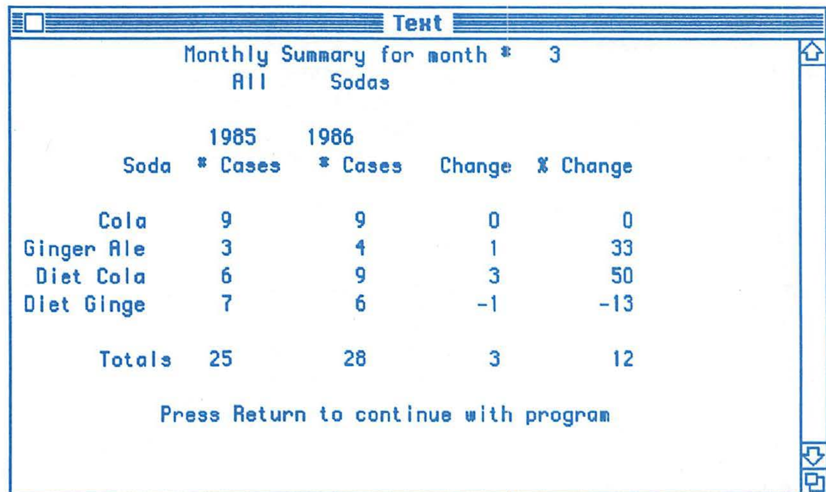
Display sum85, sum86, sumchange, sumperchange

Wait until user is ready to continue (Delay)

FIGURE 12.21 *(continued)*

```
procedure SodaSurvey (a : string;
        b, c : integer);
 {Calculates differences, percentage differences in sales 1985 and 1986}
 {a : portion of survey desired (all, regular, or diet)}
 {b : lower range for columns of portion of survey desired}
 {c :  upper range for columns of portion of survey desired}
 var
  change,              {sales difference from 1985 to 1986}
  perchange  {percent change in sales}
   : integer;
  sum85,               {total of 1985 sales for month}
  sum86,               {total of 1986 sales for month}
  sumchange,            {total of all changes for month}
  sumperchange   {total of all percent changes for month}
   : integer;
  j,                      {loop index}
  i      {month indicator}
   : integer;
begin
 page;
 sum85 := 0;
 sum86 := 0;
 sumchange := 0;
 i := month;
 writeln('Monthly Summary for month #' : 40, i : 3);
 writeln(a : 20, 'Sodas' : 10);
 writeln;
 writeln('1985' : 19, '1986' : 8);
 writeln('Soda' : 12, '# Cases' : 9, '# Cases' : 10, ' Change ' : 10, '% Change ' : 10);
 writeln;
 for j := b to c do
  begin
   write(year85[j].name : 10, year85[j].sales[i] : 6);
   write(year86[j].sales[i] : 10);
   change := year86[j].sales[i] - year85[j].sales[i];
   perchange := trunc(change / year85[j].sales[i] * 100 + 0.5);
   writeln(change : 10, perchange : 10);
   sum85 := sum85 + year85[j].sales[i];
   sum86 := sum86 + year86[j].sales[i];
   sumchange := sumchange + change
  end;
 sumperchange := trunc(sumchange / sum85 * 100 + 0.5);
 writeln;
 write('Totals' : 10, sum85 : 6, sum86 : 10, sumchange : 10, sumperchange : 10);
 Delay
end;
```

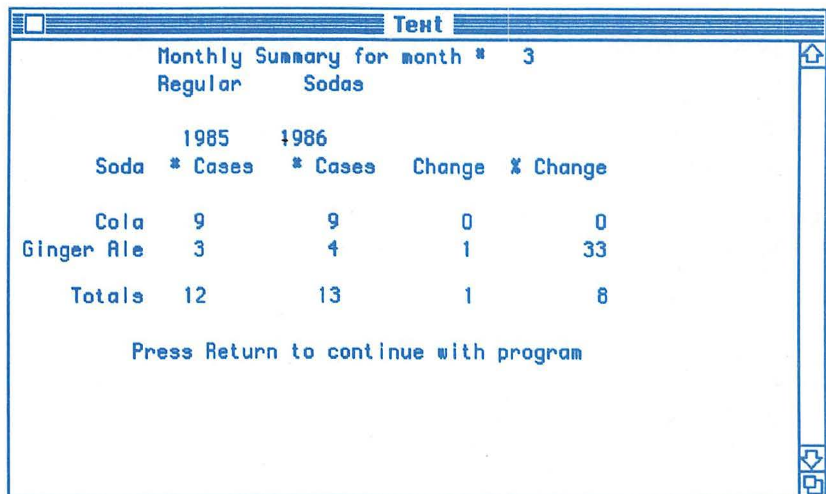
FIGURE 12.21


Text

Monthly Summary for month # 3
All Sodas

Soda	1985 # Cases	1986 # Cases	Change	% Change
Cola	9	9	0	0
Ginger Ale	3	4	1	33
Diet Cola	6	9	3	50
Diet Ginge	7	6	-1	-13
Totals	25	28	3	12

Press Return to continue with program


Text

Monthly Summary for month # 3
Regular Sodas

Soda	1985 # Cases	1986 # Cases	Change	% Change
Cola	9	9	0	0
Ginger Ale	3	4	1	33
Totals	12	13	1	8

Press Return to continue with program

FIGURE 12.21A *(continued)*

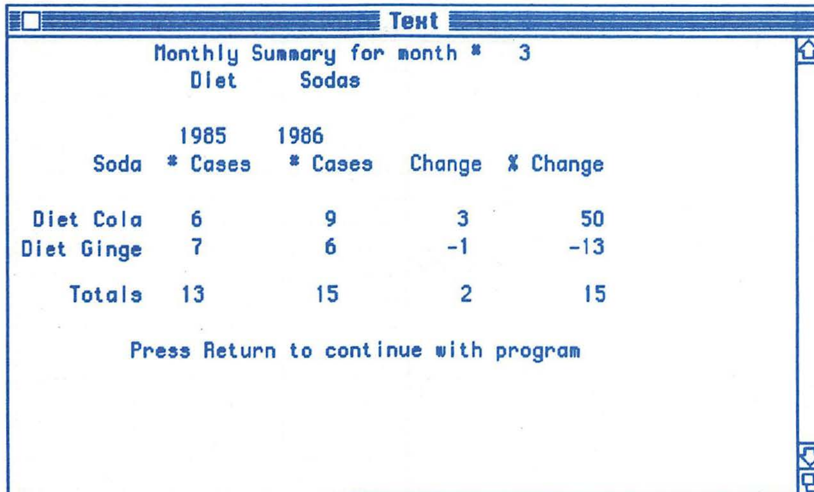


FIGURE 12.21A

IPO CHART

Main Program   Function   (Procedure) (circled)   (circle one)

Name *Quit*

Class	Identifier	Description
INPUT		
PROCESSING		

OUTPUT	Identifiers		Drawing
	msg		

Quit

Display message — Processing complete

```
procedure Quit;
begin
 writeln;
 writeln(' = = = > Processing Complete <= = = ' : 40)
end;
```

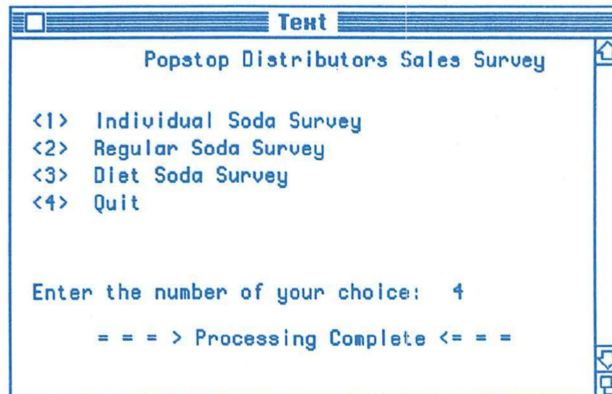


FIGURE 12.22

YearToDate

- Clear screen
- Initialize grtot85, grtot86, to zero
- Display survey titles
- for j = 1 to4
  - Initialize tot85, tot86 to zero
  - for i = 1 to month
    - tot85 = tot85 + sales for that soda
    - tot86 = tot86 + sales for that soda
  - Change = tot86 - tot85
  - perchange = change/tot85 * 100
  - grtot85 = grtot85 + tot85
  - grtot86 = grtot86 + tot86
  - display soda name, tot85, tot86, change, perchange
- grchange = grtot86 - grtot85
- grper = grchange/grtot85 * 100
- Display totals grtot85, grtot86, grchange, grper
- Wait until user is ready to continue (Delay)

FIGURE 12.23 *(continued)*

```
procedure YearToDate;
 {Calculates and displays year-to-date summaries up to completed month}
 var
  grtot85,                    {sums year-to-date totals for all sodas for 1985}
  grtot86,                    {sums year-to-date totals for all sodas for 1986}
  tot85,                  {year-to-date sum of number of cases sold for a soda in 1985}
  tot86,                 {year-to-date sum of number of cases sold for a soda in 1986}
  change,                {difference between tot85 and tot86 }
  perchg,              {percent change between tot85 and tot86}
  grchg,                {difference between grtot85 and grtot86}
  grper              {percent change between grtot85 and grtot86}
   : integer;
  i, j       {loop indices}
   : integer;
begin
 page;
 grtot85 := 0;
 grtot86 := 0;
 writeln('Year-To-Date Survey' : 30);
 writeln('Month  #' : 21, month : 2);
 writeln;
 writeln('1985' : 19, '1986' : 10);
 writeln('Soda' : 12, '# Cases' : 9, '# Cases' : 10, 'Change' : 10, '% Change' : 10);
 for j := 1 to 4 do
  begin
   tot85 := 0;
   tot86 := 0;
   for i := 1 to month do
    begin
     tot85 := tot85 + year85[j].sales[i];
     tot86 := tot86 + year86[j].sales[i]
    end;
   change := tot86 - tot85;
   perchg := trunc(change / tot85 * 100 + 0.5);
   grtot85 := grtot85 + tot85;
   grtot86 := grtot86 + tot86;
   writeln(year85[j].name : 10, tot85 : 6, tot86 : 10, change : 10, perchg : 10);
  end;
 grchg := grtot86 - grtot85;
 grper := trunc(grchg / grtot85 * 100 + 0.5);
 writeln;
 writeln('Totals' : 10, grtot85 : 6, grtot86 : 10, grchg : 10, grper : 10);
 Delay
end;
```

FIGURE 12.23

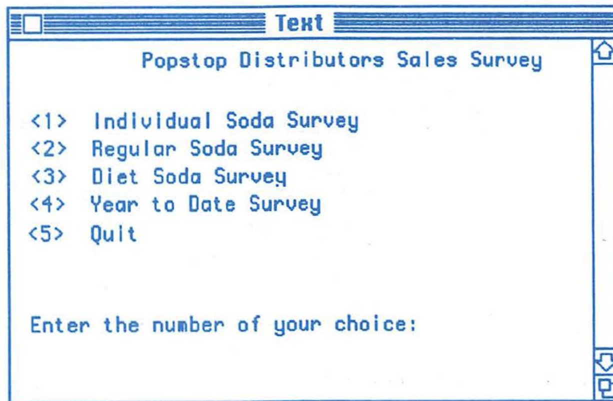


FIGURE 12.24

```
begin (*main program*)
 Entry;
 Display;
 repeat
  page;
  writeln('          Popstop Distributors Sales Survey');
  writeln;
  writeln('<1>  Individual Soda Survey');
  writeln('<2>  Regular Soda Survey');
  writeln('<3>  Diet Soda Survey');
  writeln('<4>  Year To Date Survey');
  writeln('<5>  Quit');
  writeln;
  writeln;
  repeat
   write('Enter the number of your choice:  ');
   readln(choice)
  until (choice <= 5) and (choice >= 1);
  case choice of
   1 :
    SodaSurvey('All', 1, 4);
   2 :
    SodaSurvey('Regular', 1, 2);
   3 :
    SodaSurvey('Diet', 3, 4);
   4 :
    YearToDate;
   otherwise
    Quit
  end
 until choice = 5
end.
```

FIGURE 12.25

```
                    Text
            Year-To-Date Survey
              Month  #  3

              1985      1986
        Soda   # Cases   # Cases   Change  % Change
       Cola     21        23         2        10
 Ginger Ale     15        13        -2       -12
  Diet Cola     18        21         3        17
 Diet Ginge     11        12         1         9

     Totals     65        69         4         6

        Press Return to continue with program
```

FIGURE 12.26

```
program PopstopDistributors;
{Sales survey for regular and diet soft drinks}
 type
  soda = record
    name : string;
    sales : array[1..12] of integer
   end;
  drinks = array[1..4] of soda;{may expand size of this array with new memory}
 var
  year85,           {1985 sales}
  year86         {1985 sales}
   : drinks;
  choice,          {menu selection}
  month           {number of last month completed}
   : integer;

 procedure Delay;
  {Permits user to view screen display and continue at own option}
  var
   ans         {continuation response}
    : string;
 begin
  writeln;
  writeln;
  write('            Press Return to continue with program');
  readln(ans)
 end;
```

FIGURE 12.27 *(continued)*

```
procedure Entry;
 {Data is read from keyboard}
 var
  i, j      {loop indices}
   : integer;
begin
 writeln('Enter soda name and sales for 1985');
 for j := 1 to 4 do
  begin
   write('Enter name of soda #', j : 2, ' ');
   readln(year85[j].name);
   for i := 1 to 12 do
    begin
     write('Month #', i : 3, '  ');
     readln(year85[j].sales[i])
    end
  end;
 Delay;
 page;
 write('Enter number of last completed month : ');
 readln(month);
 page;
 writeln('Enter monthly sales for each completed month this year');
 for j := 1 to 4 do
  begin
   year86[j].name := year85[j].name;
   writeln(year86[j].name);
   for i := 1 to month do
    begin
     write('      Month #', i : 4, '   ');
     readln(year86[j].sales[i])
    end
  end
end;
```

FIGURE 12.27 *(continued)*

```
 procedure Display;
  {Displays input data}
  var
   i, j      {loop indices}
    : integer;
begin
 page;
 writeln('Input Data for 1985', 'Month' : 30);
 write('Soda Name' : 10);
 writeln('Jan' : 6, 'Feb' : 5, 'Mar' : 5, 'Apr' : 5, 'May' : 5, 'Jne' : 5,
      'Jly' : 5, 'Aug' : 5, 'Sep' : 5,
      'Oct' : 5, 'Nov' : 5, 'Dec' : 5);
 writeln;
 for j := 1 to 4 do
  begin
   write(year85[j].name : 10);
   for i := 1 to 12 do
    write(year85[j].sales[i] : 4);
   writeln;
  end;
 writeln;
 writeln;
 writeln('Input Data for 1986');
 for j := 1 to 4 do
  begin
   write(year86[j].name : 10);
   for i := 1 to month do
    write(year86[j].sales[i] : 4);
   writeln;
  end;
 Delay
end;
procedure SodaSurvey (a : string;
         b, c : integer);
 {Calculates differences, percentage differences in sales 1985 and 1986}
 {a : portion of survey desired (all, regular, or diet)}
 {b : lower range for columns of portion of survey desired}
 {c : upper range for columns of portion of survey desired}
```

FIGURE 12.27 *(continued)*

```
  var
   change,                {sales difference from 1985 to 1986}
   perchange      {percent change in sales}
    : integer;
   sum85,              {total of 1985 sales for month}
   sum86,               {total of 1985 sales for month}
   sumchange,          {total of all changes for month}
   sumperchange    {total of all percent changes for month}
    : integer;
   j,           {loop index}
   i      {month indicator}
    : integer;
 begin
  page;
  sum85 := 0;
  sum86 := 0;
  sumchange := 0;
  i := month;
  writeln('Monthly Summary for month #' : 40, i : 2);
  writeln(a : 20, 'Sodas' : 10);
  writeln;
  writeln('1985' : 19, '1986' : 10);
  writeln('Soda' : 12, '# Cases' : 9, '# Cases' : 10, ' Change ' : 10, '% Change ' : 10);
  writeln;
  for j := b to c do
   begin
    write(year85[j].name : 10, year85[j].sales[i] : 6);
    write(year86[j].sales[i] : 10);
    change := year86[j].sales[i] - year85[j].sales[i];
    perchange := trunc(change / year85[j].sales[i] * 100 + 0.5);
    writeln(change : 10, perchange : 10);
    sum85 := sum85 + year85[j].sales[i];
    sum86 := sum86 + year86[j].sales[i];
    sumchange := sumchange + change
   end;
  sumperchange := trunc(sumchange / sum85 * 100 + 0.5);
  writeln;
  write('Totals' : 10, sum85 : 6, sum86 : 10, sumchange : 10, sumperchange : 10);
  Delay
 end;
```

FIGURE 12.27 *(continued)*

```
procedure YearToDate;
{Calculates and displays year-to-date summaries up to completed month}
 var
  grtot85,            {sums year-to-date totals for all sodas for 1985}
  grtot86,            {sums year-to-date totals for all sodas for 1986}
  tot85,          {year-to-date sum of number of cases sold for a soda in 1985}
  tot86,          {year-to-date sum of number of cases sold for a soda in 1986}
  change,         {difference between tot85 and tot86 }
  perchg,        {percent change between tot85 and tot86}
  grchg,         {difference between grtot85 and grtot86}
  grper       {percent change between grtot85 and grtot86}
   : integer;
  i, j      {loop indices}
   : integer;
begin
 page;
 grtot85 := 0;
 grtot86 := 0;
 writeln('Year-To-Date Survey' : 30);
 writeln('Month  #' : 21, month : 2);
 writeln;
 writeln('1985' : 19, '1986' : 10);
 writeln('Soda' : 12, '# Cases' : 9, '# Cases' : 10, 'Change' : 10, '% Change' : 10);
 for j := 1 to 4 do
  begin
   tot85 := 0;
   tot86 := 0;
   for i := 1 to month do
    begin
     tot85 := tot85 + year85[j].sales[i];
     tot86 := tot86 + year86[j].sales[i]
    end;
   change := tot86 - tot85;
   perchg := trunc(change / tot85 * 100 + 0.5);
   grtot85 := grtot85 + tot85;
   grtot86 := grtot86 + tot86;
   writeln(year85[j].name : 10, tot85 : 6, tot86 : 10, change : 10, perchg : 10);
  end;
 grchg := grtot86 - grtot85;
 grper := trunc(grchg / grtot85 * 100 + 0.5);
 writeln;
 writeln('Totals' : 10, grtot85 : 6, grtot86 : 10, grchg : 10, grper : 10);
 Delay
end;
```

FIGURE 12.27 *(continued)*

```
  procedure Quit;
  begin
   writeln;
   writeln('===>Processing Complete<===' : 40)
  end;

 begin (*main program*)
  Entry;
  Display;
  repeat
   page;
   writeln('      Popstop Distributors Sales Survey');
   writeln;
   writeln('<1>  Individual Soda Survey');
   writeln('<2>  Regular Soda Survey');
   writeln('<3>  Diet Soda Survey');
   writeln('<4>  Year To Date Survey');
   writeln('<5>  Quit');
   writeln;
   writeln;
   repeat
    write('Enter the number of your choice:  ');
    readln(choice)
   until (choice <= 5) and (choice >= 1);
   case choice of
    1 :
     SodaSurvey('All', 1, 4);
    2 :
     SodaSurvey('Regular', 1, 2);
    3 :
     SodaSurvey('Diet', 3, 4);
    4 :
     YearToDate;
    otherwise
     Quit
   end
  until choice = 5
 end.
```

FIGURE 12.27

NONPROGRAMMING EXERCISES

1. Draw the design chart used to illustrate the module structure of the solution to the following Programming Exercises:
   a. Programming Exercise 3
   b. Programming Exercise 4
   c. Programming Exercise 5
   d. Programming Exercise 6
   e. Programming Exercise 8
   f. Programming Exercise 10
2. Use Pascal code to create a menu for each of the Programming Exercises listed in Nonprogramming Exercise 1.

PROGRAMMING EXERCISES

(B) 3. Write a program to process employee payroll records. The program should contain three modules: one to accept an employee's hourly pay rate and number of hours worked in a week and to return that information to a main program; a second to calculate the employee's gross pay; and a third to produce a report including the employee's name and the gross pay determined by the subprogram.

(B) 4. Add a module to Programming Exercise 3 to accept the amount of deductions and to return the employee's net pay. The net pay should be included in the report produced.

(B) 5. Add a module to Programming Exercise 4 to calculate the total wages paid by the company to all employees and to return that value to the main program. This total should be included in the report module.

(B) 6. Write a program to process the inventory records of the Mindgames Software Company. Each record contains the name of the software product, its selling price, and the number in the current inventory. The main program should accept the information for the five packages sold by Mindgames and should store these data in an array of records. Write the program so that it contains modules that perform the following functions:

- Change the selling price of each item by a given percent.
- Produce a list of all items whose inventory falls below a given level.
- Update the inventory after sales and deliveries.
- Determine the value of the entire inventory.
- Display an inventory list.

(G) 7. Given a list of the names of students registered for a course, write a menu-driven program that permits the following options:

- Sort the list alphabetically.
- Count the number of students enrolled.
- Add additional students.
- Output the list of names.
- Quit.

(G) 8. Write a menu-driven program that asks the user to input the coordinates of two points A and B, which can be displayed in the **Drawing** window. Include the following options:

- Calculate and output the slope of the line determined by A and B.
- Graph the line connecting A and B.
- Graph the line that passes through the midpoint of the line connecting A and B with the negative reciprocal of the slope of the line connecting the original points.
- Quit.

(B) 9. Write a program to simulate a theater ticket reservation agency. The program should provide a menu offering the customer four plays, each play having three seat categories. It should (1) display the number of seats available in each category and the price of each seat and (2) provide for an automatic updating of the inventory of seats as each customer purchase is processed.

(B) 10. Write a program to simulate the preparation of a bank statement that updates activities in four areas: savings accounts, checking accounts, bank loans, and charge accounts. The main program should be menu-driven, providing the user with the four options, and each option may provide a menu of its own. For example, the savings account option might provide a menu listing options such as passbook account, money-market account, certificate of deposit account, and the like. Output should show starting balances at the start of a month, interest credits, finance charges, deposits, withdrawals, dates of the transactions, and closing balances.

(G) 11. Write a program to profile a population. The data for each person in the group should include sex, marital status, age, number of years of formal education, and annual income. The program should be menu-driven and offer the user options to calculate the average annual salary for everyone in the group by sex, by education, by marital status, or by age.

(B) 12. Write a program to calculate the monthly gross incomes for the salespeople at the Friendly Investor Services. Salespeople are classified in four categories. The total amount of monthly base pay and commissions differs according to category, as illustrated in the following table:

CATEGORY	*MONTHLY BASE PAY*	*COMMISSION POLICY*	
A	$400	5% first	$ 50,000
		8% next	50,000
		10% over	100,000
B	350	3% first	$ 50,000
		5% next	50,000
		8% over	100,000
C	300	3% first	$100,000
		5% over	100,000
D	200	3% first	$100,000
		4% over	100,000

The program should permit the user to enter the name of each of the 20 salespeople, the category, and the total amount of sales for the one-month period. Output options should include a listing of a salesperson's monthly total income by name; a full listing of the total income figures for every salesperson, sorted in descending order; and the average total monthly income, either by category or for the entire staff.

(G) 13. Write a program, for an elementary school pupil, that presents a drill in basic arithmetic skills. The program offers the pupil a selection of exercises in addition, in subtraction, in multiplication, and in division. Each option gives one question, awaits a response from the student, and displays an appropriate message telling the pupil if the response was correct or not. This loop repeats five times. In addition, a tally of the number of correct responses should be kept and output at the close of the drill session, again with an appropriate message. The random function may be used to generate the numbers in the exercises.

(G) 14. Write a program that allows a user to create a picture in the **Drawing** window with a menu that provides the following options: draw a line, draw a framed rectangle, draw a solid rectangle, draw a circle, and draw a solid circle. After each object is drawn, the user should have the option of erasing that object alone, not the entire **Drawing** window, before returning to a main menu.

CHAPTER 13

Advanced Data File Techniques

13.1 INTRODUCTION

The retrieval of previously stored information is an important activity in many data processing environments. For example, employment agencies keep files for client requests and applicant qualifications; doctors consult patient medical information stored in files; businesses maintain accounting and inventory files; schools access information stored in student record files. Required information may be recovered from any of these files by knowing how the information is arranged in the file. For example, two of a number of indexing methods are (1) alphabetically according to client names and (2) numerically according to Social Security numbers or account numbers.

We have already solved problems by storing and accessing data in simple external files (see Chapter 5). In this chapter, we explore and expand the use of external files to accommodate a more sophisticated data structure, the record. Instead of considering a file of integer, real, or string, these files are files of records. This organization of records permits different types of data to be stored as a single component of a file and thus reflects the organization of information in a home or office physical file. In a payroll application, employee records are used to process paychecks, and each record consists of several fields of varying types including name (string), Social Security number (integer), and pay rate (real).

This chapter reflects real-world applications by letting the computer replace the file cabinet. The speed and ease of manipulating and maintaining the data in a file using the computer explains the rapid transition of office (and, to a lesser degree, home) management from manual to machine work.

In order to create a file of records, you must have the ability to design the record for file storage, create the file to hold those records, store and access information from any record in the file, add and remove records from the file, search through the file for specific records, and change parts of any record in the file.

13.2 CREATING A FILE OF RECORDS

The declaration for a file of records requires at least two statements. The first one defines the record with its field identifiers and data types. The second statement defines the file type whose components are those records. The **var** section can then define identifiers to be of that file type.

FORMAT:
```
type
        <type identifier> = record
         <field identifier> : <data type>;
         <field identifier> : <data type>;
            ...
         <field identifier> : <data type>
        end;
        <file type> = file of <type identifier>;
       var
        <file identifier> : <file type>;
```

The file type can be defined in the **var** section instead of in the **type** section. The following example is a simplified payroll file design for the ABC Company.

Example:

```
type
 employee = record
   name : string;
   ssno : integer;
   hrrate : real;
   nohrs : real;
   gross : real;
   noofdep : integer;
   deduct : real;
   net : real;
   ytd : real
  end;
 list = file of employee;
var
 ABCCo : list;
```

The fields listed in the record `employee` represent some of the information that is needed to determine an employee's salary. The `ytd` field holds the year-to-date salary and is the last field in the record. `list` is defined as a file of these records, and the identifier, ABCCo, is the internal file name for a file of type `list`.

Now that a file of records has been defined and created in the declaration section, it can be opened and connected to an external file of the exact same structure. Once this connection is established, the internal file can send data to and receive data from the external file. The **open** statement accomplishes linking. Its format is repeated from Chapter 5.

THE OPEN FILE STATEMENT

FORMAT: **open** (`<file identifier>`, `<file name>`);

The statement `open (ABCCo, 'Workers');` establishes a link between the internal file named ABCCo and the external file named `Workers`. Remember, the two files must have an identical component structure; that is, each record must consist of the same fields of the same type defined in the same order. If there is no external file named `Workers` on the disk, one is created. An external file icon now appears when the disk files are displayed. Figure 13.1 shows the icon for the `Workers` file. This type of file cannot be opened or printed by activating its icon.

FIGURE 13.1

In addition to connecting the files, the **open** statement creates a file "pointer" that points at records in a file. Since only one record can be activated at a time, only one record can be written to or read from the file at a time. The pointer identifies that record. Once a record has been used, the file pointer moves to the next record in the file, and that record becomes the active record and can be written to or read from.

File records are numbered consecutively beginning with 0, and after the **open** statement is executed, the file pointer is at the first record in the file, Record 0.

13.3 WRITING TO A FILE OF RECORDS

After creating and opening a file, data can be stored in it. An identifier of <record type> should be declared in the **var** section of the program. This identifier is used to hold the information that is to be written to the active record in the file. Once the information for a record is complete, it is put into the file by a write-to-file statement.

THE **WRITE** FILE STATEMENT

FORMAT: **write(**<file identifier>, <record identifier>**);**

This statement places the record identified by <record identifier> in the file represented by <file identifier> at the current position of the file pointer.

In the following example, a study is to be made of the age and sex of participants in a speed-reading experiment in a school. The information for each student is placed in a four-field record, and the record is stored in a file.

Example:

```
type
 person = record
   name : string;
   age : integer;
   sex : char;
   speed : real
  end;
 Trial = file of person;
var
 human : person;
 Group : Trial;
begin
  . . .
 open(Group,'ExperimentA');
with human do
 begin
  write ('Name ;');
  readln(name);
  write ('Age :  ');
  readln(age);
  write('Sex :')
  readln(sex);
  write('Speed Achieved :  ');
  readln(speed);
  write(Group, human)
 end;
. . .
```

The **open** statement associates the internal file **Group** with the external file **ExperimentA** and "places" a file pointer on the first record in each of the files. The identifier **human** is a record of the same structure as those stored in the file and receives information for each of its fields from the keyboard. When the record is complete, the contents of **human** are stored in the file by the write-to-file statement. The file pointer then moves to the next record in the file, Record 1.

THE **FILEPOS** FUNCTION

Filepos is a system-defined integer function. It returns the record number, file position, of the active record. The **filepos** value is always equal to the number of records in that file that precede the active record. After the write-to-file statement is executed, the file pointer moves to the next record, increasing the value of the **filepos** function by 1. The next record then becomes the active record.

FORMAT: **filepos** `(<file identifier>)`

The following example demonstrates how you can determine the number of records that were written to a file.

Example:

```
open(Group, 'ExperimentA');
write('Name : ');
readln(human.name);
while human.name<>'ZZZ' do
 begin
  write('Age : ');
  readln(human.age);
  write(Group,human);
  write('Name : ');
  readln(human.name)
 end;
last := filepos(Group);
writeln('Records in file',last:3);
close(Group);
```

After the last write-to-file statement, the file pointer points to the next record. Therefore, the value returned by the **filepos** function is equal to the number of records that have been stored in the file.

Recall that when the connection between an internal file and an external file is no longer needed, the file is closed.

THE **CLOSE** FILE STATEMENT

FORMAT: **close** `(<file identifier>)`;

For example, `close(abc);` disassociates `abc` from the external file named in the corresponding open statement. No information can pass between the two files unless another **open** statement connects them.

13.4 READING FROM A FILE OF RECORDS

Information can be accessed from an existing external file by using a read-from-file statement.

THE **READ** FILE STATEMENT

FORMAT: **read(**`<file identifier>, <record identifier>`**)**;

A record is read from the external file specified in an **open** statement to the internal file identified by <file identifier> at the current position of the file pointer. That record is read from the file into <record identifier>. After a read-

from-file statement is executed, the file pointer points to the next record in the file.

The following example shows how the file ExperimentA, created in the previous example, can be opened and read from in another program.

Example:

```
var
 Area : trial;
 human : person;

begin
...
open(Area, 'ExperimentA');
read(Area, human);
with human do
 writeln(name, age, sex, speed);
```

The **open** statement connects the internal file **Area** with the existing external file **ExperimentA** and sets the file pointer at Record 0. The information stored in the first record is read into **human**, and the data in each field of that record are displayed.

THE EOF FUNCTION

The end-of-file (**eof**) function is a system-defined Boolean function that has a value *true* if the file pointer is beyond the last record in the file identified by <file identifier> and has a value *false* if it is not.

FORMAT: **eof(<file identifier>)**

The **eof** function can be used when reading from a file where the number of records in that file is not known in advance.

Example:

```
open(Group, 'ExperimentA');
while not (eof(Group)) do
 with human do
  begin
   read(Group, human);
   writeln(name:15, age:3)
  end;
writeln;
writeln('End of the file');
```

After the read-from-file statement is executed, the value of **filepos** increases by 1. This process moves the file pointer to the next record in the external file, and the contents of each record in the external file are read into **human** until the eof(Group) returns a value *true*. Now, the file pointer is past the last record in ExperimentA, not eof(Group) becomes false, and the **while...do** loop ends.

13.5 MODIFYING A FILE OF RECORDS

One advantage of using the computer in establishing a file is the ease with which the structure can be modified. Often a file is changed by adding a record or changing the information stored in a record.

ADDING A RECORD

The process of adding a record to a file is known as "appending a record." The record is "attached" to the end of the file. The file pointer must be placed in the correct position (beyond the last record in the current file) before the write-to-file instruction is given. If the number of records in the file is not known, the file can be "rewound" by a **reset** statement so that the end of the file can be determined.

THE **RESET** FILE STATEMENT

FORMAT: **reset (<file identifier>);**

This statement sets the file pointer for <file identifier> at Record 0 for reading the file. **reset** can be used after the file has been opened.

Example:

```
reset(Group);
repeat
 read(Group,human)
until eof(Group);
write('Name of participant: ');
readln(human.name);
write('Age : ');
readln(human.age);
write('Sex <M/F> : ');
readln(human.sex);
write('Speed score : ');
readln(human.speed);
write(Group, human);
close(Group);
```

reset rewinds the file to Record 0. The file is read until the **eof** function returns a value *true*. The file pointer is then beyond the last record in the file. The **readln** statements place information from the keyboard into the fields of a record, and the write-to-file statement appends the record to the end of the file.

CHANGING A RECORD

Often the data in a specific field in a record must be changed or updated. Suppose the speed-reading rate for Roger Less has to be increased 10 percent in the file `ExperimentA`. If the record number is known, a **seek** statement can be used to locate that record.

THE **SEEK** FILE STATEMENT

This option permits you to select any record for updating without going through other records in the file. Once the record is located and read, it can be updated and written back to the file.

FORMAT: **seek** `(<file identifier>, <file position>);`

The **seek** procedure places the file pointer at the record represented by the integer <file position> in the internal <file identifier>. The record selected by the file pointer becomes active and can be read or written to. Any field of that record can be altered.

Example:

```
seek(Group, 20);
read(Group, human);
human.speed := human.speed * 1.10;
seek(Group, 20);
write(Group.human);
```

The first **seek** statement locates the 21st record in the file. That record is read into `human`, and the speed field is increased by 10 percent. The file pointer has moved to the next record after the read-from-file statement is executed and must be moved back to the 21st record in order to update the correct record. A second **seek** statement sets the proper position.

If the number of the record that is to be updated is not known but the name of the participant is known, a sequential search for the desired record can be performed. This file can be rewound to record and the file searched until the name field of a record matching the known name is found. Then the `speed` field can be corrected and the record written back to the file at the same position.

Example:

```
reset(Group);
repeat
 read(Group,human)
until (eof(Group))
  or (human.name := 'Roger Less');
if eof(Group) then
 writeln('No such person.')
else
 begin
  human.speed := human.speed * 1.10;
  seek(Group, filepos(Group) - 1);
  write(Group, human)
 end;
```

After the file is rewound, the **name** field in each record is checked to see if it matches the given name. The possibility of no match or the end of the file being found must be considered. Both options are dealt with in the selection structure. A **seek** statement is used to reposition the file pointer at the correct position for the write-to-file statement.

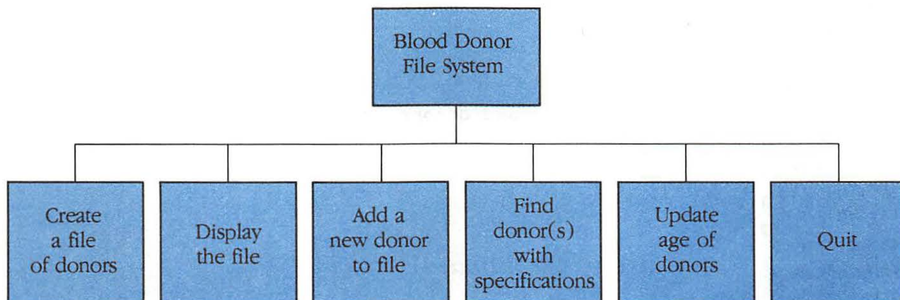
13.6 A PRACTICAL APPLICATION

The techniques used in the creation of this program have been discussed in the previous sections of this chapter. You are encouraged to see how these processes are applied in the solution of this problem.

Problem: Mercy Hospital is designing a file system to store information on blood donors. Each record should contain the name, address, age, blood type, and Rh factor for a donor. Once the file is created, the hospital must be able to display a list of all donors in the file, append the record of a new donor to the file, find the names and addresses of donors in the file who have a specific blood type and Rh factor, and update the ages of all the donors in the file once a year. Write a menu-driven program that offers each of the preceding options.

Figure 13.2 displays a modular design for the program.

The entire program is listed in Figure 13.3.



```
program BloodDonors;
{File management program for blood donors at Mercy Hospital}
 type
  info = record
    name : string;
    address : string;
    age : integer;
    bltype : string;
    Rhfactor : char
   end;
  list = file of info;
 var
  MercyHosp     {file of records containing donor information}
   : list;
  option            {menu selection}
   : integer;

 procedure Delay;
  {Permits user to view screen display before continuing}
  var
   ans       {continuation response}
    : char;
 begin
  writeln;
  write('Press Return to continue' : 30);
  readln(ans)
 end;

 procedure CreateAFile;
  {Creates a file of donor information}
  var
   x       {blood donor information record}
    : info;
   i        {loop index}
    : integer;
   count     {number of blood donors}
    : integer;
 begin
  page;
  rewrite(MercyHosp);
  write('Enter number of blood donors : ');
  readln(count);
  for i := 1 to count do
   with x do
    begin
```

FIGURE 13.3 *(continued)*

```
      write('Name : ');
      readln(name);
      write('Address : ');
      readln(address);
      write('Age : ');
      readln(age);
      write('Blood type (A, B, AB, O ): ');
      readln(bltype);
      write('Rh factor ( + or - ) : ');
      readln(Rhfactor);
      write(MercyHosp, x)
     end;
   Delay
  end;

  procedure WriteAFile;
   {Displays contents of the file}
   var
    x       {blood donor information record}
    : info;
 begin
  page;
  reset(MercyHosp);
  writeln('Name ' : 10, 'Address' : 15, 'Age' : 6, 'BloodType' : 10, ' Rh Factor ' : 10);
  while not (eof(MercyHosp)) do
   begin
    read(MercyHosp, x);
    with x do
     begin
      write(name : 12);
      write(address : 15);
      write(age : 3);
      write(bltype : 8);
      writeln(Rhfactor : 6);
     end
   end;
  Delay
 end;
```

FIGURE 13.3 *(continued)*

```
procedure AddADonor;
 {Appends a record to the end of the file}
 var
  x     {blood donor information record}
   : info;
begin
 page;
 reset(MercyHosp);
 repeat
  read(MercyHosp, x)
 until eof(MercyHosp);
 writeln('This is record # ', filepos(MercyHosp) + 1);
 writeln;
 with x do
  begin
   write('Name : ');
   readln(name);
   write('Address : ');
   readln(address);
   write('Age : ');
   readln(age);
   write('Blood type <A, B, AB, O >: ');
   readln(bltype);
   write('Rh factor < + or - > : ');
   readln(Rhfactor);
   write(MercyHosp, x)
  end;
 Delay
end;

procedure FindADonor;
 {Searches file for donors with specified data}
 var
  x     {blood donor information record}
   : info;
  bt     {sought-after blood type}
   : string;
  rhf     {sought-after Rh factor}
   : char;
  valid     {indicates whether a donor has been found}

   : boolean;
begin
 page;
 writeln('Search for blood donors with specified blood type and Rh factor');
 writeln;
 write('Enter the blood type <A,B,AB, or O> :');
```

FIGURE 13.3 *(continued)*

```
  readln(bt);
  write('Enter the Rh factor : ');
  readln(rhf);
  reset(MercyHosp);
  valid := false;
  while not (eof(MercyHosp)) do
   begin
    read(MercyHosp, x);
    if (x.bltype = bt) and (x.Rhfactor = rhf) then
     begin
      writeln(x.name : 12, x.address : 20);
      valid := true
     end
   end;
  if not (valid) then
   writeln('There are no donors that match the requirements ');
  Delay
 end;

 procedure UpdateRecords;
  {Yearly update of the age field in all records in the file}
  var
   x      {blood donor information record}
    : info;
 begin
  reset(MercyHosp);
  while not (eof(MercyHosp)) do
   begin
    read(MercyHosp, x);
    x.age := x.age + 1;
    seek(MercyHosp, filepos(MercyHosp) - 1);
    write(MercyHosp, x)
   end;
  page;
  writeln('Age update has been completed.' : 30)
 end;

 procedure Quit;
  {Terminate processing}
 begin
  page;
  writeln('End of processing.' : 30)
 end;
```

FIGURE 13.3 *(continued)*

```
begin(*main program*)
 open(MercyHosp, 'Donors');
 page;
 option := 0;
 while option <> 6 do
  begin
   page;
   writeln('Blood Donor List for Mercy Hospital' : 40);
   writeln;
   writeln('<1>  Create Blood Donor List');
   writeln('<2>  Display Blood Donor List');
   writeln('<3>  Add a donor to the List');
   writeln('<4>  List of available donors');
   writeln('<5>  Update age of donors');
   writeln('<6>  Quit');
   writeln;
   repeat
    write('Select option 1, 2, 3, 4, 5, or 6 ---> ');
    readln(option)
   until (option <= 6) and (option >= 1);
   case option of
    1 :
     CreateAFile;
    2 :
     WriteAFile;
    3 :
     AddADonor;
    4 :
     FindADonor;
    5 :
     UpdateRecords;
    otherwise
     Quit
   end
  end;
 close(MercyHosp)
end.
```

FIGURE 13.3

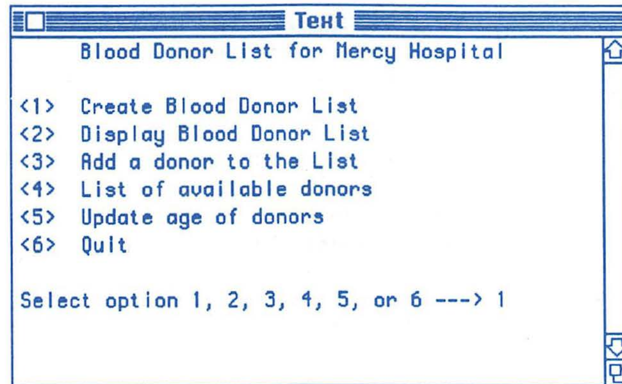


FIGURE 13.4

Figure 13.4 is a display of the program menu.

Displays of the runs for each of the menu selections are shown in Figure 13.5.

13.7 TYPICAL PROGRAMMING ERRORS

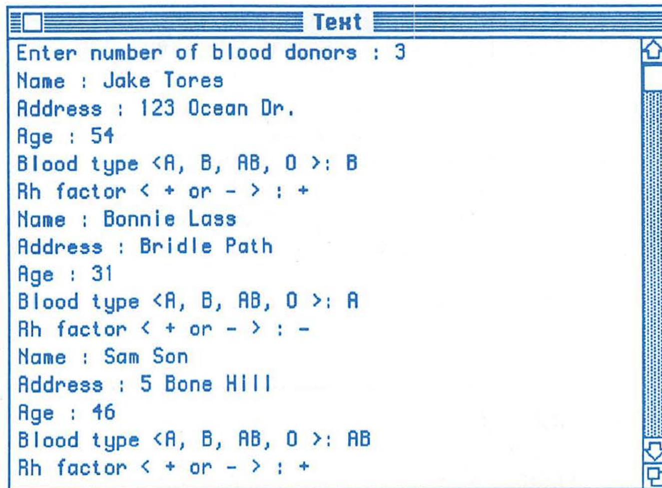
The following programming errors are typical:

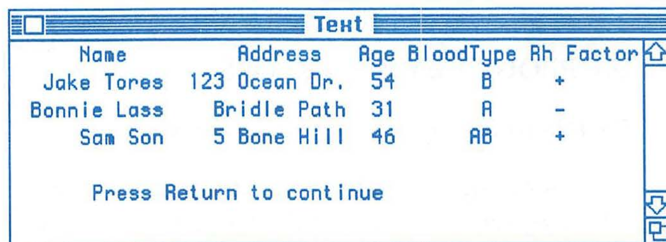
1. Associating files with dissimilar record structure
2. Incorrect positioning of the file pointer for reading from or writing to a record
3. Attempting to access data from an unopened file
4. Attempting to read beyond the end of a file

NONPROGRAMMING EXERCISES

1. Classify each of the following Pascal statements as valid or invalid. If the statement is invalid, explain why.

   b. `reset('Drivers');`
   c. `close;`
   d. `seek(100);`
   e. `filepos(n+1);`
   f. `writeln(f,x);`
   g. `read(f,x);`
2. Write a Pascal statement to perform the following file operations:
   a. Open an internal file *ESG* and connect it to an external file *JJD*.
   b. Place the file pointer at Record 3 in the *ESG* file.





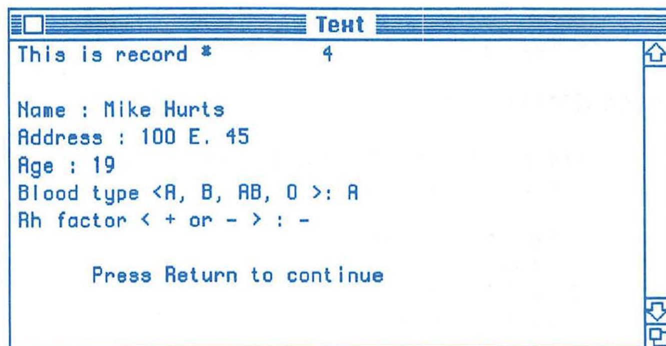


FIGURE 13.5 *(continued)*

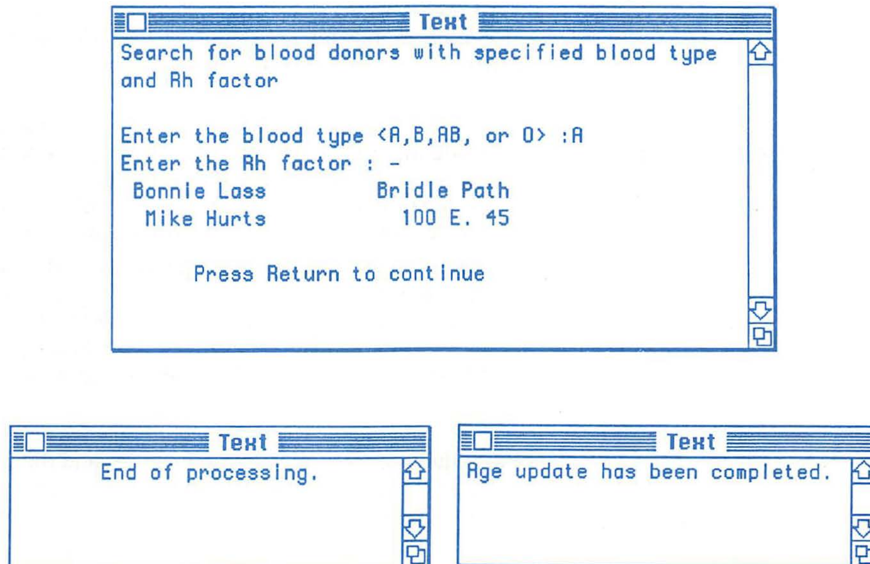


FIGURE 13.5

c. Return to the beginning of the *ESG* file.
d. Remove access to the *JJD* file.
e. Determine the current position of the file pointer in the *ESG* file.
f. Display the message *End of File* on the screen when the end of the *ESG* file is reached.

3. Devise an appropriate record format for each of the Programming Exercises:
   a. Programming Exercise 5
   b. Programming Exercise 7
   c. Programming Exercise 12
   d. Programming Exercise 14
   e. Programming Exercise 16

PROGRAMMING EXERCISES

(B) 4. Write a program to process employee records that contain an employee name, Social Security number, and hourly wage. Create a sample file of employee records containing the information specified above. Accept the current week's hours worked for each employee from the keyboard, read the additional information from the employee file, and output a listing of all employees by name, Social Security number, and gross weekly wage.

(B) 5. Write a program using files to help a manager make telephone calls to a firm's customers. Each file record should contain a customer name and telephone number. Accept a customer's name and search the file by customer name to find the associated telephone number.

(B) 6. Write a program to process a file of records containing the names, addresses, and loan payments due each month for customers of the Friendly Finance Co. The program should output a letter to each borrower stating the amount due on the loan.

(G) 7. Write a program to help the Pine City Police Department check for traffic violations. Prepare a file containing the license plate number of an operator, the name of the operator, and the number of violations issued in the past. The program should permit a search of the file by operator name or by license plate number and output the license plate number, name, and number of past violations of offenders.

(G) 8. Write a program for the Yule College Registrar that employs a file containing student names, class years, and current grade-point averages (based on a scale of four points). The program should permit a search of the file by student name to determine a student's grade-point average, calculate the average of all grade-point averages for students in the same class year, and determine the average of all the grade-point averages for all students at Yule College.

(B) 9. Write a program to help the hotel manager of the Elm Inn. Create a file that contains the room numbers at the inn, the names of room guests, and the room charge per day. The program should permit a search of all rooms to accommodate a room swap between guests for rooms with the same room charge or to update the guest name for any of the rooms.

(G) 10. Write a program to help a lawyer process her cases. Create a client file containing the client name, type of case (civil or criminal), the date the case was opened, and the date the case was closed. Some cases are pending, and the fourth field in each record should accordingly permit this determination. Display a listing of all pending cases, all closed cases, all pending civil cases, and all pending criminal cases.

(B) 11. Write a program to help in the processing of inventory for the New World Mail Order Co. Prepare a file that contains a product code number designation, current number in stock, the date of the last order, and the number of units in the last order. Permit a user to search the file by product number, search the file for all products whose current inventory is less than 50, and search the file for those products for which the last order was made in the previous year.

(G) 12. Write a program for the Good Buys Realty Mart to include a file containing data on current real estate listings. Design a file that contains the asking price of a home, owner's name, location of the home, and the type of house (colonial, cape, ranch, or split-level). The program should permit a salesperson to search the file by price, by location, or by style of home. If by price, the search should disclose all listings less than or equal to the stated price.

(G) 13. Write a program to maintain statistics for a bowling league. Create a file that includes the team name, current week's team average, and team season high average to date. Permit a user to display a listing of all the teams in the league, together with their current week's team average, arranged in descending order from the highest to the lowest. The program should also permit a search of the file to see if any team's current week's average equals or exceeds the league record.

(G) 14. Write a program to assist the head librarian at the Inner City Library. Create a file of borrowed books where each record contains the title of a book, author, library classification number, and name of borrower. The program should permit a search of the file by title, by author, and by classification number.

(B) 15. Write a program to process the transactions at the Stage Door Video Rental store. Customer transactions are kept in a file containing the customer name, address, telephone number, title of film rented, date of the rental, and the cost of the rental per day. Allow the file to be searched by film title, customer name, or telephone number. In addition, the program should be capable of preparing a bill for a customer who returns a film by accepting the number of days rented and calculating a charge based on the following formula: first-day rental is the stated figure; second day, an additional $1 charge; for each day thereafter, a penalty charge of $2.50.

(B) 16. Write a menu-driven program to process the inventory records of the Mindgames Software Co. stored in a file. Each record contains the name of the software product, its selling price, and the number in the current inventory. The main program accepts the information for the five packages sold by Mindgames and stores these data in an array of records. Write a program that contains modules that offer the following options:

- Change the selling price of each item by a given percent.
- Produce a list of all items whose inventory falls below a given level.
- Update the inventory after sales and deliveries.
- Determine the value of the entire inventory.
- Display an inventory list.
- Add a new software product.

(G) 17. Write a program to profile a population. The data for each person in the group should include sex, marital status, age, number of years of formal education, and annual income. This information should be stored in a file with an appropriate record format. Design a menu-driven program that offers a user options to calculate the average annual salary for everyone in the group by sex, by education, by marital status, or by age.

(G) 18. Add the following two modules to the blood donor program at the end of the chapter:

- Delete a record from the file of donors based on the age field.
- Create an additional file with the records in alphabetical order.

CHAPTER 14

Advanced Pascal Structures

14.1 INTRODUCTION

In the previous chapters, you've been exposed to the "basics" of Pascal. Almost any problem is solvable using these basics. However, Pascal has some additional features that make it easier for you to code a problem solution and to make more efficient use of computer memory.

This chapter provides an introduction to some of the more esoteric statements in Pascal, including the definition of two new data types (enumerated and pointer) and their implementation, and the use of subprograms that call themselves (recursive subprograms).

14.2 ENUMERATED DATA TYPES

An enumerated data type is a data type in which the allowable values describing that type are listed *(enumerated)* in a type declaration.

FORMAT: **type**

```
        <type identifier> = (<list of values>);
```

You define the data type by giving the type a name, <type identifier>, and a set of allowable values enclosed within parentheses and separated by commas. Any attempt to employ a value other than those listed for an identifier declared to be of the given type produces an error message.

Enumerated data types are necessarily ordered. The first item listed is "less than" or comes before the second item, the second before the third, and so on.

You have used ordered data types before, although that designation was not given to them. **integer** and **char** are two such data types. They are considered ordered, because given any value from these types, it is always possible to

determine the next item that follows it. For example, the integer type 4 always follows 3, and the character n always follows m. **real** and **string** are not ordered data types since, given any value of these types, you cannot determine what the next item is. For example, it is impossible to determine what real number immediately follows 5.367. If you say 5.368, then you're missing 5.3677.

Examples:

Code	Explanation

`type`
`facecard = (jack, queen, king);`
 	
`var`
`honor : facecard;`
	The type called `facecard` has only three defined values: `jack`, `queen`, and `king`. Since **honor** is of type `facecard`, the only allowable values for **honor** are these three. In terms of ordering, `jack` precedes `queen`, and `queen` precedes `king`.

`type`
`day = (M, Tu, W, Th, F, Sa, Su);`
`weekday = (M..F);`
 	
`var`
`work : weekday;`
	The identifier **work** is of type `weekday`. `weekday` is a subrange of the type **day** consisting of all those values from `M` to `F`. Since enumerated data types are ordered, you can just list the first element in a range, followed by two dots and the last element. This is not equivalent to data of type **char** since `M` is not the same as `'M'`.

Two places in which enumerated data types simplify the meaning of identifiers and the logic of a program are when they are used in **case** statements and when they are used as subscripts or indices of an array. We previously said that only integers or characters were allowed as selectors in a **case** statement or indices for an array. However, generally speaking, any ordered data type can be employed for these applications. The two problems that follow illustrate this flexibility.

Problem: Write a program that accepts the name of a day of the week and prints the store hours for Kitchen Floorist on that day.

A complete program listing and sample run are shown in Figure 14.1.

Values entered for **when** may be in any combination of upper- or lowercase letters. MacPascal does not differentiate between upper- and lowercase letters when considering enumerated data types. After a value is entered for **when**, the **case** structure selects the hours corresponding to the matched value.

```
program StoreHours;
{Displays the hours that Kitchen Floorist is open}
 type
  days = (Mon, Tues, Weds, Thurs, Fri, Sat, Sun, Holiday);
{enumerated data type with 8 valid values}
 var
  when   {a day of the week}
   : days;
begin
 writeln('Enter a day of the week using the following abbreviations:');
 writeln('Mon, Tues, Weds, Thur, Fri, Sat, Sun, Holiday');
 write('---> ');
 readln(when);
 writeln;
 write('Kitchen Floorist hours for ', when, ' are:  ');
 case when of
  Mon, Tues, Weds :
   writeln('10:00 am - 6:00 pm');
  Thurs, Fri :
   writeln('10:00 am - 9:00 pm');
  Sat :
   writeln('9:00 am - 7:30 pm');
  Sun :
   writeln('11:00 am - 5:00 pm');
  Holiday :
   writeln('Store closed')
 end
end.
```

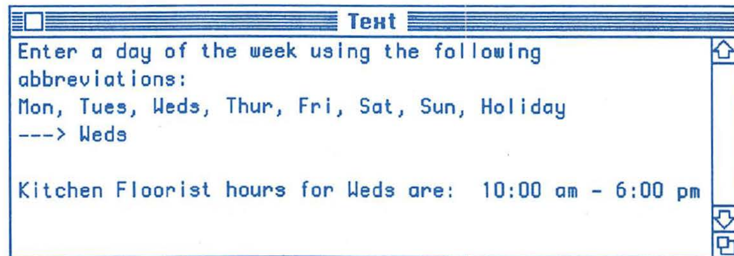


FIGURE 14.1

Problem: Write a program that simulates a registration process for enrollments in courses offered at Tylerville College. Create a record for each course with fields for the name of the course, the total seating capacity, and the number of seats already taken. Input consists of a course's code. If a course is not closed (the seats taken are not equal to the capacity), the count for the seats taken is updated and another request is made.

Figure 14.2 gives a complete listing and sample run of the solution.

The type declaration defines a type `course` that contains the courses that Tylerville offers. The record of type `enrollment` is an array of records of type `courseinfo`. The index of that array is of type `course` and therefore can only have values `EN12`, `EN13`, `MA15`, and `MA20`.

`SetUpRecord` initializes the fields of the array of records with the information necessary to process course registrations. The values for the enumerated data type `course` serve as values for the index of the **for...do** loop, in effect, cycling through records `size[EN12]`, `size[EN13]`, `size`

```
program CourseEnrollment;
{Updates the course enrollment for Tylerville College}
 type
  course = (EN12, EN13, MA15, MA20);    {course codes}
  courseinfo = record
    title : string;                  {title of the course}
    capacity : integer;           {number of students allowed in a class}
    seatstaken : integer        {number of seats currently taken}
   end;
  enrollment = array[course] of courseinfo;
 var
  size      {collection of information on courses}
   : enrollment;

 procedure SetUpRecord;
  {Initializes course information in records}
  var
   i    {loop control identifier}
    : course;
 begin
  for i := EN12 to MA20 do
   begin
    write('Enter course title for ', i, ':  ');
    readln(size[i].title);
    write('Enter total seating capacity for ', i, ':  ');
    readln(size[i].capacity);
    size[i].seatstaken := 0;
   end;
 end;
```

FIGURE 14.2 *(continued)*

```
procedure SignUp;
 {Accepts student request and updates count for courses}
 var
  j,                {loop control identifier}
  choice       {student course choice}
   : course;
  answer      {continue response}
   : char;
begin
 repeat
  page;
  writeln('Enter course desired from the list below:');
  for j := EN12 to MA20 do
   writeln(j : 6, size[j].title : 30);
  writeln;
  write('--->  ');
  readln(choice);
  with size[choice] do
   if capacity = seatstaken then
    writeln('Course is closed.')
   else
    seatstaken := seatstaken + 1;
  write('Do you want to continue (y/n)?');
  readln(answer)
 until answer = 'n'
end;

procedure Display;
 {Displays course enrollments for all courses}
 var
  k      {loop control identifier}
   : course;
begin
 page;
 writeln('Course' : 10, 'Course Title' : 25, 'Enrollment' : 20);
 for k := EN12 to MA20 do
  begin
   writeln(k : 10, size[k].title : 25, size[k].seatstaken : 12)
  end
end;

begin
 SetUpRecord;
 SignUp;
 Display
end.
```

Text

Course	Course Title	Enrollment
EN12	American Literature	0
EN13	Composition	1
MA15	College Algebra	0
MA20	Statistics	2

FIGURE 14.2

[MA15], and size[MA20]. The SignUp procedure lists the courses available and asks for a course code. Again, the values of course are used for the index of the **for...do** loop. If a seat is available, the seat count is updated and the process is repeated for the next student request. Display produces a summary of the course registrations.

14.3 POINTER DATA TYPES

One of the major problems inherent in using arrays in a program is that you may never be sure just how many array elements you will need. An error message results if you attempt to reference an array element that is outside of the boundaries specified in the declaration statement. To avoid this error, you have to declare a size for an array that is much larger than you currently require, to allow space for growth.

When an array is declared, the computer reserves consecutive memory locations for elements of the array. For example, if you declare an array to have 100 elements, then 100 locations are reserved for that array. These locations are not available for any other purpose. Not only that, but if 101 locations are needed, you would have to change the declaration in the program to accommodate the extra memory requirement. Declaring large arrays puts a strain on the system's resources, specifically on memory. Because the size must be declared in advance and does not change after declaration, an array is termed a *static data structure*.

A data type that allows you to make better use of available memory is the pointer. A *pointer data type* is a dynamic data structure that allocates memory cells as they are needed and deallocates cells when there is no longer any use for them. In effect, this structure can be thought of as a pool or heap of available locations. When a location is required, it is taken from the pool, and when it is no longer required, it is returned to the pool.

The format for a pointer data type is slightly different from other data types.

FORMAT:
```
type
    <type identifier> = ^<data type>;
```

The caret (^) preceding <data type> signifies that any identifier declared to be of ^ <data type> contains the address of a location whose contents are <data type>. In other words, that identifier points to a location. If a pointer is declared but does not currently point to a specific location, that pointer has value **nil**. **nil** is a keyword that indicates that a pointer does not currently reference another memory location.

When using pointers to reference data locations, you never know the exact address of the location. All you know is what memory location contains the address of that location.

Example:

`type`
` pointint = ^integer;`
 	
`var`
` p : pointint;`
	The identifier **p** points to a memory location that contains an integer; that is, **p** contains the address of a location whose content is an integer. When using **p** in a program, **p^** is the identifier for the contents of the location that **p** points to.

A pointer data type can also be declared in the **var** declaration of a program. The following declaration is equivalent to the one in the previous example:

```
var
 p : ^integer;
```

Figure 14.3 illustrates the pointer concept. The address for p^ is contained in location p. The contents of p^ are accessed indirectly (through p) rather than directly as in previous memory references.

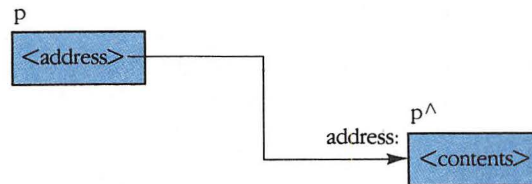


FIGURE 14.3

THE NEW PROCEDURE

MacPascal contains a system-defined procedure that pulls a memory location from the pool of available ones.

FORMAT: **new** `(<pointer identifier>);`

The **new** procedure selects the address of a memory location and stores that address in the location represented by <pointer identifier>. Each time the **new** procedure is called, a new memory location is selected.

Examples:

```
var
 p : ^real;

...
 new(p);
```

The pointer identifier **p** points to a location that contains real data. When **new(p)** is invoked, an address is selected from the pool and placed in the location identified by **p. p** points to the location **p^**, which can contain a real.

```
new(p);    { Step a }
new(p);    { Step b }
```

If **p** is a pointer identifier, the first call in Step a selects a location and lets **p** point to it. If **new** is invoked again using **p**(Step b), a new location is selected and **p** points to this location. The first location selected has no pointer pointing to it, and so you have no means of accessing the data stored in it. You have to be careful when employing **new**. See Figure 14.4.

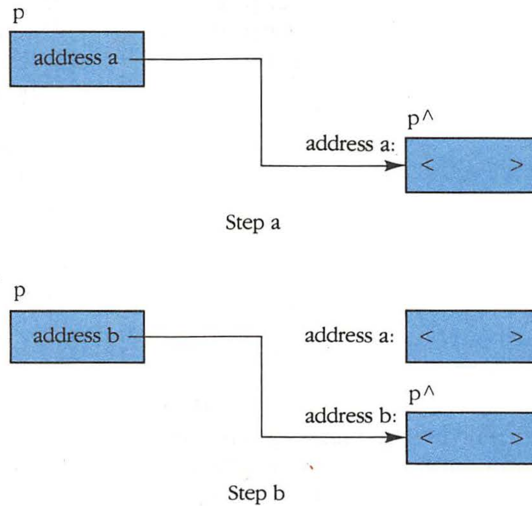


FIGURE 14.4

THE **DISPOSE** PROCEDURE

The Pascal procedure that returns a memory location to the available pool is **dispose**.

FORMAT: **dispose** (<pointer identifier>);

The **dispose** procedure takes the location that <pointer identifier> points to and releases it to the heap. A memory location must have been allocated before **dispose** is used, or an error message results. You can't dispose of something you've never had.

Example:

```
var
  p, q : ^integer;
begin
  new(q);       { Step a }
  q^ := 5;      { Step b }
  p := q;       { Step c }
  dispose(q);   { Step d }
  q := p;       { Step e }
  new(p);       { Step f }
  p^ := 2;      { Step g }
  dispose(p);   { Step h }
  dispose(q)    { Step i }
end.
```

Figure 14.5 illustrates what happens in memory when the program statements are executed. After **p** and **q** are declared to be pointers, **q** is chosen to represent a newly allocated location, and 5 is put into that location (Step b). Step c points **p** to the same location. The location pointed to by **q** is disposed of in Step d, so **q** has the value **nil.** This reference to the location pointed to by **p** is restored to **q** (Step e). The **new** in Step f pulls another location from the memory pool and stores its address in **p**. Step g puts 2 in that location. The last two statements (Steps h and i) disconnect **p** and **q** from their respective addresses. The values 5 and 2 are no longer accessible.

LINKED LISTS

Pointers can be used to form a data structure called a *linked list* that makes more efficient use of memory. This structure consists of memory locations called *nodes,* each of which contains the address of the next node in the list. Figure 14.6 illustrates the format of a node of a linked list.

The information portion of the node may contain many fields, each containing related data of varying types. Each node must also contain a field that gives the address of the next node in the list. Pointers are well suited for this next address field. The entire structure can be defined as a record.

The diagram in Figure 14.7 shows how these nodes can be connected to form a linked list.

q^ q p step a

q^ 5 q p step b

q^, p^ 5 q p step c

p^ 5 q p step d

q^, p^ 5 q p step e

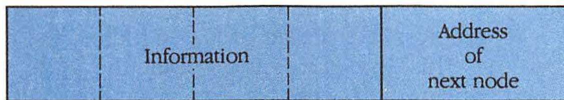
q^ 5 p^ q p step f

q^ 5 p^ 2 q p step g

q^ 5 2 q p step h

5 2 q p step i

FIGURE 14.5



node format

FIGURE 14.6

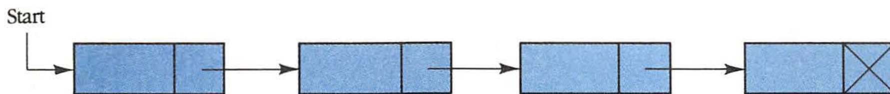


FIGURE 14.7

The pointer **start** points to the first node in the list, and the next node field of each node gives the address of the next node in line. The next node field of the last node in the list is the **nil** pointer and is represented by an *X* in the diagram.

The linked list is a dynamic data structure, because it uses exactly the number of memory locations it needs. There is no need to declare a maximum number of locations, as was done with arrays. When a node is needed, just "grab" one from the available pool, fill that node with all the pertinent information, and connect it to the current list, using the pointer in the next node field of the item it is to follow. When a node is no longer required, it can be returned to the memory pool and removed from the linked list by readjusting pointers in the next node field.

Although many operations can be performed on linked lists, it is beyond the scope of this text to explore all of these. Two of the more useful operations are discussed in the next problem: (1) inserting an item into a linked list and (2) displaying the contents of an entire list.

Problem: Write a program that creates a linked list of the names of the members of Grand Slam Bridge Club and displays the entire roster by traveling through the list.

Figure 14.8 gives a complete list and sample run of the solution.

The declaration indicates that each node is a record with two fields: the member's name (**name**) and the address of the next node (**next**). **name** is a string while **next** is a pointer that points to another node. The pointer **p** points to a node acquired from the reserve pool, while pointers **q** and **r** are used to find the end of the current list so that the new node is placed at the end of the list. The pointer **start** always points to the first node in the linked list.

After **start** is initialized to **nil** (the list is initially empty), control is transferred to **MakeRoster**, which adds nodes to the linked list. **FillNode** gets a node from the pool and fills its two fields. This newly acquired node goes at the end of the linked list, and its next field is **nil** for that reason.

If the list is empty, the new node becomes the first node in the list. If there is at least one node in the list, the **else** part of the selection structure starts a search to find the current last node. The pointer **r** takes the value of the next field of a node. In effect, it points to the next node in the list. The pointer **q** always points to the node under consideration. If **r** is **nil**, the end of the list is found, and the new node is inserted after the node pointed to by **q**.

When the entry of names is completed, the program invokes **DisplayRoster**. This procedure moves along the list by starting a pointer, in this case **p**, at the beginning of the list, displaying the contents of that node, and moving **p** to the next node by giving it the value of the next field. This display ends when **p** is **nil**.

```
program BridgeClubRoster;
{Create and display a roster for the Grand Slam Bridge Club}
{using linked lists}
 type
  nodepointer = ^nodetype;
  nodetype = record
    name : string;              {name of club member}
    next : nodepointer        {address of next node in the list}
   end;
 var
  p,              {points to newly acquired node}
  q,              {follows r down the linked list}
  r,              {points to next node in list}
  start      {points to first node in list}
   : nodepointer;
  member  {name of club member}
   : string;

 procedure FillNode;
  {Gets a new node and fills its fields}
 begin
  new(p);
  p^.name := member;
  p^.next := nil
 end;

 procedure InsertNode;
  {Puts newly acquired node in the linked list}
 begin
  if start = nil then
   begin
    FillNode;
    start := p
   end
  else
   begin
    r := start;
    q := start;
    repeat
     q := r;
     r := r^.next
    until r = nil;
    FillNode;
    q^.next := p
   end
 end;
```

FIGURE 14.8 *(continued)*

```
procedure MakeRoster;
 {Creates a roster for the club}
 var
  answer : char;
  name : string;
begin
 repeat
  write('Enter name of a member:  ');
  readln(member);
  InsertNode;
  writeln;
  write('Enter another name (y/n)?  ');
  readln(answer);
  writeln;
  writeln
 until answer = 'n'
end;

procedure DisplayRoster;

 {Displays entire roster of club}
begin
 page;
 writeln('Roster');
 writeln('------');
 p := start;
 while p <> nil do
  begin
   writeln(p^.name);
   p := p^.next
  end
 end;

begin
 start := nil;
 MakeRoster;
 DisplayRoster
end.
```

FIGURE 14.8

14.4 RECURSION

We have seen how one subprogram calls another, but Pascal allows a subprogram to call itself. This process is known as *recursion,* and a subprogram that calls itself is a *recursive subprogram.* If a subprogram is recursive, it necessarily invokes the same set of statements until a terminating condition stops the process. Although no Pascal looping structures are used, repetition takes place. You can think of recursion as a type of repetitive structure where the repetition is implied rather than explicitly stated.

Not all problems lend themselves to recursive solutions. Many problems must be solved using the nonrecursive loop structures discussed in Chapter 10. The converse is not true. If a problem has a recursive solution, the solution can also be written without using recursion.

Recursive subprograms, in general, use much more memory than nonrecursive ones do. Each time a recursive subprogram is called, the computer allocates an entirely new set of memory locations, with the same identifiers, for each of the parameters listed in the function definition. The computer must keep track of all these locations.

Why use recursion if it causes so many difficulties? Problems that have recursive solutions can allow a programmer to code difficult logic using relatively few Pascal statements.

Before writing programs using recursive subprograms, you must be able to recognize what types of problems lend themselves to recursive solutions.

THE STRUCTURE OF A RECURSIVE DEFINITION

Tasks that can be written recursively have the following format:

```
if an identifier equals a terminating value then
  the recursive subprogram value equals a constant
else
  the value of the identifier must be decreased and
    the subprogram called again
```

There must be a way for the identifier to reach the terminating value, or the calling process never ends.

Finding the factorial of an integer is a good example to illustrate what a recursive subprogram is. The factorial of an integer n is the product of all the integers from n down to 1. For example, 5 factorial is $5 \times 4 \times 3 \times 2 \times 1$ or 120. An exclamation point following a number indicates this process is to be performed. 5 factorial is then represented by 5!. By definition, 0! is 1.

Let's see how the factorial process can be written in the format shown above. Assume n is the identifier that represents the integer on which the factorial is to be found. The recursive definition of factorial then becomes

```
if n = 0 then
    n! = 1
else
    find (n − 1)!
```

$n!$ equals $n \times (n - 1) \times (n - 2) \times (n - 3) \times \ldots \times 3 \times 2 \times 1$, which is equivalent to $n \times (n - 1)!$ since $(n - 1)!$ is the product of all the integers from $n - 1$ down to 1.

Suppose you're looking for 3!. Using the recursive definition (3 is not equal to 0), let n become $(n - 1)$, find 2!, and multiply it by 3. Well, 2 is not 0; to find 2! let n become 1, find 1!, and multiply it by 2. 1 is not 0. To find 1!, decrease n by 1 to 0, find 0!, and multiply it by 1. But now n is 0, and 0! equals 1, the terminating constant value. Use 0! to find 1!, 1! to find 2!, and 2! to find 3!. The process is broken down until a constant value is found and, using that value as a base, builds back up to find the answer. A summary of this process follows:

```
3! = 3 × 2!
         2! = 2 × 1!
                  1! = 1 × 0!
                           0! = 1
                  1! = 1 × 1 or 1
         2! = 2 × 1 or 2
3! = 3 × 2 or 6
```

THE IMPLEMENTATION OF A RECURSIVE DEFINITION

Creating code to implement a recursive definition is very similar to writing code for any procedure or function. The exception is that the body of a recursive subprogram contains a call to itself.

Problem: Write a program employing a recursive function that accepts an integer and finds the factorial of that integer.

A complete program listing and sample run of the solution are given in Figure 14.9.

Since factorials grow rapidly, `answer` is declared to be of type **longint**. Even at that, 12 is the largest value for `number` that does not cause an error message.

After the number is entered, `RecursiveFactorial` is called for the first time. For example, if 3 were entered for `number`, the first call is actually `RecursiveFactorial(3)`. The value of 3 is passed to the local variable `n`. Since `n` is not 0, the else part of the selection structure is traversed. The value of `n` is decreased by 1 and becomes the contents of `numtemp`, and `RecursiveFactorial` is called, but this time with a parameter of 2. See Figure 14.10a. The right side of the `y := RecursiveFactorial(x);` statement calls `RecursiveFactorial` before `y` receives a value.

When the second call is made, the computer "creates" a new level, Level 2, and stacks the new values (2 for `n` and 1 for `numtemp`) on top of the values from the previous call (Figure 14.10b). Level 1 is not accessible unless Level 2

```
program Factorial;
{Determine the factorial of an integer}
 var
  answer          {factorial result}
   : longint;
  number          {value for factorial}
   : integer;

 function RecursiveFactorial (n : integer) : longint;
  {Calculate factorial}
  {n : value for factorial}
  var
   numtemp    {temporary location for n}
    : integer;
   anstemp    {temporary location for the answer}
    : longint;
 begin
  if n = 0 then
   RecursiveFactorial := 1
  else
   begin
    numtemp := n - 1;
    anstemp := RecursiveFactorial(numtemp);
    RecursiveFactorial := n * anstemp
   end
 end;

begin
 write('Enter a number:  ');
 readln(number);
 answer := RecursiveFactorial(number);
 writeln(number : 2, ' factorial is ', answer : 8)
end.
```

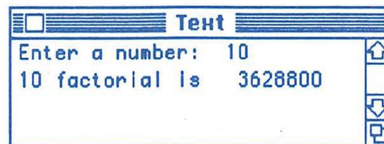


FIGURE 14.9

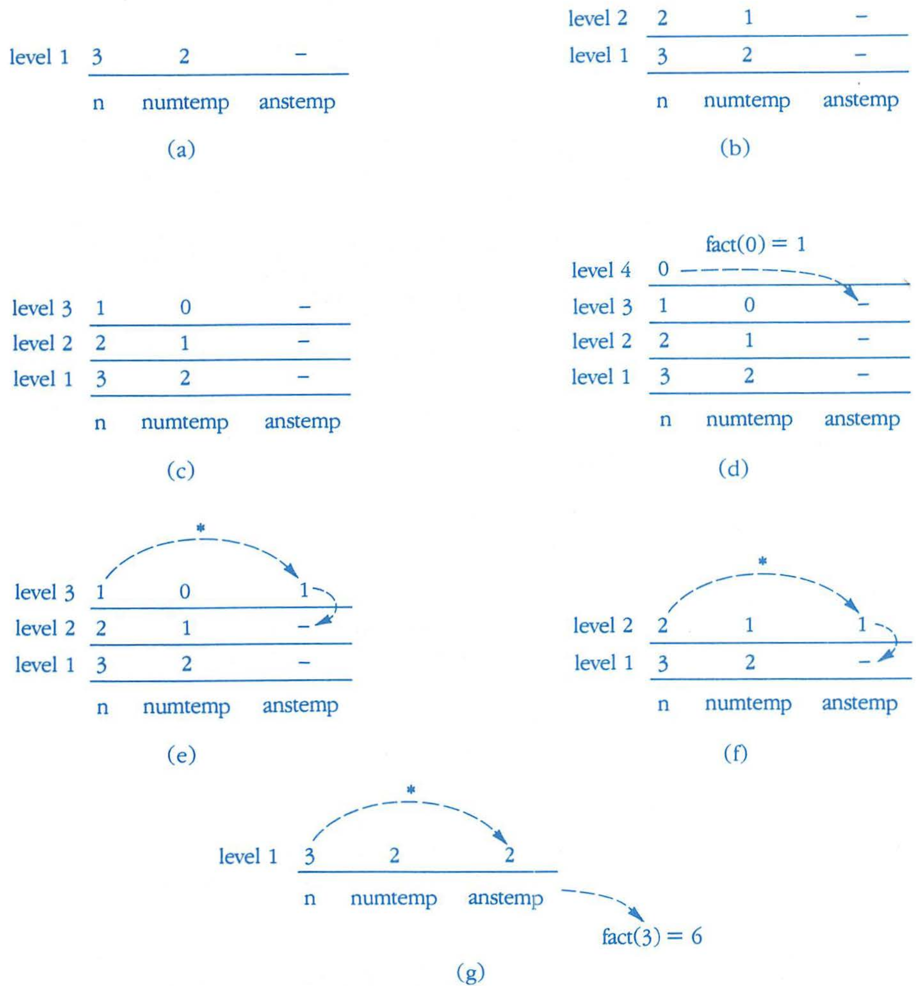


FIGURE 14.10

receives all the data it needs, specifically, a value for **anstemp**. Level 3 is formed in a similar fashion (Figure 14.10c).

When **n** attains a constant value of 0 (Figure 14.10d), the **then** part of the selection structure is executed. This constant is then returned through the function name to **anstemp** in the level that requested it (in this case Level 3) by removing all the Level 4 locations so that the Level 3 locations are accessible (Figure 14.10e).

The value of **n** from Level 3 is multiplied by the value of **anstemp** on the same level, Level 3 is removed, and the product is stored at the location represented by **anstemp** on Level 2 (Figure 14.10f). In a similar fashion, **n** from Level 2 is multiplied by **anstemp** from Level 2, the level is removed, and the product

2 is stored in anstemp on Level 1. See Figure 14.10g. The last multiplication uses n and anstemp from Level 1 and returns the final product 6 to the main program, where it is stored in the identifier represented by answer.

Although the discussion of recursion in this section is limited, it is an important feature of Pascal. Other applications appear in the exercises at the end of the chapter.

The **else** part of the selection structure in RecursiveFactorial consisted of three statements using identifiers for temporary values for the integer and its factorial value. This documentation was done to make it clearer how recursion handles memory. All three could have been combined into the single statement RecursiveFactorial := n * RecursiveFactorial (n - 1) without affecting the results.

14.5 TYPICAL PROGRAMMING ERRORS

The following are typical programming errors.

1. Not using parentheses to enclose a list of enumerated data types.

   Example:
   ```
   type
     wintermonth = Dec, Jan, Feb;
   ```
2. Using a pointer data type before specifying a location for it to point to.

   Example:
   ```
   type
     ptr = ^integer;
   var
     p : ptr;
   begin
     p^ := 8;
     ...
   ```
3. Returning a location to the memory pool before getting data from it.
4. Failing to include a terminating value in a recursive subprogram.

   Example:
   ```
   function Recur(a : integer) : real;
   begin
     b := Recur(a);
     ...
   ```

NONPROGRAMMING EXERCISES

1. Classify each of the following Pascal statements as valid or invalid. If the statement is invalid, explain why.

   a.
   ```
   type
     color = (red, white, blue);
   ```
   b.
   ```
   type
     size = (small, medium, large, xlarge);
     quantity = array[size] of integer;
   ```

c.
```
type
  size = (small, medium, large, xlarge);
  quantity = array[size] of integer;
var
  dress : quantity;
begin
  dress[small] := dress[small] + 1;
  ...
```

d.
```
type
  color = (red, white, blue);
var
  flag : color;
  cost : real;
begin
  case color of
    red, white :
      cost := 4.75;
    blue :
      cost := 5.25
end;
...
```

e.
```
type
  pointreal = real;
var
  list : ^pointreal;
```

f.
```
type
  ptr = ^char;
var
  letter : ptr;
begin
  letter^ := 'h';
  ...
```

g.
```
type
  ptr = ^char;
var
  letter : ptr;
begin
  letter := 'h';
  ...
```

h.
```
type
  ptr = ^integer;
var
  p : ptr;
  n : integer;
begin
  n := 25;
  new(p);
  p := n;
  ...
```

2. Show the exact output for each of the following Pascal programs.

a.
```
program One;
type
 intpointer = ^integer;
var
 r, s : intpointer;
begin
 new(r);
 r^ := 10;
 writeln(r^:6);
 s := r;
 writeln(r^:6, s^:6);
 new(r);
 r^ := 5;
 writeln(r^:6, s^:6);
 dispose(s);
 writeln(r^:6);
 dispose(r)
end.
```

b.
```
program Two;
type
 intpointer = ^integer;
var
 r, s : intpointer;
begin
 new(r);
 r^ := 10;
 writeln(r^:6);
 new(r);
 r^ := 5;
 writeln(r^:6);
 s := r;
 writeln(r^:6, s^:6);
 s^ := 8;
 writeln(r^:6, s^:6);
 dispose(s);
 writeln(r^:6);
 dispose(r)
end.
```

c.
```
program Three;
 var
  f, s : integer;
 procedure P(x, y : integer);
 begin
  if x <> y then
   begin
    writeln(x:6, y:6);
    if x < y then
     P(x, y-x)
    else
     P(x-y, y)
   end
```

```
    else
     writeln('Answer is ',x:2)
   end;
  begin
   f := 108;
   s := 42;
   P(f, s)
  end.
```

d.
```
program Four;
 var
  f, s : integer;
 procedure P(x, y : integer);
 begin
  if x <> y then
   begin
    writeln(x:6, y:6);
    if x < y then
     P(x, y-x)
    else
     P(x-y, y)
   end
  else
   writeln('Answer is ',x:2)
 end;
begin
 f := 7;
 s := 1;
 P(f, s)
end.
```

PROGRAMMING EXERCISES

(B) 3. Write a program that defines an enumerated data type called *icecream* that has three values: *vanilla, chocolate,* and *strawberry.* Use this data type as the index of an array that keeps a running total of the number of half-gallons of each of the three flavors of ice cream sold. The flavor and number of half-gallons of that flavor sold should be entered at the keyboard. Display, in table form, a summary of the number of each flavor sold.

(B) 4. Write a program for a dentist that defines an enumerated data type called *charge* that has three values: *clean, cavity, extract.* Use this data type as the selector value in a **case** structure that determines the cost for each of the specified services based on the following table:

SERVICE	*UNIT CHARGE*
Clean	$35
Cavity	$30
Extract	$50

The program accepts the patient's name, the service, and how many times that service was performed and displays a bill for the patient.

(B) 5. Write a program that determines the cost of airing a number of commercials on the three major networks: ABC, CBS, and NBC. The networks' initials should serve as the values for an enumerated data type and the cost per 30-second commercial stored and accessed through these data types. After a user enters the number of 30-second commercials desired on each of the three networks, display the total cost.

(B) 6. Write a program that determines the cost of catering a wedding. Use an enumerated data type with values *appetizer, entree,* and *dessert.* After the number of guests is entered, a menu of possible choices and prices for each of the three categories is displayed in order, and the user is asked to choose one or many of the options in each of the three menus. The program then displays the chosen meal with the total cost.

(B) 7. Write a program to create a linked list of inventory records, each containing the identification number of an item, the number of items currently in stock, and the unit price of the item. Suppose the unit price must be increased by 7.5 percent. Update and display all the changed records.

(B) 8. Firmware City is conducting a contest. The winner must guess the exact number of integrated circuit chips contained in a coffee cup in the store. If nobody guesses correctly, the prize goes to charity. Write a program using linked lists to store a contestant's name and guess. Input the winning number and output an appropriate message to the winner or the charity.

(B) 9. Gumbel's Department Store maintains inventory control by storing sales and inventory information in a linked list. The following is an example:

ITEM NO.	*NO. SOLD*	*COST PRICE*	*SALE PRICE*	*PROFIT*
458	0	$17.50	$23.99	0
217	0	9.95	14.95	0
874	0	11.75	13.95	0
121	0	21.00	29.99	0
553	0	15.00	21.75	0
348	0	13.50	17.95	0

Write a program that creates a linked list to store the information shown above and updates the list by inputting the number of items sold in Field 2 and computing the profit in Field 5. Output the entire filled list in tabular form with appropriate headings.

(G) 10. Write a program to assist Professor Ima Sue Smart in processing her grades. Assume that her class has 10 students and each has taken three exams. Enter the following data in a linked list.

NAME	*EXAM 1*	*EXAM 2*	*EXAM 3*
Jones	75	83	78
Wallace	91	88	75
Gomez	82	88	80
Smith	90	72	78
Caruso	77	84	68
Black	81	65	87
Kojak	68	93	85
Chen	73	82	70
Goldberg	86	71	73
O'Grady	62	76	85

Calculate and print the class average score for each exam. The program should also be capable of searching the list for an input name and calculating the corresponding average of the three exam scores for that student.

(G) 11. Change Exercise 10 to include a module that deletes the record corresponding to a student name entered by the user.

(G) 12. Statistics for a baseball team are listed here in array form:

PLAYER	*AT BATS*	*NO. HITS*	*NO. HOMERS*	*AVERAGE*
Mitts	258	76	10	
Wall	328	108	8	
Cage	311	97	5	
Whiffo	291	101	9	
Mound	287	88	12	
Batts	307	95	3	
Aoute	318	107	7	
Runner	270	83	0	
Bunter	321	103	10	

Write a program that loads a linked list with the information given above, computes the batting average for each player by dividing the number of hits by the number of "at bats," and stores the results in a field of the node record. Averages should be rounded to three decimal places before being stored. Output the entire list, appropriately labeled.

(G) 13. Write a program that selects random integers and stores these integers in a linked list. After the list has been created, go through the entire list and find and display the average of the random integers in the list.

(G) 14. While the list in Exercise 13 is being created, show a graphic representation of the list similar to the one shown in Figure 14.7.

(G) 15. Write a program that creates a linked list of records with fields consisting of a person's name, Social Security number, and year of birth. Search through this master list, and create a second linked list consisting of eligible voters (all those who are 21 years of age or older). Display the contents of the second list.

(B) 16. The Discount Furniture Outlet would like to have a bar graph showing what percentage of yearly sales occurred in each month for the year 1985. Write a program that creates a linked list of records with fields consisting of the name of a month, the total sales in that month, and the percentage of the yearly sales that month represented. This last field is empty until the data for the entire year are entered. Go through the list and find the total sales for the year, calculate the desired percentage, and store it in the third field of each record. Draw a bar graph displaying the percentages for each month.

(G) 17. Write a program that uses a recursive function called *Product* to multiply two numbers x and y. The product can be determined by the following recursive definition of multiplication:

```
if y = 1 then
  the product equals x
else
  find the product of x and (y - 1) and add it to x
```

(G) 18. Write a program that uses a recursive function *Fibonacci* to find the terms in a Fibonacci sequence. The first ten terms are 1, 1, 2, 3, 5, 8, 13, 21, 34, 55. Each successive term is determined by adding the two terms immediately preceding it. For example, the 11th term (89) is found by adding the 9th and 10th terms, 34 and 55, respectively. The *n*th term in this sequence can be found using the following recursive definition:

if n is 1 or 2 then
  Fibonacci equals 1
else
  Fibonacci equals the sum of the $(n - 1)$st and $(n - 2)$nd terms

(G) 19. Write a program using a recursive subprogram that calculates the sum of the elements of an array.

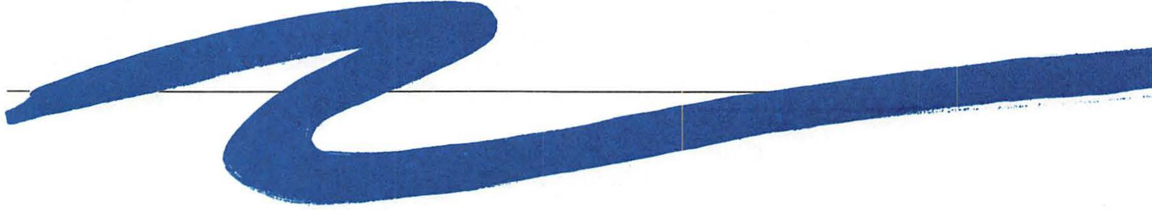
(G) 20. Write a program using a recursive subprogram that calculates the product of the elements of an array.

(B) 21. The search algorithm discussed in Chapter 11 was a linear search. Given a list of items, each item in the list was tested, starting with the first element in the array and ending with the last one. This process can be time consuming, especially for a large list. A more efficient method for searching an ordered list of array elements is the binary search. This search takes the target value and compares it to the middle element of the array. If the middle term equals the target value, the search ends, because the correct array element has been found. If it is larger than the middle, the term following the middle term becomes the new low term in the search and the "upper" half of the list is searched using the same method until a match is found. If the target value is less than the middle element, the term preceding the middle term becomes the high element in the search and the "lower" half of the list is searched using the same method until a match is found. Since the same searching algorithm is employed on a smaller and smaller portion of the list, the solution can be recursive.

Write a program that fills an array of customer account numbers, names, and addresses. Sort the list in ascending order by account number. Use a recursive subprogram that implements a binary search to find the name and address of a customer given the account number.

(B) 22. Repeat Exercise 21 with the customer records stored in a linked list rather than an array of records.

APPENDIX C



ASCII Character Codes

DEC CODE	CHAR	DEC CODE	CHAR	DEC CODE	CHAR	DEC CODE	CHAR
0	NUL	32	SP	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	`	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

APPENDIX D

Reserved Words

and	file	of	then
array	for	or	to
begin	function	otherwise	type
case	goto	packed	until
const	if	procedure	uses
div	in	program	var
do	label	record	while
downto	mod	repeat	with
else	nil	set	
end	not	string	

APPENDIX E

Macintosh Pascal Instruction Summary

```
INSTRUCTION                                              PAGE
begin                                                    (page 63)

case <var expr> of
  <option 1> :
    <statement a>;
  <option 2> :
    <statement b>;
  ...
  otherwise
    <default statement>
end;                                                     (page 215)

const
  <identifier> = <constant>;                             (page 59)

end                                                      (page 63)

for <identifier> := <expr1> to <expr2> do
  <statement>;                                           (page 233)

function <identifier>( <parameter list> ) : <fct type>;
  <local identifier declarations>
begin
  <body of function>
end;                                                     (page 166)
```

INSTRUCTION	PAGE

if <condition> then
<statement a>
else
<statement b>;
	(page 206)

procedure <identifier>(<parameter list>);
<local identifier declarations>
begin
<body of procedure>
end;
	(page 168)

program <program identifier>;	(page 63)
repeat <statement> until <condition>;	(page 238)
type <type identifier> = <definition>;	(page 61)
var <identifier list> : <type>;	(page 60)
while <condition> do <statement>;	(page 236)

Macintosh Pascal Function and Procedure Summary

`abs ( <real or integer identifier or value> )`
Returns the absolute value of its argument. (page 129)

`close( <file identifier> );`
Deactivates the file specified by its parameter. (page 77)

`concat( <string>, <string> , ... )`
Returns the concatenation of its string arguments. (page 138)

`copy( <source string> , <index> , <count> )`
Returns a copy of a substring of a given string. (page 139)

`delete( <string identifier> , <index> , <count> );`
Deletes a substring from a given string. (page 140)

`dispose( <pointer identifier> );`
Eliminates reference to a dynamic variable. (page 390)

`drawline( <horiz1> , <vert1>, <horiz2>, <vert2> );`
Draws a straight line between the points specified in its parameters. (page 147)

`eof( <file identifier> )`
Returns a Boolean value that indicates whether a file pointer is beyond the last record in a file. (page 368)

`filepos( <file identifier> )`
Returns the record number of the active record in a file. (page 366)

`framerect( <top> , <left> , <bottom> , <right> );`
Draws a framed rectangle with the boundaries specified by its parameters. (page 148)

`insert( <source> , <destination> , <index> );`
Inserts a substring into a given string. (page 140)

`invertcircle( <xcenter> , <ycenter> , <radius> );`
Inverts the pixels in a circle with center coordinates and radius given by its parameters. (page 151)

`invertrect( <top> , <left> , <bottom> , <right> );`
Inverts the pixels in a rectangle with boundaries specified by its parameters. (page 150)

`length( <string identifier or value> )`
Returns the length of its string argument. (page 137)

`lineto( <horiz coord> , <vert coord> );`
Draws a line from the last pixel illuminated to the point with coordinates specified by its parameters. (page 147)

`moveto( <horiz coord> , <vert coord> );`
Moves the drawing pen location to the coordinates specified by its parameters. (page 75)

`new( <pointer identifier> );`
Creates a new dynamic variable and sets a pointer variable to it. (page 388)

`odd( <integer identifier or value> )`
Returns a Boolean value indicating whether its argument is odd. (page 130)

`open( <file identifier> , '<file name>' );`
Permits access to the file specified by its parameters. (page 76)

`paintcircle( <xcenter> , <ycenter> , <radius> );`
Draws a solid circle with center coordinates and radius specified by its parameters. (page 151)

`paintrect( <top> , <left> , <bottom> , <right> );`
Draws a solid rectangle with boundaries specified by its parameters. (page 149)

`pos( <substring> , <string> )`
Returns the starting position of a substring in a given string. (page 138)

`random`
Returns a random integer between −32768 and 32767, inclusive. (page 131)

`read( <identifier> );`
Transfers a value from the keyboard into a main memory location specified by its parameter; termination of input is indicated by a space. (page 79)

`read( <file identifier> , <identifier> );`
Transfers values from a file specified in an open statement into main memory locations specified by its parameters. (page 81)

`readln( <identifier> );`
Transfers values from the keyboard into a main memory location specified by its parameter where termination of input is indicated by a Return. (page 79)

`reset( <file identifier> );`
Sets a file pointer at the beginning of an open file for reading. (page 369)

`rewrite( <file identifier> , 'printer:' );`
Associates an internal file with the printer for output. (page 78)

`round( <real identifier or value> );`
Returns the rounded value of its argument. (page 131)

`seek( <file identifier>, <file position> );`
Places a file pointer at the record represented by a file position. (page 370)

`sqr( <real or integer identifier or value> )`
Returns the square of its argument. (page 129)

`sqrt(<nonnegative real/integer identifier or value>)`
Returns the square root of its argument. (page 130)

`trunc( <real identifier or value> )`
Returns the integer part of its real argument. (page 131)

`write( <identifier or data list> );`
Outputs the data specified by its parameters to the Text window and suppresses the line feed. (page 69)

`write( <file identifier> , <data list> );`
Outputs the data specified by its parameters to a data file. (page 77)

`writedraw( <identifier or data list> );`
Outputs text specified by its parameters to the Drawing window. (page 75)

`writeln( <identifier or data list> );`
Outputs the data specified by its parameters to the Text window and activates the line feed. (page 69)

`writeln( <file identifier> , <data list> );`
Outputs the data specified by its parameters to an external file or device and activates the line feed. (page 78)

APPENDIX G

Answers to Selected Exercises

CHAPTER 1

1. Press and hold down the mouse button while on **View** in the menu header, move the screen arrow down the menu until **by Size** inverses, and release the mouse button.
3. Press and hold down the mouse button while the screen arrow is pointing to **File.**
5. Press and hold down the mouse button while on **View** in the menu header, move the screen arrow down the menu until **by Icon** inverses, and release the mouse button.
7. Press and hold down the mouse button while the screen arrow is pointing to the interlocking boxes in the lower right corner of the window, adjust the size by moving the mouse, and release the button when the desired size is reached.
9. Press and hold down the mouse button while the screen arrow is on **File,** move the screen arrow down the menu until the desired choice inverses, and release the mouse button.

CHAPTER 2

1. Choose **Go** from the **Run** menu.
3. Hold down the mouse button while the screen arrow points to **File** in the menu header.
5. Select **Quit** from the **File** menu and return to the system screen. Choose **Eject** from the system screen's **File** menu.

7. Select **Open...** from the **File** menu. Scroll down the file listings that appear in the dialog box by using the up and down arrows to the right of the file names.
9. Choose **Save As...** from the **File** menu. When the dialog box appears, type a new name in the designated area and click the mouse button while the screen arrow is on **Save.**
11. Hold down the mouse button while the screen arrow is on the name of the window, move the window to the desired location, and release the mouse button.
13. Select **Save** from the **File** menu.
15. Fill in the **What to find** box by selecting that choice from the **Search** menu; then select **Everywhere** from the same menu.

CHAPTER 3

1.

Step 2: IPO chart

Type	Identifier	Description	
INPUT	EXAM1 EXAM2 EXAM3	grade on Exam 1 grade on Exam 2 grade on Exam 3	
PROCESSING	AVER FINAL	average of first 3 exam grades final grade	
OUTPUT	EXAM1 EXAM2	EXAM3 EXAM4	FINAL

Step 3: See Examples 2 and 3 in Chapter 3.

Step 4: N-S chart. See Figure G.1.

Step 6: AVER = (EXAM1 + EXAM2 + EXAM3)/3
= (87 + 73 + 80)/3 = 80
FINAL = .60 * AVER + .40 * EXAM4
= .60 * 80 + .40 * 82
= 48 + 32.8 = 80.8

Enter exam grades
Find AVER
Find FINAL
Output results

FIGURE G.1

4.

Step 2: IPO chart

Type	Identifier	Description	
INPUT	ITEM1	price of item 1	
	ITEM2	price of item 2	
	ITEM3	price of item 3	
PROCESSING	TOTAL	total of items before discount	
	DISAMT	total amount of discount	
	FINAL	final discounted price	
OUTPUT	ITEM1	ITEM3	FINAL
	ITEM2	DISAMT	

Step 3: See Examples 1 and 3 in Chapter 3.

Step 4: N-S chart. See Figure G.2.

Step 6: TOTAL = ITEM1 + ITEM2 + ITEM3
= 1.29 + 5.43 + 10.22 = 16.94
DISAMT = .15 * TOTAL
= .15 * 16.94 = 2.541
FINAL = TOTAL − DISAMT
= 16.94 − 2.541 = 14.399

Enter item prices
Find DISAMT
Find FINAL
Output results

FIGURE G.2

9.

Step 2: IPO chart

Type	Identifier	Description	
INPUT	SIDE	length of side of square	
	RADIUS	radius of circle	
PROCESSING	AREASQ	area of square	
	AREACR	area of circle	
	DIFF	difference of areas	
OUTPUT	SIDE	AREASQ	DIFF
	RADIUS	AREACR	

Step 3. No similar examples in Chapter 3.

Step 4: N-S chart. See Figure G.3.

Step 6: AREASQ = SIDE ^ 2
= 10 ^ 2 = 100
AREACR = 3.14 * RADIUS ^ 2
= 3.14 * 5 ^ 2 = 78.5
DIFF = AREASQ − AREACR
= 100 − 78.5 = 21.5

Enter SIDE, RADIUS
Find AREASQ
Find AREACR
Find DIFF
Output results

FIGURE G.3

CHAPTER 4

1. a. Boolean
   c. real
   e. string
   g. integer
   i. Boolean
   k. string
   m. string

2. a.
```
var
  test : Boolean;
```
   c.
```
var
  small : real;
```
   f.
```
var
  x, y : integer;
```
   h.
```
var
  s : char;
```
   j.
```
var
  huge : double;
```
   l.
```
const
  answ = 'y';
```
   o.
```
var
  logic : Boolean;
```

3. a. Invalid; a semicolon should follow the statement.
   c. Invalid; the keyword is `end`, not `theend`.
   e. Valid.
   g. Invalid; cannot leave a space between * and) .
   i. Valid.

CHAPTER 5

1. a. Invalid; the closing quote is missing.
   c. Invalid; the proper instruction is **writedraw**.
   e. Valid, if **g** is the internal file name for output to a printer.
   g. Invalid; **moveto** requires a pair of coordinates as arguments.
   i. Valid.
2. a. `Happy`
   `Birthday!`
   c. `Happy`
   `Birthday!`
   e. `HappyBirthday!`
3. a. No **close** statement.
   c. The proper keyword is **writedraw**, not **draw**.
   The semicolon following the **draw** statement is unnecessary, but program will run.
4. a.

a	*b*	*c*
5	−45	−34.57

7. See Figure G.4.

```
program Corners;
{Displays at the four corners of the Drawing Window}
begin
 moveto(0, 8);
 writedraw('corner');
 moveto(160, 8);
 writedraw('corner');
 moveto(160, 200);
 writedraw('corner');
 moveto(0, 200);
 writedraw('corner')
end.
```

FIGURE G.4

9. See Figure G.5.
17. See Figure G.6.
20. See Figure G.7.

```
program MailingLabel;
{Displays a company mailing address on printer}
 var
  ptr : text;
begin
 rewrite(ptr, 'printer:');
 writeln(ptr, 'Johnson'' s Realty Corporation ');
 writeln(ptr, '123 New England Drive');
 writeln(ptr, 'South Fork, MA  01234')
end.
```

FIGURE G.5

```
program CustomerLetter;
{Creates and prints from a file}
 var
  c, d : file of string;
  ptr : text;
  customer, name : string;
  item, article : string;
begin
 open(c, 'Purchases');
 write('Enter name of customer : ');
 readln(customer);
 write('Enter item purchased : ');
 readln(item);
 write(c, customer, item);
 close(c);
 open(d, 'Purchases');
 read(d, name, article);
 rewrite(ptr, 'printer:');
 writeln(ptr, ' Dear ', name, ' , ');
 writeln(ptr);
 writeln(ptr, 'Congratulations on having purchased the finest ');
 writeln(ptr, article, ' available.  It is our pleasure to serve you');
 writeln(ptr, 'in this capacity.  Please call upon us again.');
 writeln(ptr, '                  Sincerely yours,');
 writeln(ptr, '                  Specialty Store Inc.');
 close(d)
end.
```

FIGURE G.6

```
program TreeDesign;
{Drawing a design in the Text window}
begin
 writeln('*' : 20);
 writeln('***' : 21);
 writeln('*****' : 22);
 writeln('*******' : 23)
end.
```

FIGURE G.7

CHAPTER 6

1. a. `b * b - 4 * a * c`
   c. `(x - y) / (x + y)`
   e. `p * (1 + i)`
2. a. xy^3
   c. $\frac{m}{n}a$
   e. $d - \frac{e}{j} + k$
   g. $\frac{d - e}{j + k}$
3. a. 11
   c. 4
   e. 1
   g. 1
   i. 3
4. a. Valid.
   c. Valid.
   e. Invalid; cannot store a real result in an integer identifier.
6. *c* has the following values, in order:
   0 14 14 14 2 2 2 2
7. Enter first test score: 70
   Enter second test score: 90
   Enter final exam score: 84
   Course grade is: 81.60
8. See Figure G.8.
12. See Figure G.9.
21. See Figure G.10.
22. See Figure G.11.

```
program Numbers;
{Enter two values and output their sum, difference, product and  quotient}
 var
  value1, value2 : real;
  sum, diff, prod, quot : real;
begin
 writeln('Type in two non-zero, real values.');
 writeln;
 write('Enter first number : ');
 readln(value1);
 write('Enter second number : ');
 readln(value2);
 writeln;
 sum := value1 + value2;
 diff := value1 - value2;
 prod := value1 * value2;
 quot := value1 / value2;
 writeln('SUM' : 10, 'DIFF' : 10, 'PRODUCT' : 10, 'QUOTIENT' : 10);
 writeln('________________________________________');
 writeln(sum : 10 : 2, diff : 10 : 2, prod : 10 : 2, quot : 10 : 2)
end.
```

FIGURE G.8

```
program Dimensions;
{Calculates area in room given length and width dimensions}
 var
  length, width, area : real;
begin
 write('Enter the length of the room in feet : ');
 readln(length);
 write('Enter the width of the room in feet: ');
 readln(width);
 area := length * width;
 writeln;
 writeln('The floor area of this room is ', area : 6 : 1, ' sq. ft.')
end.
```

FIGURE G.9

```
program FileofNames;
{Creates and displays a file of names}
 var
  names, data : file of string;
  n1, n2, n3, a : string;
begin
 open(names, 'List');
 write('Enter first name : ');
 readln(n1);
 write(names, n1);
 write('Enter second name : ');
 readln(n2);
 write(names, n2);
 write('Enter third name : ');
 readln(n3);
 write(names, n3);
 close(names);
 writeln;
 writeln('The names stored in the file are:');
 open(data, 'List');
 read(data, a);
 writeln(a);
 read(data, a);
 writeln(a);
 read(data, a);
 writeln(a);
 close(data)
end.
```

FIGURE G.10

```
program CampaignProfit;
{Calculates profit from selling campaign buttons}
 const
  costbut = 0.27;
  sellpr = 1.00;
  noofbut = 100;
 var
  expenses : real;
  gross : real;
  net : real;
begin
 expenses := costbut * noofbut;
 gross := sellpr * noofbut;
 net := gross - expenses;
 writeln('Cost per button : $', costbut : 4 : 2);
 writeln('Selling price : $', sellpr : 5 : 2);
 writeln('Number of buttons sold : ', noofbut : 4);
 writeln('Gross income:$ ', gross : 6 : 2);
 writeln('Expenses : $', expenses : 6 : 2);
 writeln('Net profit : $ ', net : 5 : 2)
end.
```

FIGURE G.11

CHAPTER 7

1. a. Invalid; a single identifier must appear on the left side of the assignment operator.
   c. Invalid; **paintcircle** is a procedure; its call stands as a complete statement.
   e. Valid.
   g. Valid.
   i. Invalid; **invertcircle** requires three arguments.
2. a. 3
   c. yourself
   e. true
   g. ax
   i. false
3. a. Deletes `re` from `representation`, giving `presentation`.
   c. Moves the drawing pen in the **Drawing** window to position (80,90).
   e. Draws a line from (10,20) to (30,40).
   g. Draws a shaded rectangle whose top boundary is 100, left boundary is 35, bottom boundary is 121, and right boundary is 136.
   i. Draws a shaded circle with center at (50,50) and radius 25.
4. a. `insert('ist', 'make', 2);`
   c. `x := round (100*x)/(100); mod 37;`
   e. `z := sqrt(x) - sqrt(y);`
   g. `b := odd(round(x));`
   i. `x := pos('m', 'abcdefghijklmnopqrstuvwxyz');`
   k. `str := copy(test, 3, 3);`
5. See Figure G.12.
9. See Figure G.13.
18. See Figure G.14.
20. See Figure G.15.

```
program FederalTax;
{Determines federal tax on gross pay}
 const
  rate = 0.37;
 var
  grosspay, tax : real;
begin
 write('Enter gross pay:  $');
 readln(grosspay);
 tax := rate * grosspay;
 tax := round(100 * tax) / 100;
 writeln('Federal tax is $', tax : 6 : 2)
end.
```

FIGURE G.12

```
program DiceRoll;
{Simulates the roll of a pair of dice}
 var
  die1, die2, sum : integer;
begin
 writeln('The dice are rolled... ');
 die1 := random mod 6 + 1;
 writeln('The first one is a ', die1 : 1);
 die2 := random mod 6 + 1;
 writeln('The second one is a ', die2 : 1);
 sum := die1 + die2;
 writeln('The sum is ', sum : 2)
end.
```

FIGURE G.13

```
program BankPassword;
{Creates a code for a money machine}
 var
  first, last, ssn, firstpart, secondpart, thirdpart, code : string;
  size : integer;
begin
 write('Enter user''s last name:  ');
 readln(last);
 write('Enter user''s first name:  ');
 readln(first);
 write('Enter user''s social security number:  ');
 readln(ssn);
 firstpart := copy(last, 1, 1);
 size := length(first);
 secondpart := copy(first, size, 1);
 thirdpart := copy(ssn, 4, 3);
 code := concat(firstpart, secondpart, thirdpart);
 writeln('The bank password is ', code)
end.
```

FIGURE G.14

```
program CheckeredSquare;
{Draw a checkered square}
begin
 framerect(20, 20, 100, 100);
 paintrect(20, 100, 100, 180);
 paintrect(100, 20, 180, 100);
 framerect(100, 100, 180, 180)
end.
```

FIGURE G.15

CHAPTER 8

1. a. Invalid; the function is not given a type.
   c. Valid.
   e. Valid.
   g. Invalid; **var** is not used for a procedure name.
2. a. a in Function C equals 12
   a in the main program equals 12
   b in the main program equals 12
   c. b in Function A equals 14
   g in Procedure D equals 14
   y in the main program equals 14
3. a.

```
function Product(x, y : real) : real;
 begin
  Product := x * y
 end;
```

   c.

```
function Choose : integer;
 begin
  Choose := random mod 3 + 1
 end;
```

   f.

```
procedure HalfAndHalf(top,left,bottom,right:integer);
 var
  half : integer;
 begin
  half := (right - left) div 2;
  paintrect(top, left, bottom, half);
  framerect(top, half, bottom, right)
 end;
```

   h.

```
procedure Ring(x, y, r : integer);
 var
  newradius : integer;
 begin
  paintcircle(x, y, r);
  newradius := div 2;
  invertcircle(x, y, newradius)
 end;
```

6. See Figure G.16.
11. See Figure G.17.
21. See Figure G.18.
23. See Figure G.19.

```
program CricketThermometer;
{Find the temperature using cricket chirps}
 var
  number : integer;
  temperature : real;

 procedure EnterData (var count : integer);
 begin
  write('Enter the number of chirps per minute:  ');
  readln(count)
 end;

 function FahrenheitTemp (count : integer) : real;
 begin
  FahrenheitTemp := (count + 160) / 4
 end;

 procedure Display (count : integer;
          temp : real);
 begin
  writeln;
  writeln(count : 3, ' cricket chirps per minute ');
  writeln('means the Fahrenheit temperature is ', temp : 6 : 2)
 end;

begin
 EnterData(number);
 temperature := FahrenheitTemp(number);
 Display(number, temperature)
end.
```

FIGURE G.16

```
program TakeHomePay;
{Computes take home pay for various pay periods}
 const
  monthly = 12;
  biweekly = 26;
  weekly = 52;
  taxrate = 0.15;
 var
  annualsalary, netsalary : real;

 function NetPay (grosssalary : real;
          periods : integer) : real;
 begin
  NetPay := grosssalary / periods * (1 - taxrate)
 end;
```

FIGURE G.17 *(continued)*

```
begin
 write('Enter gross annual salary:  $');
 readln(annualsalary);
 netsalary := NetPay(annualsalary, monthly);
 writeln('Net pay on a monthly basis:  $', netsalary : 6 : 2);
 netsalary := NetPay(annualsalary, biweekly);
 writeln('Net pay on a biweekly basis:  $', netsalary : 6 : 2);
 netsalary := NetPay(annualsalary, weekly);
 writeln('Net pay on a weekly basis:  $', netsalary : 6 : 2)
end.
```

FIGURE G.17

```
program MailOrders;
{Processes mail orders for forks}
 const
  shipping = 0.02;
  unitcost = 0.79;
 var
  quantity : integer;
  totalcost, subtotal, postage : real;
  customer, custstreet, custcity : string;

 procedure EnterData (var number : integer;
         var name, staddress, citystzip : string);
 begin
  write('Enter the name of the customer:  ');
  readln(name);
  write('Enter the street address of the customer:  ');
  readln(staddress);
  write('Enter the city, state and zip code of the customer:  ');
  readln(citystzip);
  write('Enter number of forks desired:  ');
  readln(number)
 end;

 procedure Costs (number : integer;
         var mdsecost, total, shipcost : real);
 begin
  mdsecost := unitcost * number;
  shipcost := shipping * mdsecost;
  total := mdsecost + shipcost
 end;
```

FIGURE G.18 *(continued)*

```
 procedure Invoice;
 begin
  page;
  writeln('The Mail Order Palace');
  writeln;
  writeln('Ship to:');
  writeln(customer : 30);
  writeln(custstreet : 30);
  writeln(custcity : 30);
  writeln;
  writeln('Qty' : 5, 'Subtotal' : 30);
  writeln(quantity : 4, subtotal : 30 : 2);
  writeln('Shipping: ' : 20, postage : 15 : 2);
  writeln('Total: ' : 20, totalcost : 15 : 2);
  writeln;
  writeln('Thank you for shopping with The Mail Order Palace.')
 end;

begin
 EnterData(quantity, customer, custstreet, custcity);
 Costs(quantity, subtotal, totalcost, postage);
 Invoice
end.
```

FIGURE G.18

```
program CircleSize;
{Draws circle, changes size, and redraws circle}
 const
  xcenter = 100;
  ycenter = 100;
 var
  firstradius, secondradius : integer;

 function NewRadius (radius : integer) : integer;
  var
   factor : real;
 begin
  write('Enter size factor:  ');
  readln(factor);
  NewRadius := round(radius * factor)
 end;

begin
 write('Enter radius of circle:  ');
 readln(firstradius);
 paintcircle(xcenter, ycenter, firstradius);
 secondradius := NewRadius(firstradius);
 invertcircle(xcenter, ycenter, firstradius);
 paintcircle(xcenter, ycenter, secondradius)
end.
```

FIGURE G.19

CHAPTER 9

1. a. Valid
   c. Valid
   e. Valid
   g. Valid
2. a. 9
   c. third option
   e. second class
3. a. false
   c. true
   e. true
   g. false
4. a.
```
if x = 4 then
  writeln('hello');
```
   c.
```
if t > 0 then
  x := sqrt(t)
else
  if t < 0 then
    x := sqrt(t)
  else
    writeln('neither');
```
   e.
```
case c of
  'x' :
    writeln('no good');
  'y' :
    writeln('may be good');
  'z' :
    writeln('always good');
  otherwise
    writeln('bad')
end;
```
7. See Figure G.20.
10. See Figure G.21.
16. See Figure G.22.
18. See Figure G.23.

```
program BattingAverage;
{Calculate batting average based on number of at bats and hits}
 var
  noatbats, nohits : integer;
  batavg : integer;
begin
 writeln('      Bush League Batting Average');
 writeln;
 write('   Enter number of at bats :');
 readln(noatbats);
 write('   Enter number of hits :');
 readln(nohits);
 batavg := round(nohits / noatbats * 1000);
 writeln('      Batting average : ', batavg : 4);
 if batavg >= 300 then
  writeln('              Heavy hitter!')
end.
```

FIGURE G.20

```
program CallMe;
{Calculates the charge for a telephone call}
 const
  charge1 = 0.20;
  charge2 = 0.07;
 var
  caller : string;
  time : real;
  cost : real;
begin
 writeln('     Ding Dong Bell Telephone Company');
 writeln;
 write(' Enter name of caller :');
 readln(caller);
 write(' Enter length of call :');
 readln(time);
 if time > 5.0 then
  cost := charge1 + charge2 * (time - 5.0)
 else
  cost := charge1;
 writeln;
 writeln('Name of caller : ', caller);
 writeln('Cost of call : $', cost : 6 : 2)
end.
```

FIGURE G.21

```
program GetADiscount;
{Determines net cost of purchase after discount deductions}
 const
  level1 = 3000.0;
  level2 = 2000.0;
  level3 = 1500.0;
 var
  name : string;
  acctno : integer;
  purchase : real;
  netcost : real;
  disct : real;
begin
 writeln('   Orange Computer Outlet');
 writeln;
 write('  Enter customer name: ');
 readln(name);
 write('  Enter account number : ');
 readln(acctno);
 write('  Enter purchased amount :$');
 readln(purchase);
 if purchase > level1 then
  disct := 0.20;
 if (purchase > level2) and (purchase <= level1) then
  disct := 0.15;
 if (purchase > level3) and (purchase <= level2) then
  disct := 0.10;
 if purchase <= level3 then
  disct := 0;
 netcost := purchase - disct * purchase;
 writeln;
 writeln('Name of customer -->', name);
 writeln('Account number-->', acctno);
 writeln('Net cost after discount--> $', netcost : 8 : 2)
end.
```

FIGURE G.22

```
program MizerMufflers;
{Determines cost of muffler installation according to selection}
 const
  base = 39.95;
 var
  customer : string;
  code : char;
  cost : real;
  factor : real;
```

FIGURE G.23 *(continued)*

```
begin
 writeln('       The Mizer Muffler Company');
 writeln;
 write('  Enter customer name : ');
 readln(customer);
 write('  Enter code A,B,C,D,E, or F : ');
 readln(code);
 case code of
  'A' :
   factor := 1.00;
  'B' :
   factor := 1.04;
  'C' :
   factor := 1.11;
  'D' :
   factor := 1.20;
  'E' :
   factor := 1.36;
  otherwise
   factor := 1.50
 end;
 writeln;
 writeln;
 writeln('Customer name : ', customer);
 cost := base * factor;
 writeln('Cost of muffler : $', cost : 6 : 2)
end.
```

FIGURE G.23

CHAPTER 10

1. a. Invalid; no loop range is specified.
   c. Valid.
   e. Valid, but an infinite loop will result.
   g. Valid, but `sum` and `count` initial values should have been outside the loop.
   i. Valid, if x is given an initial value.
2. a.
   ```
   for i := 1 to 10 do
    writeln('repeat');
   ```
   d.
   ```
   y := 2;
   while y < 500 do
    y := y * 2;
   ```
   i.
   ```
   repeat
    write('Enter a value for x:  ');
    readln(x)
   until x = 0;
   ```

j.
```
y := 2;
while y <= 20 do
  begin
    total := total - y;
    y := y + 2
  end;
```

3. a. 2
4
6
8
10
12
14
16
18
20

d. 8
6
8
4
6
8

g. 6561

5. See Figure G.24.
10. See Figure G.25.
18. See Figure G.26.
23. See Figure G.27.

```
program Sum;
{Sum integers 2, 5, 8, ... , 311}
 var
  number, total : integer;
begin
 total := 0;
 number := 2;
 repeat
  total := total + number;
  number := number + 3
 until number > 311;
 writeln('The sum of the numbers 2, 5, 8, ..., 311 is ', total : 4)
end.
```

FIGURE G.24

```
program FunctionTable;
{Produce a table of values for the function}
{     y = 2 * sqr(x) + 5 * x - 4}
 var
  x, y : real;
begin
 x := 1;
 writeln('Table of values for y = 2*sqr(x) + 5*x - 4');
 writeln('x' : 10, 'y' : 10);
 writeln('-----' : 10, '-----' : 10);
 repeat
  y := 2 * sqr(x) + 5 * x - 4;
  writeln(x : 10 : 2, y : 10 : 2);
  x := x + 0.2
 until x > 4.1  {accounts for round-off error}
end.
```

FIGURE G.25

```
program CustomerAccount;
{Determine finance charges to customer accounts}
 var
  name : string;
  oldbal, newbal, charge : real;
  days : integer;
begin
 writeln('Enter ZZZZZ to end input');
 write('Enter customer name:  ');
 readln(name);
 while name <> 'ZZZZZ' do
  begin
   write('Enter unpaid balance:  $');
   readln(oldbal);
   write('Enter number of days account is outstanding:  ');
   readln(days);
   charge := 0.015 * oldbal;
   if days >= 30 then
    charge := charge + 5.00;
   newbal := oldbal + charge;
   writeln('New balance:  $', newbal : 7 : 2);
   if days >= 30 then
    writeln('Payment overdue.');
   writeln;
   writeln('Enter ZZZZZ to end input');
   write('Enter customer name:  ');
   readln(name);
  end
end.
```

FIGURE G.26

```
program OverUnder;
 var
  bankroll, die1, die2, sum, bet, choice : integer;
  over : Boolean;
  ans : string;
begin
 over := false;
 bankroll := 100;
 writeln('Welcome to the game of Over/Under');
 writeln('Guess what the sum of the dice is and');
 writeln('triple your bet if you choose 7 or');
 writeln('double your money on any other correct choice');
 repeat
  writeln;
  write('Enter your choice (2 - 12):  ');
  readln(choice);
  repeat
   write('Enter your bet:  $');
   readln(bet)
  until bet <= bankroll;
  writeln;
  writeln('The dice are rolling...');
  die1 := random mod 6 + 1;
  die2 := random mod 6 + 1;
  sum := die1 + die2;
  writeln('The total is ', sum : 2);
  if choice = sum then
   begin
    writeln('You'' re a winner !');
    if choice = 7 then
     bankroll := bankroll + 3 * bet
    else
     bankroll := bankroll + 2 * bet;
    writeln('You now have $', bankroll : 4)
   end
  else
   begin
    writeln('Sorry, you lose.');
    bankroll := bankroll - bet;
    writeln('You have $', bankroll : 4, ' left . ')
   end;
  if bankroll = 0 then
   over := true
  else
   begin
    write('Continue?  (y/n):  ');
    readln(ans)
   end
 until ((ans = 'n') or (over));
 writeln('The games is over.  Your final bankroll is $', bankroll : 4)
end.
```

FIGURE G.27

CHAPTER 11

1. a. Invalid; subscript range is not declared.
   c. Valid.
   e. Valid.
   g. Invalid; only [and] can be used to enclose subscripts.
   i. Valid.
2. a. −13
   c. 0
   e. −5
3. a.
```
type
 client = record
   name : string;
   owed : real
  end;
```
   d.
```
i := 1;
flag := false;
for i := 1 to 100 do
 if person[i].age = 31 then
  flag := true;
if flag then
 writeln('Yes there is!')
else
 writeln('No there isn''t!');
```
   g.
```
type
  patient  = record
    name : string;
    bloodtype : string;
    pulse : integer;
    bloodpressure : string
   end;
  client = array[1..100] of patient;
```
4. a.
```
1
3
5
7
9
11
```
   c.
```
1
3
5
7
9
11
```
   e. Thomas needs long size.

5. See Figure G.28.
12. See Figure G.29.
13. See Figure G.30.
21. See Figure G.31.

```
program ArrayFill;
{Fill and display a 10-element array}
 type
  list = array[1..10] of integer;
 var
  data : list;

 procedure Entry;
  {Enter data into the list}
  var
   i : integer;
 begin
  writeln('Enter ten zero values for array elements');
  for i := 1 to 10 do
   begin
    write('Enter value #', i : 2, '-->');
    readln(data[i]);
   end
 end;

 procedure Display;
  {Displays the elements in the array}
  var
   i : integer;
 begin
  page;
  writeln('The elements in the array are: ');
  for i := 1 to 10 do
   writeln(data[i])
 end;

begin(*main program*)
 Entry;
 Display
end.
```

FIGURE G.28

```
program Sweets;
{Determines the public support for a new candy bar}
 type
  alpha = array[1..10] of string;
 var
  candy : alpha;

 procedure Survey;
  {Fills array with public responses}
  var
   i : integer;
 begin
  writeln('HIMHE Candy Company Survey');
  writeln;
  for i := 1 to 10 do
   begin
    write('Would you buy this candy bar? (yes/no)  -->');
    readln(candy[i])
   end
 end;

 procedure Counts;
  {Determines the count of yes and count of no answers}
  var
   i : integer;
   countyes, countno : integer;
 begin
  countyes := 0;
  countno := 0;
  for i := 1 to 10 do
   if candy[i] = 'yes' then
    countyes := countyes + 1
   else
    countno := countno + 1;
  writeln;
  writeln('The number of positive responses to the candy is ', countyes : 3);
  writeln('The number of negative responses to the candy is ', countno : 3)
 end;

begin(*main program*)
 Survey;
 Counts
end.
```

FIGURE G.29

```
program PickAWinner;
{Determines winner of sweepstakes}
 type
  name = array[1..5] of string;
 var
  star : integer;
  customer : name;

 procedure Load;
  {Fills the array with customer names}
  var
   i : integer;
 begin
  writeln('Hardware City Sweepstakes');
  writeln;
  for i := 1 to 5 do
   begin
    write('Enter customer name #', i : 2, '-->');
    readln(customer[i])
   end
 end;

 function Winner : integer;
  {Randomly selects an integer from 1 to 5 inclusive}
 begin
  winner := trunc(random mod 5 + 1);
 end;

begin(*main program*)
 Load;
 page;
 star := Winner;
 writeln('The lucky winner is number ', star : 3, ' --> ', customer[star])
end.
```

FIGURE G.30

```
program BaseballStatistics;
{Creating and displaying an array of baseball statistics}
 type
  player = record
    name : string;
    atbats : integer;
    nohits : integer;
    nohomers : integer;
    average : real
   end;
  team = array[1..9] of player;
 var
  slugger : team;
```

FIGURE G.31 *(continued)*

```
  procedure CreateRecords;
   {Create array of statistics}
   var
    i : integer;
  begin
   for i := 1 to 9 do
    begin
     writeln('Record #', i : 2, ' :');
     write('    Player name :');
     readln(slugger[i].name);
     write('    At bats :');
     readln(slugger[i].atbats);
     write('    No. of hits :');
     readln(slugger[i].nohits);
     write('    No. of homers:');
     readln(slugger[i].nohomers)
    end
  end;

  procedure ComputeAverages;
   {Computes batting average for each player}
   var
    i : integer;
  begin
   for i := 1 to 9 do
    slugger[i].average := round(slugger[i].nohits / slugger[i].atbats * 1000.0) / 1000.0
  end;

  procedure Display;
   {Display statistics array}
   var
    i : integer;
  begin
   page;
   writeln('Slugger Baseball Team Statistics' : 50);
   writeln;
   writeln('Player' : 10, 'At Bats' : 12, 'No. Hits' : 10, 'No. Homers' : 13, 'Average' : 12);
   for i := 1 to 9 do
    begin
     write(slugger[i].name : 10, slugger[i].atbats : 10, slugger[i].nohits : 10);
     write(slugger[i].nohomers : 13, slugger[i].average : 10 : 3);
     writeln
    end
  end;

begin (*main program*)
 CreateRecords;
 ComputeAverages;
 Display
end.
```

FIGURE G.31

CHAPTER 12

1. a. See Figure G.32.
   c. See Figure G.33.
   e. See Figure G.34.

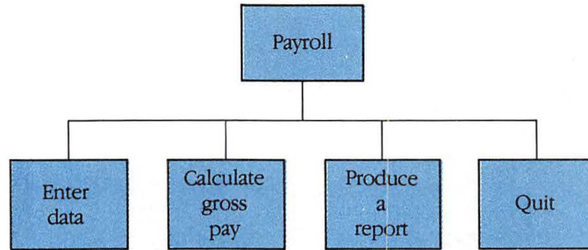


FIGURE G.32

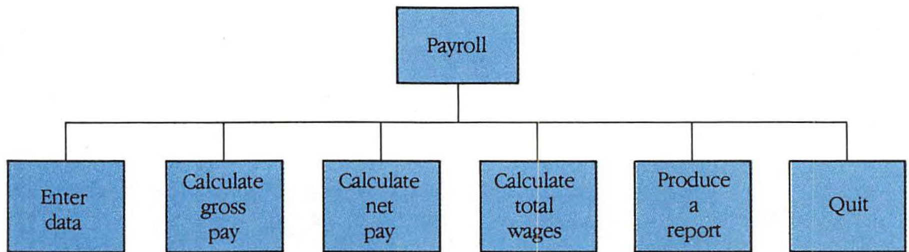


FIGURE G.33

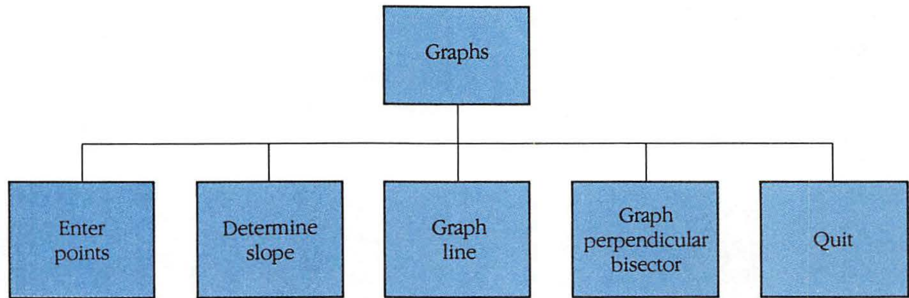


FIGURE G.34

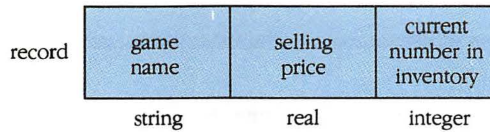
2. a.
```
repeat
  writeln('Payroll');
  writeln;
  writeln('<1>  Enter Employee Data');
  writeln('<2>  Find Gross Pay');
  writeln('<3>  Produce Report');
  writeln('<4>  Quit');
  writeln;
  repeat
   write('Enter the number of your choice:  ');
   readln(choice)
  until (choice <= 4) and (choice >= 1);
  case choice of
   1 :
    EnterData( <parameters> );
   2 :
    GrossPay( <parameters> );
   3 :
    Report( <parameters> );
   otherwise
    Quit
  end
until choice = 4
```

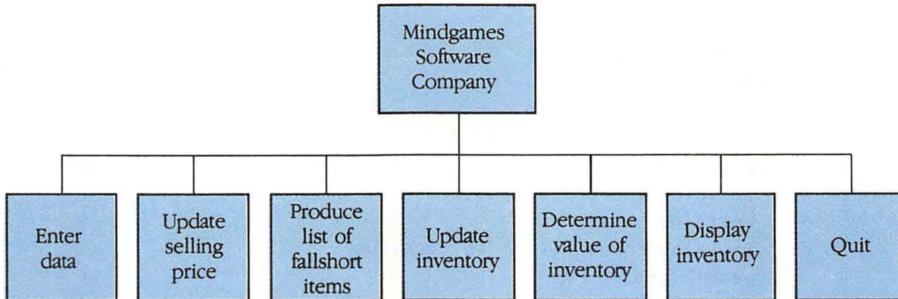
c.
```
repeat
  writeln('Payroll');
  writeln;
  writeln('<1>  Enter Employee Data');
  writeln('<2>  Find Gross Pay');
  writeln('<3>  Produce Report');
  writeln('<4>  Find Net Pay');
  writeln('<5>  Total Payroll');
  writeln('<6>  Quit');
  writeln;
  repeat
   write('Enter the number of your choice:  ');
   readln(choice)
  until (choice <= 6) and (choice >= 1);
  case choice of
   1 :
    EnterData( <parameters> );
   2 :
    GrossPay( <parameters> );
   3 :
    Report( <parameters> );
   4 :
    NetPay( <parameters> );
```

```
     5 :
      TotalPayroll( <parameters> );
     otherwise
      Quit
    end
   until choice = 6
```

e.
```
repeat
    writeln('Graph');
    writeln;
    writeln('<1>  Find Slope');
    writeln('<2>  Graph Line');
    writeln('<3>  Midpoint');
    writeln('<4>  Quit');
    writeln;
    repeat
     write('Enter the number of your choice:   ');
     readln(choice)
    until (choice <= 4) and (choice >= 1);
    case choice of
     1 :
      Slope( <parameters> );
     2 :
      GraphLine( <parameters> );
     3 :
      Midpoint( <parameters> );
     otherwise
      Quit
    end
   until choice = 4
```

6. See Figure G.35.






```
program Mindgames;
{Inventory for Mindgames Software Company}
 const
  max = 5;
 type
  product = record
    name : string;
    sell : real;
    num : integer
   end;
  list = array[1..max] of product;
 var
  MSC : list;
  pct : real;
  choice : integer;
  min : integer;

 procedure Entry (var a : list);
  {Enter data for inventory processing}
  var
   i : integer;
 begin
  page;
  writeln('Data Entry');
  writeln;
  for i := 1 to max do
   begin
    write('Name of game : ');
    readln(a[i].name);
    write('Price per : $');
    readln(a[i].sell);
    write('Number on hand : ');
    readln(a[i].num)
   end
 end;
```

FIGURE G.35 *(continued)*

```
procedure ChangePrice (var b : list;
        var percent : real);
 {Updates selling price on all games}
 var
  i : integer;
  ans : char;
begin
 for i := 1 to 5 do
  b[i].sell := b[i].sell * (1.0 + percent);
 page;
 writeln('The price update is now complete');
 writeln;
 writeln('Press Return to continue.' : 35);
 readln(ans)
end;

procedure ShortFallList (c : list;
        limit : integer);
 {Displays list of all games whose numbers fall below the  limit}
 var
  i : integer;
  flag : boolean;
  ans : char;
begin
 page;
 writeln('Fall Short List  : limit = ' : 25, limit : 4);
 writeln;
 writeln('Game Name' : 15, 'Number' : 10);
 writeln('___________________________________');
 flag := false;
 for i := 1 to max do
  if c[i].num < limit then
   begin
    writeln(c[i].name : 15, c[i].num : 10);
    flag := true
   end;
 if not flag then
  writeln('All items in stock equal to or above the limit');
 writeln;
 write('         Press return to continue');
 readln(ans)
end;

procedure InventoryUpdate (var d : list);
 {Updates inventory for a sale or a delivery}
```

FIGURE G.35 *(continued)*

```
 var
  i : integer;
  amt : integer;
  delv : integer;
  sord : char;
  game : string;
  flag : boolean;
  ans : char;
begin
 page;
 flag := false;
 writeln('Is this a sale or a delivery ? (S/D)');
 readln(sord);
 if (sord = 'S') or (sord = 's') then
  begin
   writeln('This is a sale update:');
   write('     Name of game sold --> ');
   readln(game);
   write('     Number of games sold --> ');
   readln(amt);
   i := 1;
   repeat
    if d[i].name = game then
     flag := true
    else
     i := i + 1
   until flag or (i > 5);
   if flag then
    if d[i].num >= amt then
     d[i].num := d[i].num - amt
    else
     begin
      writeln('Sorry.  Insufficient stock to cover order. ');
      writeln('Sell order amount:', amt : 4, '  Stock amount: ', d[i].num : 4)
     end
   else
    begin
     writeln('Sorry.  That game is not in our inventory stock.  Please');
     writeln('check the inventory display to see listing of our games.')
    end
  end
 else
  begin
   writeln('This is a delivery update:');
   write('     Enter name of game --> ');
```

FIGURE G.35 *(continued)*

```
    readln(game);
    write('      Enter number delivered --> ');
    readln(delv);
    i := 1;
    repeat
     if d[i].name = game then
       flag := true
     else
       i := i + 1
    until flag or (i > 5);
    if flag then
     d[i].num := d[i].num + delv
    else
     begin
      writeln('Sorry.  That game is not in our inventory stock.  Please');
      writeln('check the inventory display to see listing of our games.')
     end
   end;
  writeln;
  write('Press Return to continue.' : 30);
  readln(ans)
 end;

 procedure Value (e : list);
  {Determine value of inventory for total sale}
  var
   i : integer;
   total : real;
   ans : char;
 begin
  total := 0.0;
  for i := 1 to max do
   total := total + e[i].sell * e[i].num;
  writeln('The total selling value of this inventory is $', total : 10 : 2);
  writeln;
  write('Press Return to continue' : 35);
  readln(ans)
 end;

 procedure Display (f : list);
  {Display list of entire stock}
  var
   i : integer;
   ans : char;
 begin
  page;
  writeln('Mindgames Software Company' : 30);
  writeln('Inventory List' : 22);
  writeln;
  writeln('Game Name' : 15, 'Selling Price' : 15, 'Number on Hand' : 20);
```

FIGURE G.35 *(continued)*

```
  writeln('______________________________________________');
  for i := 1 to max do
   writeln(f[i].name : 15, f[i].sell : 10 : 2, f[i].num : 12);
  writeln;
  writeln('      Press Return to continue');
  readln(ans)
 end;

 procedure Quit;
  {End of job}
 begin
  page;
  writeln('End of processing.' : 30)
 end;

begin (***main***)
 Entry(MSC);
 repeat
  page;
  writeln('Mindgames Software Company Inventory Processing');
  writeln;
  writeln('<1> Update selling prices');
  writeln('<2> Produce Fall Short List');
  writeln('<3> Inventory Update');
  writeln('<4> Value of Inventory');
  writeln('<5> Display Inventory');
  writeln('<6> Quit');
  writeln;
  repeat
   write('        Enter number of option choice --> ');
   readln(choice);
  until (choice >= 1) and (choice <= 6);
  case choice of
   1 :
    begin
     page;
     write('Enter selling price percent increase (as a decimal) :');
     readln(pct);
     ChangePrice(MSC, pct)
    end;
   2 :
    begin
     page;
     write('Enter minimum number for stocked item :');
     readln(min);
     ShortFallList(MSC, min)
    end;
```

FIGURE G.35 *(continued)*

```
        3 :
         begin
          page;
          InventoryUpdate(MSC)
         end;
        4 :
         begin
          page;
          Value(MSC)
         end;
        5 :
         Display(MSC);
        otherwise
         Quit
       end
     until choice = 6
    end.
```

FIGURE 6.35

CHAPTER 13

1. a. Invalid; an internal file identifier is missing.
   c. Invalid; an internal file identifier is missing.
   e. Invalid; the argument should be a file identifier.
   g. Valid, if `f` is a file identifier.
2. a. `open(ESG, JJD);`
   c. `reset(ESG);`
   e. `x := filepos(ESG);`
3. a.
```
customer = record
    name : string;
    telenum : string
  end;
```
   c.
```
housetype = (colonial, cape, ranch, splitlevel);
listing = record
    ask : real;
    owner : string;
    loc : string;
    house : housetype
  end;
```
   e.
```
product = record
    name : string;
    sell : real;
    num : integer
  end;
```
4. See Figure G.36.
16. See Figure G.37.

```
program Payroll;
{Determines gross salary of all employees in the file}
 type
  employee = record
    name : string;
    ssno : string;
    hrwage : real
   end;
  inc = file of employee;
  list = array[1..12] of real;
 var
  ABC : inc;
  gross : list;
  count : integer;

 procedure CreateAFile;
  {Creates file of employee records}
  var
   person : employee;
   ans : char;
 begin
  open(ABC, 'Company');
  writeln('Enter data for employees of ABC Company');
  writeln;
  writeln('Type in ZZZ for employee name to end entering records');
  writeln;
  write('   Enter employee name : ');
  readln(person.name);
  while person.name <> 'ZZZ' do
   begin
    write('   Enter social security number :');
    readln(person.ssno);
    write('  Enter hourly wage :$');
    readln(person.hrwage);
    write(ABC, person);
    writeln;
    write('   Enter employee name :');
    readln(person.name)
   end;
  close(ABC);
  writeln;
  write('        Press Return to continue.');
  readln(ans)
 end;
```

FIGURE G.36 *(continued)*

```
procedure ProcessSalary (var i : integer);
 {Enter number of hours worked and calculate gross salary}
 {for each employee record in the file}
 var
  person : employee;
  nohours : real;
  ans : char;

begin
 page;
 open(ABC, 'Company');
 i := 0;
 while not (eof(ABC)) do
  begin
   i := i + 1;
   read(ABC, person);
   write('Employee :');
   writeln(person.name);
   write('Enter number of hours worked :');
   readln(nohours);
   gross[i] := person.hrwage * nohours;
  end;
 close(ABC);
 writeln;
 write('       Press Return to continue');
 readln(ans)
end;

procedure Display (i : integer);
 {Displays payroll data}
 var
  ans : char;
  j : integer;
  person : employee;
begin
 page;
 open(ABC, 'Company');
 writeln('ABC Company Employee Gross Salary Report' : 50);
 writeln;
 writeln('Employee' : 15, 'Soc.Sec. Number' : 20, 'Gross Pay' : 15);
 for j := 1 to i do
  begin
   read(ABC, person);
   writeln(person.name : 15, person.ssno : 20, '    $', gross[j] : 8 : 2)
  end;
 close(ABC);
 writeln;
 write('     Press Return to continue.');
 readln(ans)
end;
```

FIGURE G.36 *(continued)*

```
begin (*main program*)
 CreateAFile;
 ProcessSalary(count);
 Display(count);
 page;
 writeln('End of processing.' : 35)
end.
```

FIGURE G.36

```
program Mindgames;
{Inventory for Mindgames Software Company}
 const
  max = 5;
 type
  product = record
    name : string;
    sell : real;
    num : integer
   end;
  list = file of product;
 var
  sftwre : product;
  MSC : list;
  pct : real;
  choice : integer;
  min : integer;

 procedure Entry;
  {Enter data for inventory file processing}
 begin
  page;
  writeln('Data Entry');
  writeln;
  open(MSC, 'Gamedata');
  writeln('End processing by entering game name ''ZZZ'' ');
  writeln;
  write('Name of game : ');
  readln(sftwre.name);
  while sftwre.name <> 'ZZZ' do
   begin
    write('Price per : $');
    readln(sftwre.sell);
    write('Number on hand : ');
    readln(sftwre.num);
    write(MSC, sftwre);
    writeln;
    write('Name of game: ');
    readln(sftwre.name)
   end;
  close(MSC)
 end;
```

FIGURE G.37 *(continued)*

```
procedure ChangePrice (var percent : real);
 {Updates selling price on all games in the file}
 var
  ans : char;
  x : integer;
begin
 open(MSC, 'Gamedata');
 while not (eof(MSC)) do
  begin
   read(MSC, sftwre);
   x := filepos(MSC);
   sftwre.sell := sftwre.sell * (1.0 + percent);
   seek(MSC, x - 1);
   write(MSC, sftwre)
  end;
 page;
 writeln('The price update is now complete');
 writeln;
 writeln('Press Return to continue.' : 35);
 readln(ans);
 close(MSC)
end;

procedure ShortFallList (limit : integer);
 {Displays list of all games whose numbers fall below the limit}
 var
  flag : boolean;
  ans : char;
begin
 page;
 writeln('Fall Short List   : limit = ' : 25, limit : 4);
 writeln;
 writeln('Game Name' : 15, 'Number' : 10);
 writeln('_________________________________');
 flag := false;
 open(MSC, 'Gamedata');
 while not (eof(MSC)) do
  begin
   read(MSC, sftwre);
   if sftwre.num < limit then
    begin
     writeln(sftwre.name : 15, sftwre.num : 10);
     flag := true
    end
  end;
 if not flag then
  writeln('All items in stock equal to or above the limit');
 writeln;
 write('        Press Return to continue');
 readln(ans);
 close(MSC)
end;
```

FIGURE G.37 *(continued)*

```
procedure InventoryUpdate;
 {Updates inventory for a sale or a delivery of game in file}
 var
  amt : integer;
  delv : integer;
  sord : char;
  game : string;
  flag : boolean;
  ans : char;
  x : integer;
begin
 page;
 flag := false;
 writeln('Is this a sale or a delivery ? (S/D)');
 readln(sord);
 if (sord = 'S') or (sord = 's') then
  begin
   writeln('This is a sale update:');
   write('     Name of game sold --> ');
   readln(game);
   write('     Number of games sold --> ');
   readln(amt);
   open(MSC, 'Gamedata');
   while not (eof(MSC)) and not (flag) do
    begin
     read(MSC, sftwre);
     if sftwre.name = game then
      flag := true
    end;
   if flag then
    if sftwre.num >= amt then
     begin
      x := filepos(MSC);
      sftwre.num := sftwre.num - amt;
      seek(MSC, x - 1);
      write(MSC, sftwre)
      end
     else
      begin
       writeln('Sorry.  Insufficient stock to cover order. ');
       writeln('Sell order amount:', amt : 4, '  Stock amount: ', sftwre.num : 4)
      end
    else
     begin
      writeln('Sorry.  That game is not in our inventory stock.  Please');
      writeln('check the inventory display to see listing of our games.')
     end
  end
 else
  begin
   writeln('This is a delivery update:');
   write('     Enter name of game --> ');
```

FIGURE G.37 *(continued)*

```
    readln(game);
    write('      Enter number delivered --> ');
    readln(delv);
    while not (eof(MSC)) and not (flag) do
     begin
      read(MSC, sftwre);
      if sftwre.name = game then
       flag := true
     end;
    if flag then
     begin
      x := filepos(MSC);
      sftwre.num := sftwre.num + delv;
      seek(MSC, x - 1);
      write(MSC, sftwre)
     end
    else
     begin
      writeln('Sorry.  That game is not in our inventory stock.  Please');
      writeln('check the inventory display to see listing of our games.')
     end
   end;
  writeln;
  write('Press Return to continue.' : 30);
  readln(ans);
  close(MSC)
 end;

 procedure Value;
  {Determine value of inventory for total sale}
  var
   total : real;
   ans : char;
 begin
  total := 0.0;
  open(MSC, 'Gamedata');
  while not (eof(MSC)) do
   begin
    read(MSC, sftwre);
    total := total + sftwre.sell * sftwre.num
   end;
  writeln('The total selling value of this inventory is $', total : 10 : 2);
  writeln;
  write('Press Return to continue' : 35);
  readln(ans);
  close(MSC)
 end;
```

FIGURE G.37 *(continued)*

```
procedure Display;
 {Display list of entire stock}
 var
  ans : char;
begin
 page;
 writeln('Mindgames Software Company' : 30);
 writeln('Inventory List' : 22);
 writeln;
 writeln('Game Name' : 15, 'Selling Price' : 15, 'Number on Hand' : 20);
 writeln('______________________________________________________');
 open(MSC, 'Gamedata');
 while not (eof(MSC)) do
  begin
   read(MSC, sftwre);
   writeln(sftwre.name : 15, sftwre.sell : 10 : 2, sftwre.num : 12);
  end;
 writeln;
 writeln('       Press Return to continue');
 readln(ans);
 close(MSC)
end;

procedure AddANew;
 {Appends a record to the file inventory}
begin
 page;
 open(MSC, 'Gamedata');
 repeat
  read(MSC, sftwre)
 until eof(MSC);
 writeln('Add a record to inventory file');
 writeln;
 write('Enter name of new game --> ');
 readln(sftwre.name);
 write('Enter selling price --> $');
 readln(sftwre.sell);
 write('Enter number in inventory --> ');
 readln(sftwre.num);
 write(MSC, sftwre);
 close(MSC)
end;

procedure Quit;
 {End of job}
begin
 page;
 writeln('End of processing.' : 30)
end;
```

FIGURE G.37 *(continued)*

```
begin (***main***)
 Entry;
 repeat
  page;
  writeln('Mindgames Software Company Inventory Processing');
  writeln;
  writeln('<1> Update selling prices');
  writeln('<2> Produce Fall Short List');
  writeln('<3> Inventory Update');
  writeln('<4> Value of Inventory');
  writeln('<5> Display Inventory');
  writeln('<6> Add a new software game to the file inventory');
  writeln('<7> Quit');
  writeln;
  repeat
   write('          Enter number of option choice --> ');
   readln(choice);
  until (choice >= 1) and (choice <= 7);
  case choice of
   1 :
    begin
     page;
     write('Enter selling price percent increase (as a decimal) :');
     readln(pct);
     ChangePrice(pct)
    end;
   2 :
    begin
     page;
     write('Enter minimum number for stocked item :');
     readln(min);
     ShortFallList(min)
    end;
   3 :
    begin
     page;
     InventoryUpdate
    end;
   4 :
    begin
     page;
     Value
    end;
   5 :
    Display;
   6 :
    AddANew;
   otherwise
    Quit
  end
 until choice = 7
end.
```

FIGURE G.37

CHAPTER 14

1. a. Valid.
   c. Valid.
   e. Invalid; the caret (^) precedes **real** and not **pointreal**.
   g. Invalid; a string cannot be stored in the pointer identifier letter.
2. a.

10	
10	10
5	10
5	

   c.

108	42
	42
24	42
24	18
6	18
6	12

   Answer is 6
3. See Figure G.38.
13. See Figure G.39.
17. See Figure G.40.

```
program IceCreamSales;
{Summarize ice cream sales}
 type
  flavor = (vanilla, chocolate, strawberry);
  sales = array[flavor] of integer;
 var
  total : sales;
  kind : flavor;
  number : integer;
  ans : char;

 procedure Initialize (var total : sales);
  {Initialize array elements}
  var
   i : flavor;
 begin
  for i := vanilla to strawberry do
   total[i] := 0
 end;
```

FIGURE G.38 *(continued)*

```
procedure EnterData (var total : sales);
 {Enter data into array elements}
begin
 repeat
  write('Enter flavor:  ');
  readln(kind);
  write('Enter number of half-gallons sold:  ');
  readln(number);
  total[kind] := total[kind] + number;
  write('Continue?  (y/n) ');
  readln(ans);
  writeln
 until ans = 'n'
end;

procedure Display (var total : sales);
 {Output results}
 var
  j : flavor;
begin
 page;
 writeln('Ice Cream Sales');
 writeln;
 writeln('flavor' : 20, 'half-gallons sold' : 21);
 writeln('---------------------------------' : 41);
 for j := vanilla to strawberry do
  writeln(j : 20, total[j] : 20)
end;

begin
 Initialize(total);
 EnterData(total);
 Display(total)
end.
```

FIGURE G.38

```
program RandomNumberAverage;
{Find the average of random numbers stored in a linked list}
 type
  nodepointer = ^nodetype;
  nodetype = record
    number : integer;
    next : nodepointer
   end;
 var
  p,                   {points to newly acquired node}
  q,                   {follows r down the linked list}
  r,                   {points to next node in list}
```

FIGURE G.39 *(continued)*

```
 start      {points to first node in list}
  : nodepointer;
 value, size : integer;
 average : real;

procedure FillNode;
 {Gets a new node and fills its fields}
begin
 new(p);
 p^.number := value;
 p^.next := nil
end;

procedure InsertNode;
 {Puts a new node onto a linked list}
begin
 if start = nil then
  begin
   FillNode;
   start := p
  end
 else
  begin
   r := start;
   q := start;
   repeat
    q := r;
    r := r^.next
   until r = nil;
   FillNode;
   q^.next := p
  end
end;

procedure MakeList;
 {Creates a linked list of random numbers}
 var
  i : integer;
begin
 write('Enter number of random numbers in list:  ');
 readln(size);
 for i := 1 to size do
  begin
   value := random;
   InsertNode
  end
end;
```

FIGURE G.39 *(continued)*

```
  procedure FindAverage;
   {Finds the average of the numbers in the linked list}
   var
    sum : longint;
  begin
   sum := 0;
   p := start;
   while p <> nil do
    begin
     sum := sum + p^.number;
     p := p^.next
    end;
   average := sum / size
  end;

  procedure Display;
   {Outputs results}
  begin
   page;
   writeln('List' : 11);
   writeln('-------' : 11);
   p := start;
   while p <> nil do
    begin
     writeln(p^.number : 10);
     p := p^.next
    end;
   writeln;
   writeln('The average of the numbers in the list is ', average : 8 : 2)
  end;

begin
 start := nil;
 MakeList;
 FindAverage;
 Display
end.
```

FIGURE G.39

```
program Product;
{Find the product of two integers using recursion}
 var
  answer, x, y : integer;

 function RecursiveProduct (x, y : integer) : integer;
  {Finds the product of two integers}
 begin
  if y = 1 then
   RecursiveProduct := x
  else
   RecursiveProduct := x + RecursiveProduct(x, y - 1)
 end;

begin
 write('Enter the first factor for a product:  ');
 readln(x);
 write('Enter the second factor for the product:  ');
 readln(y);
 answer := RecursiveProduct(x, y);
 writeln('The product of ', x : 5, ' and ', y : 5, ' is ', answer : 5)
end.
```

FIGURE G.40

APPENDIX H

Macintosh Pascal Version 2.1

This text was designed to be used with the original version of Macintosh Pascal. Macintosh Pascal Version 2.1, an updated version, is now available. All the programs in this text produce correct results whether executed under the original or under the newer version of MacPascal; however, some significant enhancements are provided by Version 2.1. This appendix summarizes these enhancements according to each chapter in this text.

CHAPTER 1: THE MACINTOSH OPERATING SYSTEM

Although the menu header for the operating system screen is identical in both versions, the menus themselves differ. The Version 2.1 menus for the **File**, **View**, and **Special** menus are shown in Figure H.1.

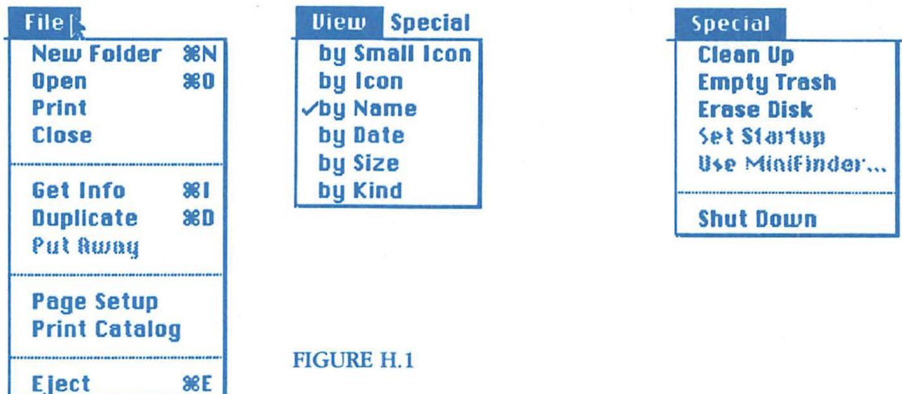


FIGURE H.1

The Macintosh Pascal icon in the upper right corner of the screen and its corresponding window now display "Macintosh Pascal 2.1."

CHAPTER 2: THE MACINTOSH PASCAL SYSTEM

As with the operating system, the menu header contains the same choices in both versions of Pascal, but some of the menus are different. In general, the biggest changes occur in the dialog boxes produced by selecting menu options. The newer dialog boxes offer more options and make Macintosh Pascal easier to use with more extensive systems such as ones employing a second disk drive or a fixed disk.

THE **FILE** MENU

The menu choices are identical in both versions, but the dialog boxes for **Open**, **Save**, **Save As**, and **Print** have significant changes.

The dialog box displayed when **Open** is chosen is shown in Figure H.2.

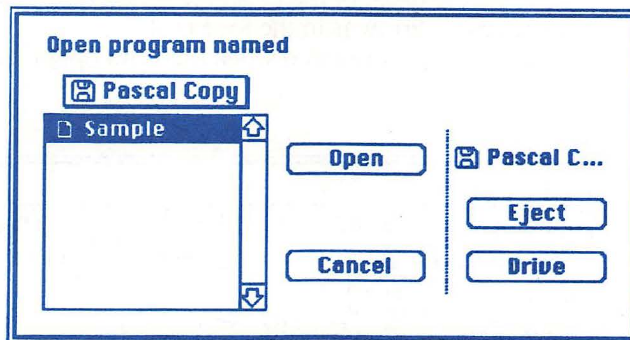


FIGURE H.2

The name of the disk appears over the catalog listing as well as over the **Eject** oval. A **Drive** oval is beneath the **Eject** oval and is used to display the catalog of a disk in an external drive.

The dialog box displayed when **Save As** is chosen is shown in Figure H.3.

Unlike the original version, the name given to the disk and a catalog listing appear at the top of the dialog box to indicate which file names are currently in use. A **Drive** oval appears below the **Eject** oval and is used to save a program on a disk in an external drive. The three options at the bottom of the box indicate three ways in which the program can be saved. **As Text** is the only one used in the original version. Details on when to use **As Object** and **As Application** are

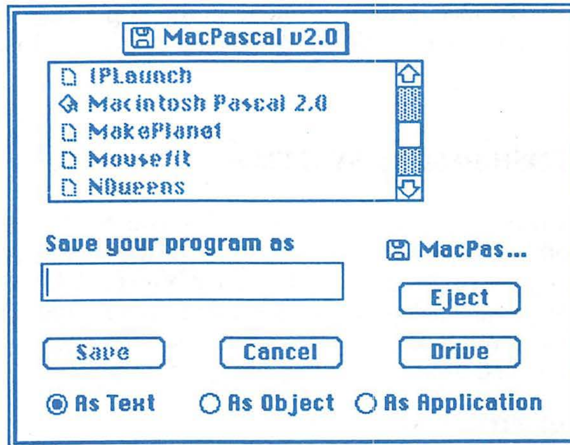


FIGURE H.3

found in the documentation accompanying Version 2.1. As in the original MacPascal, just type the file name of the program to be saved and click the mouse button while the screen arrow is in the **Save** oval.

The dialog box displayed when the **Print** option is chosen is shown in Figure H.4.

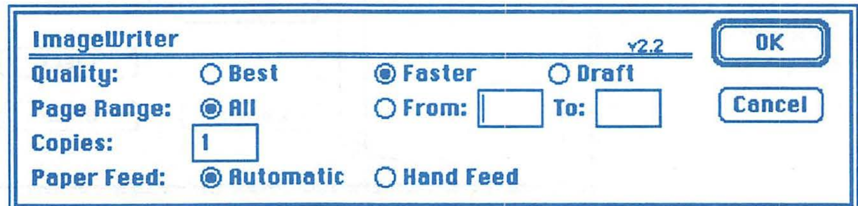


FIGURE H.4

The only significant difference in Version 2.1 is in the descriptions of the options: **High** becomes **Best**, **Standard** becomes **Faster**, **Continuous** becomes **Automatic**, and **Cut Sheet** becomes **Hand Feed**.

The **Edit**, **Search**, and **Run** menus are identical in both versions. Only the text used in the explanations of the operations of the menu options may differ.

In the **Windows** menu of Version 2.1, the **Type Size**. . . option in the original version has been replaced by **Font Control**. . . and **Preferences**. . . options as shown in Figure H.5.

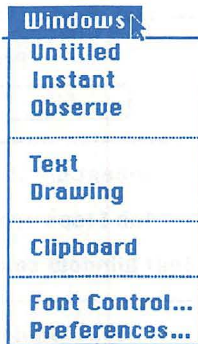


FIGURE H.5

Selecting **Font Control...** produces a dialog box similar to the one shown in Figure H.6.

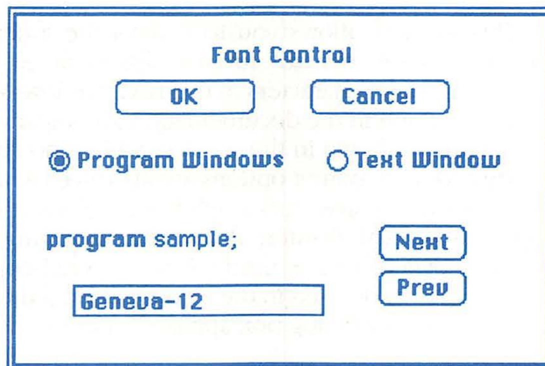


FIGURE H.6

In this option you can control the type and size of the characters shown in the program window and the **Text** window. After selecting one of these options, use the **Next** and **Prev** ovals to cycle through the fonts shown in the lower left portion of the box. After you have selected the font you want, click in the **OK** oval.

Preferences... produces a dialog box similar to the one shown in Figure H.7.

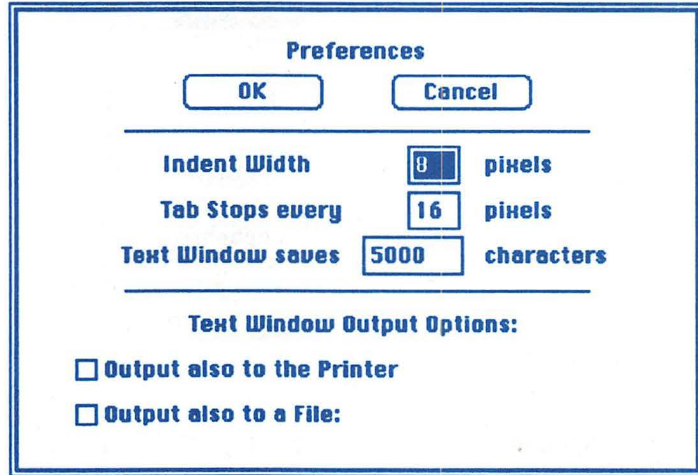


FIGURE H.7

This window allows you to control the indentation of statements in the program window, set tabs for the alignment of comments in the body of a program, and save characters in the **Text** window. More specific details on these options are given in the documentation accompanying the text. You can use the default options shown in the window with good results.

Text window output options are also given at the bottom of this dialog box. If you click the mouse button while the screen arrow is in the square next to **Output also to the Printer**, all the text appearing in the **Text** window is also printed. A default font is used for the printed copy and does not necessarily match the type style used in the **Text** window. If the **Output also to a File** option is chosen, another dialog box appears on the screen as shown in Figure H.8.

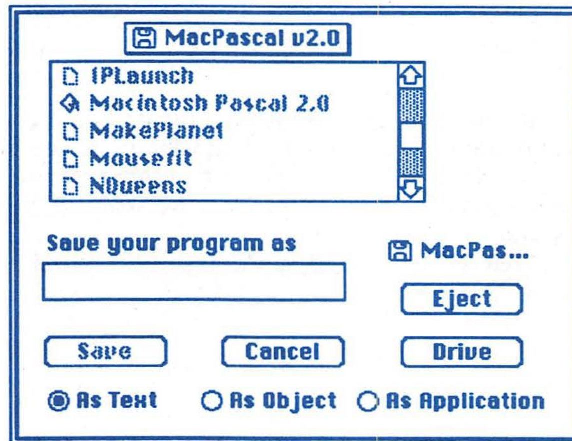


FIGURE H.8

All you have to do is to enter a file name and select **Save**. From that point on, any text appearing in the **Text** window is also saved in a file under the name you just specified.

Both of these **Text** window output options can be used at the same time and are in effect until you change them using the **Preferences...** option or until you leave MacPascal. They remain active even if you close one program and open another.

CHAPTER 3: FIRST STEPS IN PASCAL PROGRAMMING

This chapter contains material that is independent of MacPascal and therefore is valid as is.

CHAPTER 4: THE COMPONENTS OF A PASCAL PROGRAM

The error messages describing particular error conditions differ from one version to the next.

CHAPTER 5: INPUT AND OUTPUT

The default font used for the output produced in the **Text** window in Version 2.1 differs from the font used in the original version of MacPascal. Therefore, the alignment of output in the **Text** window may vary from one version to another. You may have to make adjustments in the output statements to produce aligned output.

The **Preferences...** option in the **Windows** menu in Version 2.1 can greatly simplify the output of data to any combination of the **Text** window, printer, and a text file.

In Version 2.1, the **page** instruction clears the **Text** window *and* skips a printed page if the printer is on. In the original version, only the **Text** window is cleared even if the printer is on.

CHAPTER 6: ASSIGNMENT STATEMENTS AND PROGRAMMING AIDS

The only distinctions of concern between versions of MacPascal are (1) the variation in text explaining errors, (2) the difference in the **Windows** menu discussed in previous chapters, and (3) the substitution of the term *Undefined name* in the Version 2.1 **Observe** window for the term *Unknown name* in the original version.

CHAPTER 7: SYSTEM-DEFINED FUNCTIONS AND PROCEDURES

There are no variations between the two versions for the material covered in this chapter of the text.

CHAPTER 8: PROGRAMMER-DEFINED FUNCTIONS AND PROCEDURES

There are no variations between the two versions for the material covered in this chapter of the text.

CHAPTER 9: CHOICES

There are no variations between the two versions for the material covered in this chapter of the text.

CHAPTER 10: LOOPS

There are no variations between the two versions for the material covered in this chapter of the text. However, the alignment of text when producing tables may differ because of the font type and size chosen for the **Text** window through the **Font Control...** option in the **Windows** menu.

CHAPTER 11: FUNDAMENTAL DATA STRUCTURES

There are no variations between the two versions for the material covered in this chapter of the text. However, the **page** instruction is used in many programs in this chapter. If it is on, **page** causes the printer to skip a page as well as clearing the **Text** window in Version 2.1 while only the **Text** window is cleared in the original version.

CHAPTER 12: MODULAR PROGRAMMING

There are no variations between the two versions for the material covered in this chapter of the text.

CHAPTER 13: ADVANCED DATA FILE TECHNIQUES

There are no variations between the two versions for the material covered in this chapter of the text. However, the **page** instruction is used in this chapter. In Version 2.1, if **page** is on, **page** causes the printer to skip a page as well as clearing the **Text** window; in the original version, only the **Text** window is cleared.

CHAPTER 14: ADVANCED PASCAL STRUCTURES

Figure 14.9 calculates factorials up to 12!. In the original version, if a number larger than 13 is entered, an error message results. In Version 2.1, an integer is displayed, but it is not the correct answer.

NOTE:

Some of the larger programs that were saved on a disk not containing Macintosh Pascal did not load at all when **Open** from the **File** menu header was selected, the Macintosh Pascal disk ejected, and the disk containing the desired program inserted into the drive. These programs can be loaded, however, by using the operating system window to transfer the program file onto the Macintosh Pascal disk before entering Pascal. This operation may not be a problem on later versions of MacPascal.

Index

abs, 129
absolute value, 129
active window, 29
actual parameter, 173
address, 387
All Occurrences, 26
ALU, 3
and, 204
apostrophe, 69
append, 369
argument, 128
arithmetic logic unit (ALU), 3
array, 268, 383, 387
ASCII code, 204, 406
assignment
    operator, 102
    statement, 102
As Text, 465
auxiliary storage, 5
axis
    horizontal, 74, 145
    vertical, 145
    x, 145
    y, 145

bar graph, 158
begin, 63, 166, 170, 207, 211, 217, 233, 241, 283
binary
    operator, 204
    search, 405
black box, 127
block, 180
body
    of function, 166
    of loop, 232
    of procedure, 170
boolean data type, 58
branch, 202
bubble sort, 291
bug, 47
by Icon, 9
by Name, 9

calling
    program, 128, 169
    statement, 128, 167
Cancel, 16
case, 215, 383
Case Is Irrelevant, 26
Cases Must Match, 26
central processing unit (CPU), 3
channel, 77
char data type, 58, 382
character, 58
circle, 150
Clean Up, 11
Clear, 22
Clipboard, 20, 31
close, 77, 82, 367
Close, 9, 16
coding, 46
command key, 8, 35
comment, 64, 313

compound statement, 207, 211
concat, 138
concatenation, 138
condition, 202, 232, 236, 238
conditional
    expression, 203
    statement, 202
const, 59
constant, 58
control, 3
coordinate, 145, 281
copy, 139
Copy, 21
counting statement, 245
CPU, 3
cursor, 5
Cut, 20

data, 3, 56
data file, 68, 76, 81
data structure
    dynamic, 387, 392
    static, 387
data type, 56
    boolean, 58
    char, 58
    double, 57
    enumerated, 382
    extended, 58
    generic, 61
    integer, 57
    longint, 57
    ordered, 382
    pointer, 387
    real, 57
    string, 58
debug, 47
decision, 202
declaration, 59, 64, 166
decomposition, 311
default case, 214, 217
delete, 140
delimiter, 60
device
    input, 3
    I/O, 3
    output, 3
dialog box, 15, 465
directory, 16
disk, 5, 68
disk drive, 5
display screen, 3
dispose, 390
div, 100, 102
documentation, 44, 310, 313
double data type, 57
double-option structure, 203, 207
double precision real, 57
downto, 234
Drawing window, 15, 31, 69, 74, 128, 145
drawline, 147
Drive, 465
dynamic data structure, 387, 392

e-notation, 57, 72
echo, 83
Edit, 19
Eject, 9, 16, 465
else, 208
Empty Trash, 11
end, 63, 166, 170, 207, 211, 215, 217, 233, 241, 282, 283
endcode, 237
end-of-file, 368
enumerated data type, 382
eof, 368
error, 47
    logic, 47, 113
    message, 60
    syntax, 47, 113
erroneous data, 214, 258
Everywhere, 26
exponentiation, 100
expression, 100
    conditional, 203
extended data type, 58
extended precision real, 58
external file, 76, 81, 363, 365

factorial, 247, 395
field, 71, 282
field width descriptor, 71, 73
file, 76
file, 8, 363
    data, 68, 76, 81
    external, 76, 81, 363, 365
    identifier, 76
    internal, 365
    name, 76
    of records, 363, 364

File, 7, 15, 464, 465
file pointer, 365
filepos, 365
file position, 366
Find, 27
flag, 292
Font Control, 467, 470
for. . .do, 233
formal parameter, 173
framerect, 148
function, 166
function, 127, 165, 411
call, 173
nested, 130
numeric, 128
string, 137

generic data type, 61
global identifier, 174, 194, 286
Go, 28, 116
Go-Go, 118
graphic
image, 146
output, 30
procedure, 146
grid, 145

Halt, 29, 245
hard copy, 78
hardware, 3
header line, 166, 168
heap, 387
horizontal axis, 74, 145

icon, 6, 365
identifier, 44, 59, 73, 101, 108
global, 174
local, 166
summing, 246
if. . .then. . .else, 206
independence, 175
index, 233, 269
infinite loop, 263
initialization, 36
input, 44, 68, 78, 81
input device, 3
Input-Processing-Output (IPO) chart, 44, 82, 99, 104, 178, 180, 311
insert, 140, 143, 392
instruction, 3, 49, 56, 409
integer data type, 57, 382
long, 57
internal file, 76, 81, 365
invertcircle, 151
invertrect, 150
IPO chart, 44, 82, 99, 104, 178, 180, 311
I/O device, 3

justified
left, 72
right, 71, 73

K, 9
keyboard, 5, 68, 78
keyword, 69
key equivalent, 8, 35

language, 56
left-justified, 72
length, 137
lineto, 147
linear search, 291
linked list, 390
list, 269
linked, 390
local
identifier, 166, 168, 170
scope, 175
logic error, 47, 113
logical operator, 204
loop, 231, 232, 395
body, 233
longint data type, 57, 396
long integer, 57

main memory, 3, 68, 81
main program, 180, 309, 315
maintenance, 175
mean, 320
median, 320
menu, 314
menu header, 6
merge, 293
microcomputer, 3
mod, 100
mode, 328
modularization, 310
module, 309
monitor, 3, 68
mouse, 3

moveto, 74, 146
multi-option structure, 203

Nassi-Shneiderman (N-S) chart, 46, 82, 104, 178, 207, 211, 217, 233, 237, 241, 311
nested
  function, 130
  loop, 252
  parentheses, 101, 205
  structure, 213
new, 388
New, 15
Next, 467
nil, 387
node, 390
not, 204
N-S chart, 46, 82, 104, 178, 207, 211, 217, 233, 237, 241, 311
null string, 58
numeric
  function, 128
  operator, 99

Observe window, 113, 469
odd, 130
one-dimensional array, 269
open, 76, 81, 128, 365
Open, 15, 465
operating system, 5
operand, 99, 203
operator, 99
  assignment, 120
  binary, 204
  logical, 204
  numeric, 99
  relational, 203
  unary, 204
operation, 99
option, 202
or, 204
ordered data type, 382
origin, 74, 145, 281
otherwise, 215, 217
output, 44, 68, 104
Output also to a file, 468
Output also to the printer, 468
output device, 3

page, 74, 469, 470, 471
paintcircle, 151
paintrect, 149
parameter, 128, 166
  actual, 173
  formal, 173
  value, 167, 169
  variable, 169
parentheses, 101, 205
  nested, 101, 205
Paste, 21
Pause, 28, 245
pixel, 145, 150, 281
pointer, 387
  file, 365
pointer data type, 387
pointing finger icon, 115, 117
pos, 138
position, 138
precedence, 101, 205
Preferences, 467, 469
Prev, 467
prime, 253
Print, 18, 466
printer, 5, 68, 78
printing position, 70
printout, 78
problem solving process, 43, 54, 82, 104, 310
procedure, 127, 140, 165, 411
  graphic, 146
  string, 137
procedure, 169
procedure header, 170
processing, 44
processing section, 64
program, 5, 56, 166
  calling, 128
  header line, 63
  header section, 64
  main, 309, 315
  name, 63, 166
  run, 69
  window, 15, 69, 118, 467
program, 63
programming language, 56
programming team, 309

Quit, 18

random, 131, 166
range, 62, 269
read, 79, 18, 128, 367

readln, 79, 128
real data type, 57, 102
  double precision, 57
  extended precision, 58
record, 268, 282, 364
recursion, 395
recursive subprogram, 395
refinement, 312
relational operator, 203
repeat. . .until, 236, 238
repetition structure, 232, 238
Replace, 28
Replace with, 26
reserved word, 59, 69, 408
reset, 369
Reset, 29
rewrite, 78, 218
right-justified, 71, 73
round, 131
Run, 28, 113, 115, 116

Save, 17, 465
Save As, 17, 465
scientific notation, 57
scope, 174
  local, 175
scroll, 9
search, 291, 370, 405
  binary, 405
  linear, 291
  sequential, 291
Search, 25
Search for, 26
secondary storage, 5
seek, 370
Select All, 22
selection structure, 202, 312
selector, 217
semicolon, 60
sentinel, 237
Separate Words, 26
sequence structure, 202, 312
sequential search, 291
single-option structure, 203, 206
software, 3
software package, 5
sort, 291
  bubble, 291
Special, 11, 464
sqr, 129
sqrt, 130
square, 129
square root, 130
startup disk, 36
statement
  assignment, 102
  calling, 138, 167
  compound, 207, 211
  conditional, 202
  counting, 245
  summing, 245, 246
static data structure, 387
Step, 115
Step-Step, 116
stop sign, 116
Stops In, 116
Stops Out, 117
storage
  auxiliary, 5
  secondary, 5
string
  argument, 139
  data type, 57, 76
  function, 137
  procedure, 137
structure
  double-option, 203, 208
  multi-option, 203
  nested, 213
  repetition, 232, 238
  selection, 202, 312
  sequence, 312
  single-option, 203, 206
structured programming, 312
subprogram, 127, 165, 315
  recursive, 395
subrange, 61, 269
subscript, 269
summing
  identifier, 246
  statement, 245, 246
syntax error, 47, 113

target value, 291
terminating value, 395
text, 78
Text window, 15, 30, 69, 467
top-down design, 311
Trash, 7, 11
translation, 156
trunc, 131
truncate, 131

type, 61, 269
Type Size, 31

unary operator, 204
Untitled, 15, 30
update, 369
user-friendly, 313

value parameter, 167, 169
var, 60, 64, 170, 269
variable, 58, 60, 99, 170
variable parameter, 169, 175
vector, 269
vertical axis, 145
View, 9, 464

walkthrough, 47, 292
What to find, 26
while. . .do, 236
window, 8
  active, 29
  Drawing, 15, 31, 69, 74, 145
  Observe, 113, 469
  Text, 15, 30, 69, 467
Windows, 20, 29, 113, 466
with, 283
word processing, 137
write, 69, 77, 79, 128, 365
writedraw, 75, 128, 155
writeln, 69, 78, 128

x-axis, 145

y-axis, 145

ISBN 0-02-329521-X