

# Structured BASIC for the IBM PC<sup>®</sup>

---


## *A Hands-On Approach*

---

Herbert Peckham  
Wade Ellis, Jr.  
Ed Lodi



**BIRMINGHAM  
SOUTHERN  
COLLEGE  
LIBRARY**

  
GIFT OF

WITHDRAWN  
Birmingham-Sou.  
College Lib.

PROF. JEFFREY SPEARS

1988

# Structured BASIC for the IBM PC<sup>®</sup>

*A Hands-On Approach*

Herbert Peckham  
Wade Ellis, Jr.  
Ed Lodi

A |  
Computer  
Literacy  
BOOK

McGraw-Hill Book Company

New York St. Louis San Francisco Auckland Bogotá Hamburg  
Johannesburg London Madrid Mexico Montreal New Delhi  
Panama Paris São Paulo Singapore Sydney Tokyo Toronto

**Structured BASIC for the IBM PC®: A Hands-On Approach**

Copyright © 1985 by McGraw-Hill, Inc. All rights reserved. Printed in the United States of America. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the prior written permission of the publisher.

1 2 3 4 5 6 7 8 9 0 S E M S E M 8 9 8 7 6 5

ISBN 0-07-049162-3

The editor was Gerald A. Gleason;  
the production supervisor was Joe Campanella.  
The cover was designed by Oona Johnson Design.  
Semline, Inc., was printer and binder.

IBM PC is a registered trademark of International Business Machines Corporation.

Library of Congress Cataloging in Publication Data

Peckham, Herbert D.

Structured BASIC for the IBM PC.

(A Computer literacy book)

Includes index.

1. IBM Personal Computer — Programming. 2. Basic  
(Computer program language) 3. Structured programming.

I. Ellis, Wade. II. Lodi, Ed. III. Title.

IV. Series.

QA76.8.I2594P44 1985 001.64'2 85-11296

ISBN 0-07-049162-3



PREFACE	1
Acknowledgments	2

## INTRODUCTION 3

THE PC DOS OPERATING SYSTEM	3
The Parts of Your Computer	3
What Is an Operating System	3
The PC DOS Operating System	3
Program Transportability	4
Learning to Access BASIC from PC DOS	4
COMPUTER ACTIVITIES	4
SUMMARY	7

## CHAPTER 1

### DIRECT MODE BASIC 9

1—1	OBJECTIVES	9
	Reviewing the Start-up Procedure	9
	Learning to Use PRINT with Strings and Numbers	9
	Using the PRINT Statement to Perform Simple Operations	9
	Using Variable Names in BASIC	9
	Correcting Mistakes	9
1—2	DISCOVERY EXERCISES	9
1—3	DISCUSSION	15
	Reviewing Start-up Procedure	15
	Direct-Mode BASIC	15
	Learning to Use PRINT with Strings and Numbers	15
	Using the PRINT Statement to Perform Simple Operations	16
	Using Variable Names in BASIC	17
	Correcting Mistakes	18
1—4	PRACTICE TEST	19

## CHAPTER 2

### PROGRAM MODE BASIC 21

- 2—1 OBJECTIVES 21
  - Learning Requirements of BASIC Programs 21
  - Telling the Computer What to Do 21
  - Correcting Mistakes 21
  - Learning to Store and Retrieve Programs 21
  - Recognizing the Importance of Organization and Style 21
- 2—2 DISCOVERY EXERCISES 21
- 2—3 DISCUSSION 28
  - Program Mode BASIC 28
  - Learning the Requirements of BASIC Programs 28
  - Telling the Computer What to Do 29
  - Correcting Mistakes 31
  - Learning to Store and Retrieve Programs 32
  - Recognizing the Importance of Organization and Style 32
- 2—4 PRACTICE TEST 35

## CHAPTER 3

### GRAPHICS AND EDITING 39

- 3—1 OBJECTIVES 39
  - Defining Function Keys 39
  - Accessing Graphics Mode 39
  - Learning to Draw and Name Shapes 39
  - Learning to Draw Circles and Rectangles 39
  - Using Proper Organization and Style 39
  - Using Editing Features 39
- 3—2 DISCOVERY EXERCISES 40
- 3—3 DISCUSSION 49
  - Defining Function Keys 49
  - Accessing Graphics Mode 50
  - Learning to Draw and Name Shapes 51
  - Learning to Draw Circles and Rectangles 52
  - Using Proper Organization and Style 53
  - Using Editing Features 53
- 3—4 PRACTICE TEST 54

**CHAPTER 4****SUBROUTINES AND TOP-DOWN ORGANIZATION 57**

- 4—1 OBJECTIVES 57
  - Packaging Statements into Subroutines 57
  - Using Top-Down Organization 57
  - Seeing Subroutine Bugs 57
  - Using the Top-Down Strategy 57
  - Working with Program Examples 57
- 4—2 DISCOVERY EXERCISES 58
- 4—3 DISCUSSION 64
  - Packaging Statements into Subroutines 64
  - Using Top-Down Organization 64
  - Seeing Subroutine Bugs 66
  - Using the Top-Down Strategy 68
- 4—4 WORKING WITH PROGRAM EXAMPLES 69
  - Example 1 - Plotting Big Letters 69
  - Example 2 - An Exploration 73
- 4—5 PROBLEMS 75
- 4—6 PRACTICE TEST 75

**CHAPTER 5****ARITHMETIC OPERATIONS AND PRECISION 77**

- 5—1 OBJECTIVES 77
  - Getting Numbers into a BASIC Program 77
  - Doing Arithmetic on the Computer 77
  - Using Parentheses in Computations 77
  - Learning E Notation for Numbers 77
  - Spacing the Printout 77
  - Working with Program Examples 78
- 5—2 DISCOVERY EXERCISES 78
- 5—3 DISCUSSION 88
  - Getting Numbers into a BASIC Program 88
  - Doing Arithmetic on the Computer 89
  - Using Parentheses in Computations 91
  - Learning E Notation for Numbers 92
  - Spacing the Printout 93
- 5—4 WORKING WITH PROGRAM EXAMPLES 94
  - Example 1 - Unit Prices 94
  - Example 2 - Converting Temperature 96
  - Example 3 - Sum and Product of Numbers 98
- 5—5 PROBLEMS 100



5—6 PRACTICE TEST 102

**CHAPTER 6**

**THE LOOP BLOCK 105**

6—1 OBJECTIVES 105

Creating an Infinite Loop Block 105

Writing a Counting Loop 105

Learning the Standard Loop Block 105

Using the FOR/NEXT Abbreviation 105

Working with Program Examples 105

6—2 DISCOVERY EXERCISES 105

6—3 DISCUSSION 111

Writing a Counting Loop 112

Learning the Standard Loop Block 113

Using the FOR/NEXT Abbreviation 115

6—4 WORKING WITH PROGRAM EXAMPLES 116

Example 1 - Finding the Average of a Group of Numbers 116

Example 2 - Depreciation Schedule 121

Example 3 - Table of Numbers and Their Cubes 125

6—5 PROBLEMS 126

6—6 PRACTICE TEST 127

**CHAPTER 7**

**THE BRANCH BLOCK 129**

7—1 OBJECTIVES 129

Learning the IF/THEN/ELSE Statement 129

Learning the Standard Branch Block 129

Learning an Abbreviated Branch Block 129

Learning a Case Block 129

Working with Programming Examples 129

7—2 DISCOVERY EXERCISES 130

7—3 DISCUSSION 134

Learning the IF/THEN/ELSE Statement 135

Learning the Standard Branch Block 135

Learning an Abbreviated Branch Block 136

Learning a Case Block 136

7—4 WORKING WITH PROGRAM EXAMPLES 137

Example 1 - Automobile License Fees 137

Example 2 - Averaging the Positive and Negative Numbers in a List 141

7—5 PROBLEMS 148

7—6 PRACTICE TEST 150

## CHAPTER 8 FUNCTIONS 153

- 8—1 OBJECTIVES 153
  - Seeing Functions in Action 153
  - Using Built-in String and Numeric Functions 153
  - Creating User-Defined Functions 153
  - Working with Program Examples 153
- 8—2 DISCOVERY EXERCISES 153
- 8—3 DISCUSSION 162
  - Creating User-Defined Functions 164
- 8—4 WORKING WITH PROGRAM EXAMPLES 165
  - Example 1 - Exact Division 165
  - Example 2 - String Reversal 167
  - Example 3 - Word Count 168
  - Example 4 - Graphbox 169
- 8—5 PROBLEMS 170
- 8—6 PRACTICE TEST 173

## CHAPTER 9 LISTS OF DATA 175

- 9—1 OBJECTIVES 175
  - Using Single- and Double-Subscripted Variables 175
  - Saving Space for Arrays 175
  - Using Subscripted Variables and **FOR/NEXT** Loops 175
  - Working with Program Examples 175
- 9—2 DISCOVERY EXERCISES 175
- 9—3 DISCUSSION 183
  - Using Single- and Double-Subscripted Variables 183
  - Saving Space for Arrays 184
  - Using Subscripted Variables and **FOR/NEXT** Loops 185
- 9—4 WORKING WITH PROGRAM EXAMPLES 185
  - Example 1 - Examination Grades 185
  - Example 2 - Course Grades 188
  - Example 3 - Array Operations 191
  - Example 4 - A Sales Chart as a String Array 192
- 9—5 PROBLEMS 195
- 9—6 PRACTICE TEST 201

WITHDRAWN  
FROM B'ham-Sou.  
College Lib.



## **CHAPTER 10**

### **SIMULATIONS WITH RANDOM NUMBERS 203**

- 10—1 OBJECTIVES 203
  - Understanding Random-Number Generators 203
  - Designing Sets of Random Numbers 203
  - Working with Program Examples 203
- 10—2 DISCOVERY EXERCISES 203
- 10—3 DISCUSSION 209
  - Understanding Random-Number Generators 209
  - Designing Sets of Random Numbers 209
- 10—4 WORKING WITH PROGRAM EXAMPLES 210
  - Example 1 - Dice Rolling Simulation 210
  - Example 2 - Phrase Generator 213
  - Example 3 - Random Walk 214
  - Example 4 - Circles 215
  - Example 5 - Birthday Pairs in a Crowd 216
  - Example - 6 Shuffle and Deal 217
- 10—5 PROBLEMS 219
- 10—6 PRACTICE TEST 220

## **CHAPTER 11**

### **FILES 221**

- 11—1 OBJECTIVES 221
  - Opening and Closing Buffers 221
  - Storing Information to a File 221
  - Retrieving Information from a File 221
  - Modifying Information Stored in Files 221
  - Working with Program Examples 221
- 11—2 DISCOVERY EXERCISES 221
- 11—3 DISCUSSION 228
  - Opening and Closing Buffers 228
  - Storing Information to a File 229
  - Retrieving Information from a File 229
  - Designing Record Structures with Field Widths 230
- 11—4 WORKING WITH PROGRAM EXAMPLES 231
  - Example 1 - Mail List Data Entry 231
  - Example 2 - Mailing Label Program 234
  - Example 3 - Selected Labels Program 235
  - Example 4 - Modifying the MAILLIST File 235
- 11—5 PROBLEMS 240
- 11—6 PRACTICE TEST 241

<b>CHAPTER 12</b>	
<b>GRAPHICS REVISITED</b>	<b>243</b>
12—1 OBJECTIVES	243
Creating and Using Graphics Regions	243
Learning to Draw Arcs	243
Learning to Use Arrays in Graphics	243
Labeling Displays	243
Working with Program Examples	243
12—2 DISCOVERY EXERCISES	243
12—3 DISCUSSION	253
Creating and Using Graphics Regions	253
Learning to Draw Arcs	255
Learning to Use Arrays in Graphics	255
Labeling Displays	256
12—4 WORKING WITH PROGRAM EXAMPLES	258
Example 1 - Budget Bar Chart	258
Example 2 - Budget Pie Chart	262
Example 3 - Function Plotting	264
Example 4 - Zooming	265
Example 5 - House Design	265
12—5 PROBLEMS	268
12—6 PRACTICE TEST	268
<b>APPENDIX A</b>	
<b>COMPATIBLE COMPUTERS</b>	<b>271</b>
<b>APPENDIX B</b>	
<b>PRACTICE TEST SOLUTIONS</b>	<b>273</b>
<b>APPENDIX C</b>	
<b>SOLUTIONS TO ODD-NUMBERED PROBLEMS</b>	<b>283</b>
<b>APPENDIX D</b>	
<b>GLOSSARY</b>	<b>303</b>
<b>INDEX</b>	<b>307</b>



# PREFACE

*Structured BASIC for the IBM Personal Computer* teaches modern programming methods using BASIC and the IBM Personal Computer. This book grew out of several earlier works published by McGraw-Hill Book Company. The book, however, is not a mere reworking of those earlier works. The authors have expanded and completely revised the material in previous "Hands-On" BASIC books to emphasize the importance of the top-down structured programming using only the three fundamental blocks. Even the simplest programs in this book are written in top-down fashion and use only the action, loop, and branch blocks. This modern approach produces programs that are easy to write, easy to understand, and easy to change. In addition to providing a comprehensive coverage of the BASIC language, an in-depth exploration and discussion of graphics and files is included.

This book, like the programs in it, is meant to be easy to understand. To this end, the authors have followed the common wisdom that experience is the best teacher. For this reason, the approach is inductive: Experimentation precedes formulation of principles. The reader spends time exploring the effect of taking certain actions on the computer before reading traditional material that formalizes what he or she has already experienced.

As a result, the structure of the book is fundamentally different from that of most computer language books. Each chapter begins with a statement of the objectives, then the reader is guided through a set of experiments that show the BASIC language in action. Only then does the reader move to a traditional treatment of the concepts learned. After this, the reader is given an opportunity to apply those concepts, either through problems or through programs developed step by step from a general description of the task to be performed.

The text is organized into an introduction, twelve chapters, and appendices. Each chapter forms a module of instruction requiring about one or two hours of computer work and one or two hours of text study. Review tests at the end of each chapter, let the reader test mastery of objectives.

The book can be used in several different ways. First, and probably most importantly, it can be used with no supervision as a self-study text. This approach has also been used very effectively in an open-entry, open-exit, self-paced course. In addition, the material can be presented in a traditional lecture format. The material is accessible to, and has been used by, students from junior high through graduate school. The mathematics level has intentionally been kept very low to maintain a sharp focus on the programming issues. At the level presented, nearly

anyone should be able to work through the material without getting “hung up” by the mathematics.

Two documents furnished with the IBM Personal Computer are not required, but are useful as additional references along with *Structured BASIC for the IBM Personal Computer*: the *BASIC* reference manual that lists all of the specifications and capabilities of BASIC as implemented on the IBM Personal Computer, and the *DOS* reference manual that describes some additional capabilities of the IBM Personal Computer.

The reader must have access to an IBM Personal Computer with the Color/Graphics Monitor Adapter, at least one disk drive, the Disk Operating System (DOS) diskette, and either a black and white monitor/television set or a color monitor/television set. This book can also be used with several other computers compatible with the the IBM Personal Computer including Compaq, AT&T Personal Computer, and the IBM PCjr (see Appendix A).

Comments or suggestions for improving this book will be appreciated.

### Acknowledgments

In 1981, our colleague Authur Luehrmann first proposed the structured approach to BASIC programming used throughout this book. During the next two years, Luehrmann and Peckham fully developed the method. They first presented it publicly in their textbook *Computer Literacy—A Hands-On Approach* (McGraw-Hill) and later used it in the programming manual *Hands-On Basic for the IBM PCjr* (McGraw-Hill and IBM). The present book is indebted to the pioneering concepts and methods used in those works.

The pages of this book were composed and typeset by the T<sub>E</sub>X software system, developed by Donald E. Knuth at Stanford University. We wish to thank David Fuchs and Dikran Karagueuzian of Stanford University for getting the necessary fonts in place and producing the final output. In addition, our thanks go to Antonio Padial for his fine job of copy editing. Many thanks also go to our wives Ann, Jane, and Rose Marie for their patience and support during the production of this book.

Herbert Peckham  
Wade Ellis, Jr.  
Ed Lodi



# INTRODUCTION

## THE PC DOS OPERATING SYSTEM

### The Parts of Your Computer

Your computer system consists of several connected parts. The most familiar parts are the keyboard, the screen or monitor, and the disk drive(s). Another part is the central processing unit (CPU), which is a small but powerful electronic component or chip inside the system cabinet. The CPU controls the keyboard, monitor, disk drives, and other devices you connect to your computer system.

### What Is an Operating System?

An operating system is a set of instructions the computer follows to manage the system and the peripherals. Because of the operating system, you do not have to speak the computer's fundamental language, represented by 0s and 1s. This makes computers much easier to use, and thus many more people are learning to use them. Many different operating systems have been developed because various types of computer activities require different features. Recently, operating systems have been developed to meet the special needs of small computer users. One of these operating systems is PC DOS.

### The PC DOS Operating System

PC DOS is the name of an operating system developed by the Microsoft Company for IBM personal computers. The name stands for Personal Computer Disk Operating System. This book is written for use with PC DOS and PC DOS-compatible versions of BASIC. PC DOS manages the devices in your system, which *must* include a main unit with at least one disk drive and color graphics capability. The display device can be either a black-and-white, or for best results, a color monitor. You can also use a black-and-white or color TV set.

Many powerful features of PC DOS help you use your computer. The BASIC environment, or mode, is accessed through PC DOS. In addition, PC DOS has many advanced features that allow you to check the status of your system and the peripheral devices attached to it. Among these features are the time and date, directory, and diskette formatting utilities. The time and date utilities allow you to check the clock and calendar maintained by PC DOS. The directory utility allows you to list what is stored on a diskette. The formatting utility allows you to prepare a diskette for use with your system.

## Program Transportability

A program is a set of instructions for a computer. It would be nice if different computers could follow the same set of instructions, or programs. This is true for computers using PC DOS (or PC DOS-equivalent operating systems) and BASIC provided only the standard features of BASIC are used in the program. Unfortunately, some PC DOS-equivalent versions of BASIC do not include the advanced graphics features you will learn in Chapters 3 and 12. Consult Appendix A for details if you are using a computer other than an IBM personal computer.

Ideally, you should be able to use the diskette with your BASIC program on a computer other than an IBM personal computer. Unfortunately, the format used to store the program on your diskette may not be compatible with the format another computer accepts. Even if the disk format is the same, the other computer system may not have the necessary peripheral devices to use your program. Worse yet, the versions of BASIC may not be exactly the same. Nonetheless, PC DOS and BASIC can provide a significant improvement in program transportability.

## Learning to Access BASIC from PC DOS


You will be using BASIC throughout this book. You can easily access BASIC once you have started your computer with the PC DOS operating system. The following exercises show you how to do this. Occasionally, you will be asked to write down an answer to a question. Take the time to do this.

## COMPUTER ACTIVITIES

1. Insert the diskette marked DOS in the left disk drive (drive A) and close the door. (The label on the diskette should be on the top side and on the edge closest to you as you insert the diskette.)
2. Turn on the monitor. Use the on-off switch to turn on the computer.
3. After about 30 seconds, the red light on the left disk drive will come on, and you should see a request similar to this one:


**Current date is Tue 1-01-1980**

**Enter new date:**

Type the current date in the format 5-23-1985 and then press , the large key with the curved arrow, at the right of the keyboard. The computer should respond with


**Current time: 0:00:25.43**


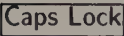
**Enter new time:**

Press  to accept this time as the current time. You will change it later. The computer will respond with several messages, including a copywrite message, and


**A>**

The prompt (A>) tells you that you are in the PC DOS operating system.

4. You can tell the operating system to do many things when this prompt is displayed. Often you will be asked to type something and press . This action sends what

you type to the operating system for processing. The word **enter** will be used to describe this action of typing something and pressing . Press the key marked  and enter

**DIR**

Did you remember to press  ?


---

How many files were listed on the screen?

---

Beside each file name is listed its size and the date and time it was stored. Now enter


**DIR/W**

The key with / on it also has the question mark (?) on it. Remember to press . How is the format of the directory listing changed?

---

## 5. Enter


**TIME**

and remember to press . Although you can enter the hour, the minutes, the seconds, and the tenths of seconds in setting the current time, this is not necessary. For now, enter the hour and minutes in the format shown. (14:05 for 2:05 pm, for example). Enter

**TIME**


again to see if the change has been made in the current time. How many seconds have passed since you changed the time?

---

Press  to accept this time as the correct time.

6. The DOS system diskette should be used only to make working copies of the operating system. To prepare a working diskette for use with your computer, do one of the following: If your system has two disk drives, place a blank diskette in the right drive (drive B), enter

**DISKCOPY A: B:**

and press . (Note the space between **A:** and **B:**). Strike any key as indicated on the screen to begin the copy process. This process should take about one minute. Go to step 7.



If your system has a single disk drive, enter

### DISKCOPY

and remember to press . Since the diskette to be copied (source diskette) is already in drive A, press  as indicated on the screen. Remove the DOS diskette and insert a blank diskette (target diskette) in its place as indicated on the screen. Press . Continue to swap the target diskette and source diskette, following the instructions on the screen, until the copy process is completed. This should take about 90 seconds.

7. This process ends when you are asked if you wish to make another copy. Press  in answer to the question **Copy another (Y/N)?**
8. Be sure the working copy of the operating system (the one you just made) is in the left drive (drive A). Place the original PC DOS system diskette back in its protective envelope.
9. To access BASIC, enter

### BASICA

Notice the A appended to BASIC. BASICA is the advanced version of BASIC you will be using throughout the book. (For some IBM PC compatibles, BASICA is called GWBASIC. See Appendix A for details.) Several messages will appear on the screen. Then

**Ok**

—

The **Ok** prompt indicates that you are in BASIC. The flashing underline on the screen is the **cursor**. The cursor shows you where the next characters you type will appear on the screen.

10. Enter

### FILES

Is the format the same as it is with **DIR/W** or **DIR**?

---

Notice the periods in this format.

11. To return to the DOS operating system, enter

### SYSTEM

Which prompt is now displayed?

---

Enter

**DIR/W**

Are there periods in this format?

---

12. Enter

**FILES**

What was displayed?

---

At times, you will be asked to enter something that will result in an error. It is important to become familiar with errors that can occur when you use your computer. Here, you entered **FILES** in the PC DOS operating system rather than in BASICA, where it is allowed.

13. This ends the computer exercises. Whenever you are asked to turn off the computer, you should first return to the PC DOS operating system level where you can see the **A>** prompt (see step 12), remove the diskette(s), and switch the computer off.
14. Turn off the computer and the monitor (or the TV set if you are using one).

## SUMMARY

You have learned to recognize the PC DOS operating system prompt (**A>**) and the BASICA prompt (**0k**). You have also learned how to access BASICA and how to return to the DOS operating system from BASICA. Finally, you have learned how to turn the computer on and off correctly.



# DIRECT MODE BASIC

## 1—1 OBJECTIVES

### Reviewing the Start-up Procedure

You will review how to turn on your computer and learn a standard procedure for accessing BASIC.

### Learning to Use PRINT with Strings and Numbers

You can use the **PRINT** statement to display strings and numbers. You will learn how to do this.

### Using the PRINT Statement to Perform Simple Operations

You will learn how to use the **PRINT** statement to carry out arithmetic operations on numbers and to join strings together.

### Using Variable Names in BASIC

Variables are an important part of all programming languages. You will name variables and learn how to assign string or numeric values to them in BASIC.

### Correcting Mistakes

You will learn how to correct typing errors.

## 1—2 DISCOVERY EXERCISES

In the Introduction, you learned to boot up PC DOS by turning on the computer with the DOS system diskette in the left drive. You then used the PC DOS operating system to create a working diskette. Next, you'll learn a standard procedure to access BASIC using your working diskette. This procedure is called *bringing up BASICA*. If you have not created a working diskette, do so now. (See step 6 in the last section of the Introduction if you have forgotten how to do this.)

**Bringing up BASICA**

1. Place your working diskette in the left drive and turn on the computer. If necessary, turn on your monitor. Press the **Caps Lock** key. Recall that entering information requires that you first type whatever is to be entered and then press the key marked **↵**. Enter the correct date. Enter the correct time. At the PC DOS prompt (**A>**), enter BASICA. Look at the monitor. Is the **A>** or **Ok** prompt displayed?

---

If the **A>** prompt is displayed, you have typed BASICA incorrectly. If this happened, enter BASICA again. If you make typing errors, simply retype the line correctly and press **↵**.

**The PRINT Statement**

2. Remembering to press **↵** after you have typed the line, enter

```
PRINT "AWAY WE GO!"
```

Be sure to type the quotation marks. What happened?

- 
3. Enter

```
PRINT "COMPUTING IS FUN."
```

What happened?

- 
4. Locate the **Alt** key on the lower left side of the keyboard. Hold it down and press **P**. What happened?

---

Now type

```
"COMPUTING IS EASY."
```

Again, note the quotation marks. What happened?

---

If you did not press **↵**, do so now. Remember that typing is different from entering.

5. You may wish to use the **Alt** and **P** keys as a shorthand way to type **PRINT**. Enter

```
PRINT 2 + 3
```

What happened?

---

6. Now enter

```
PRINT "2 + 3"
```

Notice the quotation marks. How did these marks change the results?

---

7. When the screen gets cluttered, you need a way to erase unwanted information. To clear the screen, locate the **Ctrl** key at the left of the keyboard. Hold this key down and press the **7** key at the right of the keyboard (The **7** key also has Home on it.) What happened?
- 

You can also clear the screen by entering **CLS**.

8. Press the key marked **↔**. When you press this key, the cursor should move to the right. If the number 6 is displayed on the screen instead, press the **Num-Lock** key. Press **↔**.
9. Now type, but do not enter,

```
PRINT "JELLO"
```

Locate the key marked with **←**. This key is called the left-arrow key. Press this key several times until the cursor is under the **J**. Press the **H** key. What happened?

---

Enter the line by pressing **↔**. You may also use **↩** (the key on the top row with a left arrow on it) to correct typing errors.

### Simple Operations

10. Enter

```
PRINT 5.2 + 3
```

What number was displayed?

---

11. Enter

```
PRINT 2 + 6 - 3
```

Record the output below.

---

12. Enter

```
PRINT 12 / 2
```

What arithmetic operation does the / call for?

---

13. Enter

```
PRINT 2 * 50
```

(The \* is on the top row of the keyboard, above 8.) What arithmetic operation does the \* call for?

---

14. Clear the screen (see step 7). Enter




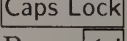

```
PRINT "DOG" + "HOUSE"
```

What did the computer print on the screen?

---

How many spaces were between the two words or strings of characters?

---

Press the  key (the key with an up arrow and an 8 on it) until the cursor is under the P in PRINT. Use  (the key with the right arrow and a 6 on it) until the cursor is under the " following G in DOG. Press the  key (next to the  key). Now press the spacebar to insert a space between the G and the ". Press  to enter the line. What happened?


---

The PRINT statement displays exactly the characters enclosed in quotation marks. The + operator joins the strings on each side of the operator.



## 15. Type

```
PRINT "22" + "33"
```

What do you think will happen when you press  ?


---

Try it and see if you were right. Is **22** a number or a string of characters?

---

## 16. Type

```
PRINT 22 + 33
```

Now what do you think will happen if you press  ?

---

Try it and see if you were right.

**Variable Names**

## 17. Clear the screen (see step 7). Enter

```
LET TEST$ = "A"
```

Notice the \$ after **TEST**. What happened?

---

The **LET** statement assigns a value to **TEST\$**, which is the name of a string variable. This value is whatever is between the quotation marks following the = sign. In this case, the name **TEST\$** has been assigned the value **A**.

## 18. Enter

```
PRINT TEST$
```

What value was displayed?

---

Now enter

```
LET TEST$ = "B"  
PRINT TEST$
```

What happened?

---

You changed the value of **TEST\$** when you assigned the value **B** to it.



## 19. Enter

```
LET TEST = 95
```

If a variable name ends with a dollar sign it names a string variable. If there is no dollar sign, the variable named has a numeric value. Therefore, **TEST** is a numeric variable. What numeric value do you think was assigned to **TEST**?

---

Enter

```
PRINT TEST
```

What numeric value was displayed?

---

## 20. Enter

```
LET TEST$ = 95
```

What happened?

---

**TEST\$** is a string variable and must be assigned a string of characters enclosed in quotes.

## 21. Take a few moments to examine the lines below:

```
LET LENGTH = 10  
LET DEPTH = 6  
LET HEIGHT = 4  
LET VOLUME = LENGTH * DEPTH * HEIGHT  
PRINT VOLUME
```

What do you think will happen if you enter these lines?

---

Now enter the lines. What happened?

---

22. Study the lines below briefly.

```
LET LENGTH = 12
LET DEPTH = 9
LET SQYDS = (LENGTH * DEPTH) / 9
PRINT SQYDS, "SQYDS"
```

What will happen if you enter these lines?

---

Clear the screen and enter the lines. What happened?

---

23. This concludes the discovery material for this chapter. Turn off your computer. (See step 13 of the last section of the Introduction if you have forgotten the steps.)


### 1—3 DISCUSSION

#### Reviewing Start-up Procedure

Each time you use the computer you will need to bring up BASICA. To do this, insert a working diskette in the left drive and turn on the computer. When prompted, enter the date and time. When you see the **A>** prompt, you know that the computer is in the PC DOS operating system. Then enter BASICA. When you see the **Ok** prompt on the screen, the computer is in BASIC.

When you finish using the computer, return to the PC DOS operating system by entering the **SYSTEM** command. Once again, you should see the **A>** prompt. Remove the diskette(s) and turn off the computer. This procedure for turning off the computer helps to prevent loss of information.

#### Direct-Mode BASIC

In the Discovery Exercises, you gave BASIC instructions to the computer in the **direct mode**. In this mode, statements like **PRINT** and **LET** are performed by the computer as soon as you press the  key. Therefore, you can see their effect as soon as the statements are entered. In the next chapter, you will learn about the **program mode**. In that mode, the statements are not performed until you tell the computer to do so.

#### Learning to Use PRINT with Strings and Numbers

The **PRINT** statement has several uses. You have used it to display strings of characters and numbers. For example,

```
PRINT "GETTING STARTED IS EASY."
```

tells the computer to display the string of characters between the quotation marks. Any characters inside quotation marks after **PRINT** are printed verbatim on the screen.

On the other hand,

```
PRINT 5 + 8
```

tells the computer to first add 5 and 8 together and then display the sum on the screen. If the characters after **PRINT** are *not* enclosed in quotation marks, and if they define a valid arithmetic operation, the computer does the arithmetic and prints the result.

Recall that you can use the **Alt** and **P** keys as a shorthand way to type **PRINT**. (Since this takes two keystrokes and only five are needed to type **PRINT**, you may not want to bother if you are a good typist.) The **Alt** key can be used to type other BASIC words as well. These will be introduced as the new BASIC words arise.

Recall also that you can clear the screen by holding down **Ctrl** and pressing **Home**, or by entering **CLS**.

- Use **Alt** and **P** to type **PRINT**.
- Use **Ctrl** and **Home** to clear the screen.
- Enter **CLS** to clear the screen.

### Using the PRINT Statement to Perform Simple Operations

In the Discovery Exercises, you used numeric operations to add, subtract, divide, and multiply numbers. The numeric operations are indicated by the following symbols:

+	<i>Addition</i>
-	<i>Subtraction</i>
/	<i>Division</i>
*	<i>Multiplication</i>

For example,

```
PRINT 5 + 3 - 6
```

tells the computer to display the number 2. While

```
PRINT 7 * 5
```

causes the number 35 to be displayed.

You also used a string operator to join (or concatenate) two strings of characters. This operator is indicated by the symbol **+**. For example,

```
PRINT "HELLO" + "THERE"
```

causes the two strings to be joined. Then

**HELLO THERE**

is displayed on the screen. Notice that there is no space between the two words. A space is like any other character. If you want a space between two strings joined together with the + operator, the space must be part of the string of characters.

Also, digits like **2** and **5** are characters if they are contained in a character string bounded by quotation marks. For example, if you enter

```
PRINT "24 " + " 57"
```

the computer will display

```
24  57
```

on the screen. Notice that there are two spaces between **24** and **57**. The first space came from the space after **24** in the **PRINT** statement. The second space came from the space before **57**.

Also,

```
PRINT "24 + 57"
```

tells the computer to display

```
24 + 57
```

where the digits **2**, **4**, **5**, and **7** along with the + character and two spaces are displayed exactly as they appear between the quotation marks.

**Using Variable Names in BASIC**

Variables are places in memory where the computer stores information. These places must have names. One way to name variables is with the **LET** statement. For example,

```
LET A$ = "GOOD"
```

names a string variable **A\$** and assigns to it the value **GOOD** (a character string). Consider the statements

```
LET A$ = "BETTER"  
LET A$ = "BEST"
```

The first assignment statement assigns **BETTER** to **A\$**. The second assignment statement then reassigns **A\$** the value **BEST**.

So far, you have seen two types of variables in BASIC, string variables and numeric variables. A string variable name always ends with a **\$** while a numeric variable does not. For example, the lines

```
LET A = 5
PRINT A
```

assign the value 5 to the numeric variable **A** and then print the value of **A**. As a result, 5 will be displayed on the screen when these lines are entered.

If you tell the computer to print the value of a numeric variable that has not yet been assigned a value, the computer will display the value 0. For example, if **TAX** is a numeric variable that has not been assigned a value, then the direct-mode statement

```
PRINT TAX
```

will cause 0 to be displayed on the screen.

If you tell the computer to print the value of an unassigned string variable, the computer will display the **null string** (the string with no characters in it). For example, if **EMPTY\$** is a string variable that has not been assigned a value, then the statement

```
PRINT EMPTY$
```

results in a blank line being displayed on the screen. The only thing that happens is that the cursor moves down one line. No characters are written on the screen.

Every variable has associated with it a name, a type, and a value. For example, the statement

```
LET A$ = "GREETINGS"
```







names **A\$** as a string variable and assigns to it the string **GREETINGS**. The statement




```
LET A = 32
```

names **A** as a numeric variable with assigned value 32.

- Assigned variables have a *name*, a *type*, and a *value*.

### Correcting Mistakes

If you are typing a line and detect an error before pressing  to enter the line, you can use  to correct your mistake.  is on the top row of the keyboard. The only marking on the key is a left arrow. This key erases the character to the left of the cursor. One character is erased each time you press . You can then correct the mistake, retype the balance of the line, and enter the corrected line by pressing . If you hold down the  key, characters are continually erased until you release the key or reach the leftmost position on the screen.

You can also correct mistakes using the arrow keys and  and . The arrow keys are used to place the cursor at the error. The arrow keys (those with the numbers 2, 4, 6, and 8 above them) do not erase characters as they pass over them. Pressing  puts the computer in the insert mode, which allows you to insert characters to the left of the cursor. A square blinking cursor indicates that you are



in the insert mode. Pressing **Ins** a second time terminates the insert mode. The **Del** key deletes or erases the character under the cursor and moves the character to the right of the cursor to the position under the cursor. If you hold this key down, characters to the right of the cursor are removed until you release the key.

- Use **↑**, **↓**, **→**, and **←** to move the cursor about the screen.
- Use **←** to erase typing errors.
- Use **Ins** and **Del** to insert and delete characters.

Use these editing features of BASICA to correct errors and make changes. You may wish to experiment with these editing features now since you will use them extensively in future chapters.

### 1—4 PRACTICE TEST

Take the test below to discover how well you have learned the objectives of this chapter. The answers to this and all subsequent practice tests are given in Appendix B.

1. How do you let the computer know that you are through typing a line?

---

2. How do you enter the following statement?

**PRINT AUTHOR\$**

---

3. What symbol indicates multiplication on the computer?

---

4. How do you clear the screen display?


---

5. What operation does the symbol / indicate?

---

6. What will happen if you type


```
PRINT 3 * 4 / 6
```

and then press ?

---

7. What will happen if you type


```
PRINT "25 / 5 + 2"
```

and then press ?

---

8. If you typed

```
PRING 2 + 3 * 4
```

and notice the spelling error before you pressed , how could you correct the error?

---

9. What will the computer display if the following statements are entered?

```
LET QUIZ = 8  
LET QUIZ = 10  
PRINT QUIZ
```

---

# PROGRAM MODE BASIC

## 2—1 OBJECTIVES

### Learning Requirements of BASIC Programs

All BASIC programs have common characteristics. You will enter and run some very simple programs to learn about these characteristics.

### Telling the Computer What to Do

Commands are action words that tell the computer to do something to or with a BASIC program. These action words are used to control a program. You will look at the following commands: **LIST**, **RUN**, **NEW**, **RENUM**, and **AUTO**.

### Correcting Mistakes

You will enhance your ability to correct errors, modify program statements, and learn to use the **EDIT** command.

### Learning to Store and Retrieve Programs

You need to know the commands that permit you to store programs on and retrieve them from a diskette. You will learn these and other program management commands.

### Recognizing the Importance of Organization and Style

You will begin to learn a writing style that uses remark statements and indentation. This style helps you create programs that are easy to read, understand, and modify.

## 2—2 DISCOVERY EXERCISES

If you are still using the DOS System diskette then you will need to create a working diskette to complete the Discovery Exercises. You can find instructions for creating a working diskette in step 6 of the computer work in the Introduction.



1. Turn on the computer and bring up BASICA. Remember to press the Caps Lock key.
2. Recall that *enter* means to type the line and press ↵. Also, if you don't want to type **PRINT**, hold down Alt, press P, and the computer will type **PRINT** for you. Enter

```
10 PRINT "A COMPUTER WILL DO AS YOU SAY"
```

This is the first line of a BASIC program.

3. Remembering to press ↵ after each line, enter the balance of the program as listed below:

```
20 PRINT "IF YOU LEARN THE CORRECT WAY"
30 PRINT "IT DOES EXACTLY AS IT'S TOLD"
40 PRINT "NO MATTER HOW FUNNY OR BOLD"
50 END
```

If you make a mistake before you finish typing a line, correct it using the methods learned in Chapter 1. If you notice a mistake after you have entered it, do not try to return to that line. Simply press ↵ and retype it for now. The new line will supersede and cancel the old one.

4. Type **CLS** and press ↵. What happened?  
\_\_\_\_\_
5. Fortunately, all is not lost. The computer remembers what you entered even though your program has been removed from the screen. Type **LIST** and press ↵. What happened?  
\_\_\_\_\_
6. The program you just entered should appear on the screen. If the computer is told to carry out the instructions in this program, what do you think will happen?  
\_\_\_\_\_





Type **RUN** and press ↵. What happened?  
\_\_\_\_\_

7. Type **EDIT 40** and press ↵. What happened?  
\_\_\_\_\_

Replace **40** with **25** by simply typing **25**. Notice that **25** replaces **40** in the line. Press ↵ to enter the line. Display the program by entering **LIST**. Where is line 25 located?  
\_\_\_\_\_



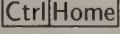
What happened to line 40?


---

8. To delete line 40, type **40** and press . Display the program (see step 7) to see that line 40 has been deleted.
  9. Let's run the program. Look at the bottom of the screen and locate **2 RUN<--**. If you press the key marked  at the top left of the keyboard, the effect will be the same as typing **RUN** and pressing . Press . What happened?
- 

In what way has the poem changed?



---



10. Another way to clear the screen is to hold down the key marked  at the left of the keyboard and press the  key at the right of the keyboard. This action will be referred to as . Two-key sequences will be referred to in this way. Use this method to clear the screen now. What are the first two words at the bottom of the screen?
- 

11. Notice that **RUN** has a left arrow after it while **LIST** does not. Press the key marked . What happened?
- 


Press . What happened?

---

You can use  and  to list and run programs.

12. Use the arrow keys to move the cursor up to the **P** in **PRINT** in line 10. Delete the word **PRINT** using . Use  to insert **LET A\$ =** so that line 10 looks like



**10 LET A\$ = "A COMPUTER WILL DO AS YOU SAY"**

Press  to enter the line. Where is the cursor now?

---

13. The cursor is on line 20. Modify line 20 as follows (see step 12):

**20 LET B\$ = "IF YOU LEARN THE CORRECT WAY"**

Press  to enter the line. Modify lines 25 and 30 as follows, remembering to press  to enter lines:

```

25 LET C$ = "NO MATTER HOW FUNNY OR BOLD"
30 LET D$ = "IT DOES EXACTLY AS IT'S TOLD"

```

Where is the cursor?

---

Clear the screen using **Ctrl|Home** (see step 10).

14. Use **F1** to display the program (see step 11). Run the program using **F2**. What happened?
- 

Enter the direct-mode statement

```
PRINT A$
```

What happened?

---

15. To have the computer display the lines, the program needs **PRINT** statements. Type

```
AUTO 41, 2
```

and press **↵**. What happened?

---

The cursor is next to line number 41. Using **Alt|P**, type

```
PRINT A$
```

and press **↵**. What is the next line number displayed?

---

You are in the automatic line-numbering mode. You can exit this mode by holding down **Ctrl** and pressing **Break** (the key with Scroll Lock on it) at the upper right of the keyboard.

16. The cursor is next to line number 43. Type

```
PRINT B$
```

and press **↵**. Then type the following beside the line numbers displayed, remembering to press **↵** at the end of each line.

```
PRINT C$
PRINT D$
```

When line number 49 is displayed, hold down **Ctrl** and press **Break** to exit the automatic line-numbering mode.

17. Clear the screen and use **F1** to display the program. Use **F2** to run the program. What happened?
- 

18. Type **RENUM** and press **↵**. Display the program. What happened to line 41?
- 

Enter

```
RENUM 100
```

Display the program. What is the smallest line number?

---

19. Now enter

```
RENUM 100, , 20
```

Notice the two commas. Display the program. What is the difference (or increment) between line numbers?

---

20. Enter

```
80 INPUT AUTHOR$
250 PRINT
260 PRINT "BY " + AUTHOR$
```

Display the program. Run the program. What happened?

---

Type your name and press **↵**. Who wrote the poem?

---

The plus sign between **"BY "** and **AUTHOR\$** tells the computer to join the two string values together.

**Program Storage and Retrieval**

- 21.** For the next four steps, you must have your working diskette in the left drive (see step 6 of the Introduction). (If you do not have a working diskette, go to step 25.) Look at the bottom of the screen and locate **3 LOAD**" and **4 SAVE**". Notice the quotation marks. Press **F4**. What happened?
- 

Type **POEM**" and press **↵**. Did the disk drive light go on?

---

- 22.** Type **FILES** and press **↵**. Is there a file with **POEM** in its name?
- 

What has been added to the name of the program you just saved to diskette?

---

- 23.** Display the program. Type **NEW** and press **↵**. Again, display the program in memory. What happened?
- 

**NEW** clears the memory where the programs you enter reside.

- 24.** Let's retrieve the program from diskette to memory. Press **F3**, type

**POEM**"

and press **↵**. How long did the disk drive whirr?

---

Display the program to see that it is in memory. You have used **NEW** to clear the program in memory. Now you will delete the program from diskette. Type

**KILL "POEM.BAS"**

and press **↵**. Now enter

**FILES**

Is the BASIC (BAS) file **POEM** listed?

---

Do you still have a copy of the program? If so, where?

---

Display the program in memory. If you were to enter **NEW**, would you have a copy of **POEM** anywhere? Do *not* enter **NEW** at this time.



## Organization and Style

25. Add the following line to the program in memory.

```
60  'PROGRAM POEM
```

Be sure to type in the apostrophe ('). Display the program. Run the program. Notice that the new line 60 has no effect on what was displayed. Line 60 is an example of a **remark statement**. Remark statements are useful in naming a program and describing the actions performed by it. An apostrophe (') begins a remark statement.

26. Renumber the lines by entering

```
RENUM 100, , 20
```

Add the following statements. Notice that there are three spaces before the apostrophe (') in the first two statements.

```
130  'CREATE POEM LINES
210  'DISPLAY THE POEM
330  END
```

Display the program. Run the program. What happened?

---

27. Display the program. Notice the lines are misaligned. Organization and style are important in good programming. You have used the remark statement to indicate the organization of the program. You will now use an indentation style to make the organization easier to see. Use the arrow keys to move to the I in the **INPUT** statement. Press Ins and press the spacebar three times to insert three spaces. Press ↵ to enter the line.
28. Use the arrow keys to move to the L in **LET** in line 140 and insert five spaces at that point. Remember to press ↵ to enter the line. Insert five spaces in the remaining **LET** statements and in all of the **PRINT** statements. Renumber the program by entering

```
RENUM 100
```

Display the program. Your program should look like

```
100  'PROGRAM POEM
110      INPUT AUTHOR$
120  'CREATE POEM LINES
130      LET A$ = "A COMPUTER WILL DO AS YOU SAY"
140      LET B$ = "IF YOU LEARN THE CORRECT WAY"
150      LET C$ = "NO MATTER HOW FUNNY OR BOLD"
160      LET D$ = "IT DOES EXACTLY AS IT'S TOLD"
170  'DISPLAY THE POEM
180      PRINT A$
```

```

190      PRINT B$
200      PRINT C$
210      PRINT D$
220      PRINT
230      PRINT "BY " + AUTHOR$
240      END

```

29. A further improvement can be made in the program. Edit line 110 as follows:

```

110      INPUT "AUTHOR "; AUTHOR$

```



Display the program. Run the program. Follow the instructions given by the program. Since instructions are given by the program about what to enter, the program is easier to use. For instance, **ENTER AUTHOR** is a better message than **AUTHOR**. What would be an even better message in line 110?

---

30. Turn off the computer and go on to the next section.

## 2—3 DISCUSSION

### Program Mode BASIC

In Chapter 1, you learned how to enter BASIC statements in the direct mode. The computer performed each statement as soon as you pressed the  key after entering it. In the Discovery Exercises in this chapter you learned how to enter BASIC statements in the program mode. If a BASIC statement begins with a line number, the computer does *not* perform the statement when you press . Instead, the computer stores all the statements and performs them all when you enter the **RUN** command.

### Learning the Requirements of BASIC Programs

You have seen several important facts about BASIC programs demonstrated. As a point of reference, you will use the program below, which adds the numbers assigned to **A** and **B**, then prints the result **C**.

```

100  LET A = 1
110  LET B = 8
120  LET C = A + B
130  PRINT C
140  END

```

A BASIC program consists of one or more lines. Each line begins with a **line number** followed by a **BASIC statement**. In the program above, there are three types of BASIC statements: **LET**, **PRINT**, and **END**. The way the first two statements work was treated in the previous chapter. The **END** statement marks the end of the main part of the program. In Chapter 4, you will see why the **END** statement is needed.

- **A program line consists of a line number followed by a BASIC statement.**

Generally, the line numbers in a BASIC program are not numbered consecutively (such as 100, 101, 102, etc.). The reason is that often programmers need to insert additional lines later if they discover errors or want to modify the program. If the lines were numbered consecutively (spaced 1 apart), it would be less convenient to make changes. If you leave gaps in the line numbers, you can insert statements by simply typing in the new statements using line numbers not already in the program.

In BASIC, you can enter program lines in any order. If, for example, you type

```
140  END
120  LET C = A + B
110  LET B = 8
130  PRINT C
100  LET A = 1
```

and list this new program, the computer will sort out the statements and display them in numeric order. In the same way, if you ran the program as entered above, the computer would first arrange the statements into numeric order before performing them.

You can tell the computer to generate sequential line numbers automatically with the **AUTO** command. You can type **AUTO** or use **AltA** to display the command **AUTO** on the screen. For example,

```
AUTO 100, ,5
```

generates a new line number each time you press **↵**. The line numbers start at 100 and are 5 apart.

You can remove a BASIC statement from a program by typing the line number and pressing **↵**. You can modify statements by retyping the lines involved, pressing **↵** after each line is typed, or by using the **EDIT** command. As indicated above, you can add statements by using line numbers not already in the program. Thus, you can add, remove, or change BASIC statements as you wish.

## Telling the Computer What to Do

There is an important distinction between the statements in a BASIC program and BASIC commands. Commands tell the computer to do something with a program. You have seen how several commands work. Here is a brief review of the use of each:

**LIST:** When you enter a BASIC program, it goes into memory. Quite often, you need to see the program contained in memory, perhaps because you need to make changes in the program, or perhaps because you simply need a copy of the program. In any case, you can use the **LIST** command to tell the computer to display the program.

You can list any part of a program by putting a **line-number range** after **LIST**. The computer will print all statements that have line numbers in that range.

For example, **LIST 200-350** tells the computer to list all the lines from **200** through **350**.

The **LIST** command is very useful. If a program doesn't work as it should, your first step should be to display the program in memory. Have the computer furnish a copy of the program on the screen and use this copy to troubleshoot the program.

**NEW:** When you turn off the computer, the contents of memory are cleared out automatically. While working on the computer, however, you may unknowingly mix programs together. Suppose you are working with one program and decide to go on to another. If you don't clear the first program out of memory, the second program will go into memory right over the first. As a result, parts of both programs may be in memory. The way to avoid jumbling programs is to clear (or erase) a program with the **NEW** command when you are through with it.

**RUN:** A BASIC program is simply a set of instructions on which the computer will act. However, the computer needs to be told to start this process. When the computer receives the **RUN** command, it goes to the statement with the lowest line number, carries out its instructions, then goes to the statement next in numeric order. It keeps on performing instructions in numeric order, unless the program directs that a statement be done out of order.

- Use **Ctrl|Home** to clear the screen. Then enter **LIST** to display the program in memory.
- Enter **NEW** to clear the program in memory.
- Enter **RUN** to tell the computer to execute a program.

**RENUM:** Quite often programmers want to renumber the lines in a program after changes have been made. Renumbering does not affect the execution of the program, but merely makes the program look nicer or allows more line insertions, if needed. The **RENUM** command renumbers the lines in a program. For example,

```
RENUM 1000, 540, 100
```

leaves all line numbers up to **540** as they were and renumbers line **540** as **1000**. Each line after the new line **1000** is numbered in increments of **100** — **1100**, **1200**, etc. In other words, the three whole numbers after **RENUM** tell the computer, respectively, what number to use for the first line to be renumbered, where to begin the renumbering, and what interval to use between subsequent renumbered lines. Thus, if **I**, **J**, and **K** stand for whole numbers, the command **RENUM I, J, K** tells the computer to use **I** for the first number to be changed, to start renumbering at line **J** of the existing program, and to space subsequent renumbered lines by **K**.

As you saw in the Discovery Exercises, you can renumber all of the lines of a program by omitting the value of **J**. Thus the command

```
RENUM 500, , 20
```



instructs the computer to number the first line in the program **500**, the second line **520**, the third **540**, and so on.



If you enter the command



**RENUM 100**

all the lines in the program will be renumbered starting at **100**, spacing subsequent lines by 10. **RENUM** with no numbers following tells the computer to renumber beginning at **10** with an increment of **10**.






Later on you will see that BASIC programs can branch to other line numbers in the program. The **RENUM** command renumbers these branch line numbers as well as the line numbers themselves.


You have seen several shortcuts for entering commands into the computer. The table below summarizes the shortcuts you have used so far.

AltA	<b>AUTO</b>
AltP	<b>PRINT</b>
CtrlHome	<b>CLS</b> and 
F1	<b>LIST</b>
F2	<b>RUN</b> and 
F3	<b>LOAD "</b>
F4	<b>SAVE "</b>

Often, in the instructions for typing programs or commands, you were reminded to press the  key. This habit should be well developed now, so you will not be reminded further. From now on, when you are through typing a statement or command, press  to let the computer know you are finished.

### Correcting Mistakes

Since most of us make mistakes while typing, we need to be able to correct errors. Suppose you make a mistake while you are typing a line. How you correct it depends on whether you have pressed  yet. If you are on the line containing the error, you can move the cursor left using the , right using the , and make corrections using the techniques you learned in Chapter 1. Also, if necessary, you can insert or delete characters using the **Ins** and **Del** keys. When all the corrections are made in a line, press . Note that the cursor does not have to be at the right end of the line when you press .

If you wish to change a line after pressing , you can use the **EDIT** command. For example, if you wish to change line 110 in a program, you should enter

**EDIT 110**

The computer will display line 110 with the cursor on that line. You may now correct the line as discussed above.

A **syntax error** occurs when the computer has trouble interpreting a BASIC statement. If the computer detects a syntax error while the program is being run, it will type out an error message and the number of the line in which the error occurs. In addition, the line will be displayed on the screen with the cursor on that line. The line may now be corrected as discussed above.



## Learning to Store and Retrieve Programs

When you turn off the computer, you lose the program in memory. If you had to type in programs from scratch every time you turned on the computer, very little work would get done. Fortunately, you can type in long or complicated programs, troubleshoot them, and then store the programs on a formatted diskette for future use. You need only turn on the computer, bring up `BASICA`, and then retrieve any of the programs you stored previously.

You might not want to keep a specific program on a diskette forever. A command is provided to clear programs from diskette storage. If you move a copy of a program from a diskette into memory and then clear the copy on the diskette, the copy in memory is not touched. Likewise, you can have one program in memory and clear out another program in diskette storage without changing the first.

The commands for performing these program-management tasks are given below:

- To move a program from memory to diskette storage enter  
`SAVE "<name of program>"`
  
- To move a program from disk storage to memory enter  
`LOAD "<name of program>"`
  
- To clear out a program on diskette storage enter  
`KILL "<name of program>.BAS "`
  
- To display a directory of programs in diskette storage enter `FILES`

You must be very careful while using the `SAVE` command. For instance, suppose you have just brought up `BASICA` and wish to move a program to memory. If you inadvertently type `SAVE` instead of `LOAD`, then the program on the diskette will be replaced by the "program" in memory, which has no lines. Thus, you would have destroyed the program you wanted to use. If this should happen to you, you can recover the lost program if you keep back-up or spare copies of programs on a separate diskette.

## Recognizing the Importance of Organization and Style

Good organization and style are important considerations when writing *any* program, no matter how large or small. Too often, these concepts are ignored and programs are written in a jumbled style that is hard to read, hard to understand, and hard to change. If, from the very beginning, you give attention to questions of organization and style, you will learn good programming habits which become invaluable later on. Therefore, all the programs you will see throughout the balance of this book will demonstrate a consistent approach to organization and style.

The best way to begin writing a program to solve any problem is with paper and pencil, away from the computer. The first step is to write an outline, partly in `BASIC` and partly in English, that defines the major tasks to be accomplished. Then

the English phrases are replaced with the necessary BASIC statements. Portions of the program are indented to associate the statements with the task they perform. Let's use the final version of program **POEM** (in step 28 of the Discovery Exercises) as an example of how to develop a solution to a problem.

The problem is to define the lines of a poem and then display them on the screen. Here is how an outline might look:

```
PROGRAM POEM
  Create poem lines
  Display poem
END
```

The first and last lines of the outline will be converted into BASIC statements as is. The two English phrases between define the major tasks to be performed. Now you can focus your attention on these separate tasks and proceed more easily to program the solution. The indentation clearly sets off the tasks from the name of the program and the **END** statement.

Next you can add line numbers, convert the phrases to remark statements and enter the program. Here is how the program looks at this point:

```
100 'PROGRAM POEM
110 'CREATE POEM LINES
120 'DISPLAY POEM
130 END
```

This **program skeleton** is the framework for completing the solution. There are either two or four spaces between each line number and the text that follows. The apostrophe ('), used to create remark statements, is not counted in the spacing used in this indentation style. Everything is in capital letters since the Caps Lock key has been pressed.

- Use two spaces for each level of indentation.
- Use ' (the apostrophe) to set off remark statements.

You can now enter the program lines that perform the first task. The partially complete program now looks like this:

```
100 'PROGRAM POEM
110 'CREATE POEM LINES
112   LET A$ = "A COMPUTER WILL DO AS YOU SAY"
114   LET B$ = "IF YOU LEARN THE CORRECT WAY"
116   LET C$ = "IT DOES EXACTLY AS IT'S TOLD"
118   LET D$ = "NO MATTER HOW FUNNY OR BOLD"
120 'DISPLAY POEM
130 END
```

Each of the statements used to create the poem lines is indented two spaces further to the right. This emphasizes that the statements carry out the task described by the remark statement above them.

Next you focus on the task of displaying the poem. After the necessary statements are entered, the program now looks like this:

```

100 'PROGRAM POEM
110   'CREATE POEM LINES
112     LET A$ = "A COMPUTER WILL DO AS YOU SAY"
114     LET B$ = "IF YOU LEARN THE CORRECT WAY"
116     LET C$ = "IT DOES EXACTLY AS IT'S TOLD"
118     LET D$ = "NO MATTER HOW FUNNY OR BOLD"
120   'DISPLAY POEM
121     PRINT A$
122     PRINT B$
123     PRINT C$
124     PRINT D$
130   END

```

If you wished to refine the program to display the author's name, you might add the following lines:

```

105     INPUT "AUTHOR "; AUTHOR$
128     PRINT
129     PRINT "BY " + AUTHOR$

```

After renumbering with **RENUM 100**, the program will look exactly like this:

```

100 'PROGRAM POEM
110   INPUT "AUTHOR "; AUTHOR$
120   'CREATE POEM LINES
130     LET A$ = "A COMPUTER WILL DO AS YOU SAY"
140     LET B$ = "IF YOU LEARN THE CORRECT WAY"
150     LET C$ = "IT DOES EXACTLY AS IT'S TOLD"
160     LET D$ = "NO MATTER HOW FUNNY OR BOLD"
170   'DISPLAY POEM
180     PRINT A$
190     PRINT B$
200     PRINT C$
210     PRINT D$
220     PRINT
230     PRINT "BY " + AUTHOR$
234   END

```

You can glance at this program and see its name, and where it begins and ends. Each of the tasks to be performed is defined by remark statements. The statements to carry out each task are indented to the right for emphasis.

Paying this much attention to organization and indentation for a simple program like this may seem like overkill to you now. However, these techniques are lifesavers later on. Make it a point to write your own programs in this style. As your programs get longer and more complicated, you will see why it is important to think about the issues of organization and style.

## 2—4 PRACTICE TEST

Take this practice test to see how well you understand the goals of this chapter. The answers are in Appendix B at the end of the book.

1. How do you signal the computer that you are through typing a line or instruction?

---

2. What is wrong with the following program?

```
100 LET A = 1
110 LET B = 3
120 LET C = B - A
PRINT C
130 END
```

---

3. What would happen if the program in question 2 were corrected and executed?

---

4. How do you insert a line in a BASIC program?

---

5. How do you replace a line in a BASIC program?

---

6. How do you remove a line from a BASIC program?

---

7. How do you erase the screen?

---

8. What commands are needed to carry out the following operations?

a. Move a program from diskette storage to memory.

---

b. Move a program from memory to diskette storage.

---

c. Clear a program from diskette storage.

---

d. Clear a program from memory.

---

e. Display the program in memory.

---

f. Execute the program in memory.

---

g. Display the names of all the files saved on diskette.

---

9. What commands are needed to carry out the following operations?

a. Edit line 120 in the program in memory?

---

b. Automatically obtain line numbers starting at line 100?

---

c. Renumber the lines of a program in memory starting at line number 200 with an increment of 30?

---

10. How many spaces are required for each new level of indentation?

---

11. What is wrong with the indentation in the following program?

```
100 'PROGRAM HELLO
110  PRINT "HELLO"
120  END
```

---



12. What is wrong with the following program?

```
100 PROGRAM GOODBYE
110 PRINT "GOODBYE"
120 END
```

---



# GRAPHICS AND EDITING

## 3—1 OBJECTIVES

### **Defining Function Keys**

You will learn how to redefine the function keys to assist in typing commonly used phrases or symbols.

### **Accessing Graphics Mode**

You will learn how to access the medium-resolution graphics screen on your computer. You will learn a coordinate system with which you can locate any point on that screen.

### **Learning to Draw and Name Shapes**

You will use horizontal and vertical lines to draw various shapes. You will learn how to name such shapes for later reference.

### **Learning to Draw Circles and Rectangles**

You will learn to use the **CIRCLE** and **LINE** statements to draw circles and rectangles of various sizes easily .

### **Using Proper Organization and Style**

You will use the organizational methods discussed in Chapter 2 to develop programs with top-down structure and correct indentation.

### **Using Editing Features**

The editing features of your computer in BASICA are very powerful. You will learn to use advanced editing features to assist you while writing and modifying programs.

## 3—2 DISCOVERY EXERCISES

### The DRAW Command

1. Turn on the computer and bring up BASICA. Enter the following direct-mode statement:

SCREEN 1


This statement places the computer in medium-resolution graphics mode.

2. Enter

KEY LIST



Write down what is next to F5 on the screen. Does it match 5 at the bottom of the screen?

---


The <- indicates a .

3. To change F5 to DRAW ", enter the following statement:


KEY 5, "DRAW " + CHR\$(34)



At the bottom of the screen observe that DRAW " now appears. CHR\$(34) puts the quotation mark after DRAW. DRAW " is now stored in the key marked . Press the  key at the far left of the keyboard. What happened?

---

Press . What happened?

---

As presently defined, DRAW " will be displayed on the screen when you press .

4. Clear the screen with  and then watch as you enter the following direct-mode statement. (You can obtain DRAW " by pressing  if you defined it as indicated in step 3.)

DRAW "L60"

Was a vertical or horizontal line drawn?

---

5. Now enter the following statements:

**DRAW "U60"**

**DRAW "R60"**

**DRAW "D60"**

What figure appears on the screen?

---

What words do **L**, **R**, **U**, and **D** represent?

---

6. Clear the screen with Ctrl|Home and enter the following statement:

**DRAW "E60"**

Was a horizontal line drawn?

---

Enter

**DRAW "F60"**

**DRAW "G60"**

What can you say about the lines caused by **G60** and **E60**?

---

Enter

**DRAW "H60"**

What new figure has been drawn on the screen?

---

7. Enter the following statements (watch the screen as you enter each line):

**CLS**

**DRAW "R60 U60 L60 D60"**

**DRAW "E60 F60 G60 H60"**

In the space below, sketch the two figures that these statements produced.

---

Are the squares the same size?

---




8. Clear the screen and enter the following direct-mode statements:

```
DRAW "NE60"
DRAW "NF60"
DRAW "NG60"
DRAW "NH60"
```

Each of the instructions in the **DRAW** statement is preceded by an **N**. What change does this make? (Compare with steps 4 and 5).


---

### Top-Down Organization of a Program

9. Clear the memory with the **NEW** command. Clear the screen. The following program lines describe the structure of a program using remark statements. (Remember that if the first nonspace character after the line number is an apostrophe ('), the computer treats the line as a remark statement.) Recall that this kind of outline is known as a skeleton program. It is usually the best way to begin writing a program. Enter the lines below. (You will need to release the **Caps Lock** key to obtain lowercase. You must now use  to obtain capital letters.)

```
100 'PROGRAM BOX
110 'Clear screen
120 'Set screen
130 'Make box
140 'Draw box
150 END
```



This program describes the steps in drawing a box. The program presently consists of remark statements except for the **END** statement. In Chapter 2, you studied how organization and style can make a program easier to read. This program uses these ideas. As it stands, the program does nothing. You must insert BASIC statements to do the tasks described by the remark statements.

10. Display the program. Use the arrow keys to move the cursor to the apostrophe in line 110. Press the  key. Press the spacebar once to obtain the correct indentation. Now type


```
CLS
```

How many spaces are there between the line number and **CLS**?

---

Press the spacebar once again to separate the remark (starting with the apostrophe) from the earlier characters in the line. A remark that follows a statement is called a **trailing remark**. Press  to enter the line. Move the cursor to the apostrophe in line 120 using . Insert the following statement:

## SCREEN 1

Remember to put spaces at the beginning and end of the insertion to achieve correct indentation and to separate the trailing remark which follows the apostrophe. Again, press  to enter the line. Modify lines 130 and 140 as follows, preserving the indentation:


```
130    LET BOX$ = "R40 U20 L40 D20" 'Make box
140    DRAW BOX$ 'Draw box
```

11. Display the program. Your program should look like this:



```
100 'PROGRAM BOX
110   CLS 'Clear screen
120   SCREEN 1 'Set screen
130   LET BOX$ = "R40 U20 L40 D20" 'Make box
140   DRAW BOX$ 'Draw box
150   END
```

If your program does not look like this, make the necessary corrections. Run the program. Describe the box.

---

12. The medium-resolution screen obtained with **SCREEN 1** causes program lines with more than 40 characters to run over ("wrap") to the next line. Wrapping prevents you from seeing the organization of the program. To prevent wrapping, you will use **SCREEN 2**, the high-resolution graphics mode, to display programs. If your monitor has a 40/80 switch, it should be set to 80. Unfortunately, if you are using a TV set, it is difficult to read the screen in **SCREEN 2** graphics mode. In this case, you should use only the **SCREEN 1** graphics mode. To redefine  to make it easy to obtain **SCREEN 2**, enter the following:

```
KEY 10, "SCREEN 2" + CHR$(13)
```

The **CHR\$(13)** instructs the computer to do a carriage return () after displaying **SCREEN 2**. Press . Can you see the complete message **SCREEN 2** at the bottom of the screen?

---

If the cursor is off the screen, check to make sure that the 40/80 switch is set to 80. If your monitor does not have this switch, you will need to adjust your monitor. To see that the change has been made, enter

## KEY LIST

Observe that all of **SCREEN 2**<- appears next to F10.

13. Press **F10** and enter the command

**EDIT 130**

This displays line 130 for editing. Insert a **B** before **U20** as follows:

```
130      LET BOX$ = "R40 BU20 L40 D20" 'Make box
```

Run the program. How is the new drawing different?

---

What does the **B** before **U20** tell the computer to do?

---

14. Press **F10** and edit line 130 by inserting another **B** before **L40**.

```
130      BOX$ = "R40 BU20 BL40 D20" 'Make box
```

What figure do you think this program tells the computer to draw?

---

Run the program to see if you were correct.

### Top-Down Stick Figure

15. Clear the memory and the screen. Enter the following skeleton version of a program, whose intent is to draw a stick figure:

```
100 'PROGRAM STICK FIGURE
110 'Clear screen
120 'Set screen
130 'Draw head
140 'Draw neck and arms
150 'Draw body
160 'Draw legs
170 END
```

Enter **SAVE "STIKFIG"** to save this program skeleton on diskette for later use.

16. Now that the structure of the program is complete, you can fill in the statements that will accomplish each of the indicated tasks. Edit line 110 to look like this:

```
110      CLS 'Clear screen
```

Edit line 120 to set the screen to medium-resolution graphics mode. Here is how it should look:

```
120      SCREEN 1 'Set screen
```

17. Use **[Ins]** and **[F5]** to edit lines 130 through 160 so the computer will finish drawing the stick figure when the program is run. These lines should look like this:

```

130    DRAW "L10 U20 R20 D20 L10" 'Draw head
140    DRAW "D10 NF30 NG30" 'Draw neck and arms
150    DRAW "D30" 'Draw body
160    DRAW "NF40 NG40" 'Draw legs

```

Display the program. It should look like this:

```

100 'PROGRAM STICK FIGURE
110   CLS 'Clear screen
120   SCREEN 1 'Set screen
130   DRAW "L10 U20 R20 D20 L10" 'Draw head
140   DRAW "D10 NF30 NG30" 'Draw neck and arms
150   DRAW "D30" 'Draw body
160   DRAW "NF40 NG40" 'Draw legs
170   END

```

Run the program. Describe the figure displayed.

---

Compare the length of the body to that of the neck.

---

Compare the length of the legs to that of the arms.

---

### Scaling Figure Sizes

18. Now, let's scale the size of the figure. Press **[F10]** and enter the following remark statement:

```

122   'Scale figure

```

Now enter the lines to accomplish this task.

```

124   PRINT "ENTER SCALE NUMBER"
126   INPUT SCALE
128   DRAW "S = SCALE;"

```

The **S = SCALE;** part of the **DRAW** statement in line 128 determines the length of each line drawn. Display the program and observe its structure. Notice how the indentation helps you see this structure. Run the program. At the input prompt, enter 2. What happened to the size of the picture?

---

Run the program again and enter 8 at the input prompt. What happened to the size this time?

---

What do you think will happen if you enter 4 for the scale factor?

---

Try it and see if you were correct.

### Plotting Points

19. You should still be in **SCREEN 1**. Enter **SCREEN 1** if you are not sure. You can draw points and lines on the screen with other statements. Clear the memory and the screen. Enter the following direct-mode statement:

```
PSET (160, 100)
```

What happened?

---

You should see a point in the center of the screen. Now enter

```
PSET (319, 0)  
PSET (0, 199)
```

Are the points in the corners of your screen?

---

If your display is a TV set (rather than a computer monitor), you may not be able to see both of these corner points. What should you enter to place a point in the lower right corner of the screen?

---

Try it and see if you were correct. The upper left corner is (0,0). The medium-resolution graphics screen is 320 by 200 points. The points are indicated with two numbers. The first number is between 0 and 319 and is measured from the left edge of the screen. The second number is between 0 and 199 and is measured from the top of the screen. These two numbers are called the coordinates of the point.



20. A statement similar to **PSET** (point set) can be used to erase points. The **PRESET** (point reset) statement draws a point in the background color. Enter

**PRESET (160, 100)**

What happened to the point in the center of the screen?

---

How would you make the center point reappear?

---

Try it and see if you were right.

### Plotting Circles

21. Clear the screen and enter

**CIRCLE(160, 100), 30**

What is the center of this circle?

---

What do you think the radius of the circle is?

---

To check this, enter

**DRAW "R30"**

Where was the line drawn?

---

22. Clear the screen and enter

**CIRCLE (240, 120), 30**

Where was the circle drawn this time?

---

Do not clear the screen before you enter

```
CIRCLE (240, 120), 30, 0
```

What happened?

---

The last 0 in this line causes the circle to be drawn in the background color. This has the effect of erasing the circle. What statement would you use to draw a circle with center at (240, 120) and radius 45?

---

### Plotting Lines and Rectangles

23. Clear the memory and screen. Enter the following program:

```
100 'PROGRAM DRAW LINE
110 LINE (25, 75) - (75, 125)
120 END
```

Run the program. What happened?

---

Enter **EDIT 110** to edit line 110. Add a 0 at the end as follows:

```
110 LINE (25, 75) - (75, 125), 0
```

Do not clear the screen. Run the program. What happened?

---

The last 0 in the line sets the color to the background color. Again, use the **EDIT** command to replace the 0 with a 1 and add a B as follows:

```
110 LINE (25, 75) - (75, 125), 1, B
```

Clear the screen and run the program. What happened?

---

You should see a box.

24. Again, change this line by adding an F to the B at the end of the line, as follows:

```
110 LINE (25,25), 1, BF
```

Clear the screen and run the program. What happened?

---

Finally, do not clear the screen but edit line 110 as follows (be sure to change the last 1 to 0):

```
110    LINE (40, 90) - (60, 110), 0, BF
```

Run the program. What happened?

---

25. Clear the memory and the screen. Load the skeleton program **STIKFIG** from your diskette by typing

```
LOAD "STIKFIG"
```

Press **[F10]** and display the program. Let's recreate this program using the **LINE** statement. Insert **CLS** in line 110. Insert **SCREEN 2** in line 120. Add and insert **LINE** statements in lines 130 through 164 so that the program looks like this:

```
100 'PROGRAM STICK FIGURE
110   CLS 'Clear screen
120   SCREEN 2 'Set screen
130   LINE (80, 40) - (120, 60), 1, B 'Draw head
140   'Draw neck and arms
142     LINE (96, 60) - (104, 70), 1, B
144     LINE (40, 70) - (160, 74), 1, B
150   LINE (60, 74) - (140, 84), 1, B 'Draw body
160   'Draw legs
162     LINE (70, 84) - (78, 94), 1, B
164     LINE (122, 84) - (130, 94), 1, B
170   END
```


26. Run the program. How does the new stick figure compare to the former version in step 17?
- 

27. Remove your diskette and turn off the computer.

### 3—3 DISCUSSION

#### Defining Function Keys

As you saw in the Discovery Exercises, the function keys at the left of the keyboard can be used to cut down on the typing load. These keys are most useful if they cause frequently used commands to be displayed on the screen. When you turn the computer on, you see prompts across the bottom of the screen that remind you of the definition of each key. If you are in the mode where there are 40 characters across the screen, you can see the definition of the first five keys. If you are in 80 character mode, you can see the definition of all ten keys.

When you turn the computer on, there are predefined definitions in force for all ten keys. The table on the next page gives these definitions and explains what they mean. In the definitions, the left arrow stands for .

Key	Definition	Explanation
<b>F1</b>	<b>LIST</b>	Display the program in memory.
<b>F2</b>	<b>RUN &lt;--</b>	Perform the program in memory.
<b>F3</b>	<b>LOAD "</b>	Load a program from diskette. You must supply the program name and a closing quotation mark.
<b>F4</b>	<b>SAVE "</b>	Save a program from diskette. You must supply the program name and a closing quotation mark.
<b>F5</b>	<b>CONT &lt;--</b>	Resume execution of a program that has been halted.
<b>F6</b>	<b>, "LPT1: "</b>	A suffix used to send information to a line printer.
<b>F7</b>	<b>TRON &lt;--</b>	Turn tracing mode on.
<b>F8</b>	<b>TROFF &lt;--</b>	Turn tracing mode off.
<b>F9</b>	<b>KEY</b>	Word used in KEY commands.
<b>F10</b>	<b>SCREEN 0, 0, 0, &lt;--</b>	Sets a standard graphics mode.

If you do not wish to see the **KEY** prompts at the bottom of the screen, the command **KEY OFF** removes them. **KEY ON** causes the prompts to be displayed. **KEY LIST** causes a list of the current definitions of the ten function keys to be displayed on the screen.

You can redefine the function keys with the **KEY** command. For example, if you wish to define function key **F6** to display the characters **FILES** followed by a **←**, the following command does that:

```
KEY 6, "FILES" + CHR$(13)
```

The number before the comma tells the computer which function key is to be redefined. The string information after the comma contains the definition. In this case, **CHR\$(13)** stands for a carriage return (**←**) and is joined to the string inside the quotes. If you wish a quotation mark to be joined, substitute **CHR\$(34)**. (The **CHR\$** function will be discussed further later in the book.)

The definitions (or redefinitions) of the function keys remain in effect as long as you remain in **BASICA**. When you return to the system level (**A>** prompt), the definitions revert to the standard ones in the table above.

### Accessing Graphics Mode

There are three screen modes available on your computer. **BASICA** begins in screen mode 0, or text mode, the mode you used in previous chapters. **SCREEN 1** places the computer in medium-resolution graphics mode which has 320 points across the screen and 200 points down. **SCREEN 2** places the computer in high-resolution graphics mode that has 640 points across the screen and 200 down.

The **SCREEN 2** mode is useful for displaying programs that have lines with more than 40 characters in them. In **SCREEN 1** mode, lines with more than 40 characters are wrapped around onto the next line. This destroys the style and organization that is important to good programming. This can be avoided in most cases by displaying programs using **SCREEN 2**.



- Use **SCREEN 1** for medium-resolution graphics mode.
- Use **SCREEN 2** for high-resolution graphics mode.
- Use **SCREEN 2** or **SCREEN 0** to display programs with long lines.

Points on the screen are located with two numbers, called coordinates. The first number measures the horizontal position of the point, measured from the left edge of the screen. The second number measures the vertical position of the point, measured from the top of the screen.

On the medium-resolution graphics screen (**SCREEN 1**), the point in the upper left corner of the screen has coordinates (0, 0). The point in the center of the screen has coordinates (160, 100), and so on.

### Learning to Draw and Name Shapes

BASICA has a built-in graphics sublanguage that is accessed through the **DRAW** statement. The **DRAW** statement can be used in either the direct or program mode, as you saw in the Discovery Exercises. The graphics sublanguage includes the following statements:

	Statement	Meaning
<b>Horizontal and Vertical Statements</b>	L	Left
	R	Right
	U	Up
	D	Down
<b>Diagonal Statements</b>	E	Up and to the right
	F	Down and to the right
	G	Down and to the left
	H	Up and to the left

You can modify **DRAW** sublanguage graphics statements by adding letters before them and numbers after them. The letter **B** before a graphics substatement tells the computer to move the screen pen (or cursor) without drawing a line. Actually, it draws an "invisible" line in the screen background color. The letter **N** before a statement causes the statement to be performed and then returns the cursor to its original position.

Numbers following the statement indicate how far the graphics cursor is to move horizontally, vertically, or diagonally. The diagonal motion in the **E**, **F**, **G**, and **H** statements is measured with respect to the horizontal direction. The length of the line drawn with diagonal sublanguage statements is always longer than the number of units used (about 1.4 times longer). For example, the graphics program segment

```
120 SCREEN 1 'Enter graphics mode
130 DRAW "R20 U20 G20" 'Draw triangle
```

will draw a triangle.

The **DRAW** statement must be followed by a string of characters and/or numbers. These characters or numbers can be stored in a string variable such as **BOX\$**.



If a string of substatements is stored that way, the statement

```
130    DRAW BOX$
```

will cause the stored figure to be drawn. Quotation marks are not necessary around **BOX\$**. If the substatements are given directly in the **DRAW** statement, however, then quotation marks are required around the string of substatements.

Both these rules are illustrated in the following program segment which contains the instructions to draw a pole with a cap on it.

```
120    'Make cap
130    LET CAP$ = "R20 U20 L40 D20 NR20 L20"
140    DRAW "ND90" + CAP$ 'Draw pole and cap
```

The scale (S) substatement demonstrates the use of variables with sublanguage substatements. The scale substatement determines the size of the object. If the scale is 4, there will be no change in the size of the figure. If the scale is 8, the figure will be drawn twice as large; if the scale is 2, the figure will be half as large.

In a graphics program, the segment

```
110    LET A = 8 'Set size
120    DRAW "S = A;" 'Set scale
130    DRAW "L10 D10 R10 U10" 'Draw box
```

will draw a box 20 units long on each side. Other allowable values of the variable **A** will scale the box accordingly. The semicolon is required after each substatement that uses a variable.

More information about these draw substatements can be found in the BASIC reference manual. Also available, are the angle (A), move (M) and execute (X) sublanguage statements.

### Learning to Draw Circles and Rectangles

In addition to the graphics sublanguage accessed by the graphics statement **DRAW**, you used the graphics statements **PSET**, **PRESET**, **CIRCLE**, and **LINE**. These statements must be followed by numbers in parentheses indicating locations. For example, the statement **PSET (200, 150)** places a white point at the coordinates (200, 150). **PRESET (200, 150)** draws a point at the coordinates (200, 150) in the background color. This has the effect of erasing the point if one is already there. **CIRCLE (200, 150), 30** draws a circle of radius 30 in the lower-right corner of the screen (center at coordinates (200, 150)).

The **DRAW** statement allows you to draw both horizontal and vertical lines as well as diagonal lines. However, you can also use the **LINE** statement to draw a line between any two points on the screen. For example,

```
LINE (10, 10) - (40, 70)
```

tells the computer to draw a line from the point (10, 10) to the point (40, 70). With

a simple change, you can use the **LINE** statement to draw boxes. For example, in **SCREEN 2** mode,

```
LINE (40, 40) - (70, 100), 1, B
```

tells the computer to draw a rectangle whose diagonal is the line joining the points (40, 40) and (70, 100). The diagonal is not drawn. If you are using **SCREEN 1** with a black-and-white monitor, you would not see the vertical sides of the rectangle since only every other vertical line can be seen in medium resolution graphics mode. The **1** in the statement specifies that the box will be drawn in white. If you use **BF** instead of **B**, then the box will be filled with white. For information about other graphics statements, see your BASIC reference manual.

### Using Proper Organization and Style

You used top-down structure in the programs you wrote in the Discovery Exercises. This method allows you to break the problem into manageable segments. You can then focus all your attention on accomplishing these smaller tasks in one or two program lines. The indentation style used in both the skeleton and final versions of the program makes the program easy to write, easy to read, and easy to change. The notion of top-down organization will be amplified in the next chapter.

In program **STIKFIG**, you broke down the task of drawing the figure into several small, easy-to-write segments. You saw that it was possible to modify the program easily. Programs written this way are much more likely to run successfully the first time.

### Using Editing Features

The powerful editing features of BASICA make it easy to write in the top-down, indented style. You first create an outline of the solution of the problem on paper. Next, you write the program skeleton using remark (') statements to describe the tasks to be done. You then insert the necessary BASIC statements to accomplish each task.

The easiest way to modify the program is to first display the program skeleton on the screen. Then use the arrow keys to move the cursor to the apostrophe (') where the first BASIC statement is to be inserted. Go to the insert mode by pressing **Ins**, and then type the appropriate statement. Pressing **↵** enters the modified line, gets you out of the insert mode, and places the cursor on the first character of the next line on the screen. Repeat this process for each task in the program skeleton. If additional lines are required to accomplish any of the tasks, use extra line numbers. These lines should be appropriately indented. Later on, the lines can be renumbered for easier reading.

If you make errors while entering a program, use the arrow keys, **←**, **Ins**, and **Del** keys to correct them. **←** erases characters to the left of the cursor. **Del** erases the character under the cursor. **Ins** allows characters to be inserted into a line. Pressing **↵**, or either the left- or right-arrow keys ends the insert mode. The shape of the cursor tells you what mode the computer is in. In insert mode, the cursor is shaped like a box.

**3—4 PRACTICE TEST**

1. If you are using **SCREEN 1**, what figure does the following direct-mode statement tell the computer to draw?

**DRAW "BU20 L20 D40 R40 U40 L20"**

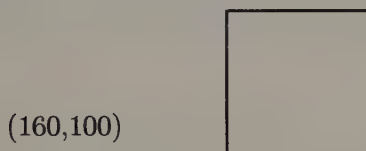
---

2. If you are using **SCREEN 1**, What figure does the following direct-mode statement tell the computer to draw?

**DRAW "E20 F20 L40 D40 R40 U40"**

---

3. Write a direct-mode statement that will draw the following diagram with the point (160, 100) as indicated.




---

4. Assume you are using **SCREEN 1**. Write statements that will set points at:  
The center of the screen.

---

The upper left corner of the screen.

---

The top center of the screen.

---

The point (240, 150)

---

5. What are the radius and center of the circle that will be drawn if the computer performs the following direct-mode statement?

**CIRCLE (140, 90), 30**

---

6. Write a program line that will draw a circle of radius 40 with center at (80, 50). Number the line 200.
- 

7. How do you set the medium-resolution graphics screen mode?
- 

8. What does the following program tell the computer to draw?

```
100 'PROGRAM WHAT FIGURE
110   CLS 'Clear screen
120   SCREEN 1 'Set screen mode
130   DRAW "NR20" 'Top
140   DRAW "D20 NR20 D20" 'Middle
150   DRAW "R20" 'Bottom
160   END
```

---





# SUBROUTINES AND TOP-DOWN ORGANIZATION

## 4—1 OBJECTIVES

### Packaging Statements into Subroutines

BASIC provides a way to collect a block of statements into a subroutine. You will explore how **GOSUB** and **RETURN** statements are used to tell the computer to perform the statements in a subroutine.

### Using Top-Down Organization

Programs written using top-down organization are easier to write and understand. They consist of a main routine followed by one or more subroutines. You will learn this organization by building on what you learned about organization and style in Chapter 2.

### Seeing Subroutine Bugs

You will be shown what happens when **GOSUB** or **RETURN** statements are missing from subroutines.

### Using the Top-Down Strategy

Programmers using the top-down strategy begin by writing an English-language description of the tasks. Then the program is developed by focusing on each individual task and writing statements that accomplish the tasks. You will learn to use this strategy.

### Working with Program Examples

You will continue to learn about programming by working with examples. The examples in this chapter, and the rest of the book, will use the top-down strategy.

## 4—2 DISCOVERY EXERCISES

### Introduction to Subroutines

1. Turn on your computer. Bring up BASICA. The first few steps will be a review of the work you did in Chapter 3. Enter the following statement:

```
SCREEN 1
```

Recall from Chapter 3 that this places the computer in medium-resolution graphics.

2. Remember that you can define **F5** to be **DRAW "** (see step 3 of Chapter 3) as a quick way to enter **DRAW "**. Enter the following statement:

```
DRAW "L60"
```

What happened?

---

3. The fact that the **DRAW** command worked means that one of the graphics screens is active. There should be a horizontal line drawn on the screen. Enter the following lines:

```
DRAW "U60"
DRAW "R60"
DRAW "D60"
```

Describe what these commands told the computer to do.

---

4. Now, give the same instructions to the computer, but this time in the program mode. Clear the screen with **Ctrl|Home**. Then enter the following statements:

```
210 DRAW "L60"
220 DRAW "U60"
230 DRAW "R60"
240 DRAW "D60"
```

This time nothing happened since there are line numbers in each statement. What command is needed to tell the computer to start carrying out the instructions in the program?

---

5. Clear the screen (see step 4). Then enter **RUN**. What happened?
- 

6. So far, so good. Everything you have seen so far in this session you have done before. Now for something new. Clear the screen and then enter the statement

**250 RETURN**

Enter the **RUN** command. What happened?

---

7. The computer drew the square, but complained about a missing **GOSUB** statement. Apparently, if the **RETURN** statement follows a group of statements you can't use the **RUN** command as you did in step 5. Clear the screen. Enter the statement

**GOSUB 210**

What happened?

---

8. This time the computer didn't complain and drew the square on the screen. Clear the screen. Use the **LIST** command to display the program in memory. It should look like this:

```
210 DRAW "L60"
220 DRAW "U60"
230 DRAW "R60"
240 DRAW "D60"
250 RETURN
```

As you have seen, if there is a **RETURN** statement at the end of a block of statements, the computer will complain if you try to use the **RUN** command. However, **GOSUB 210** seemed to work fine.

9. Add the following statements to your program:

```
310 DRAW "NL60"
320 DRAW "NU60"
330 DRAW "NR60"
340 DRAW "ND60"
350 RETURN
```

Now there are five additional statements in the program. Before you added these statements, **GOSUB 210** caused the computer to draw a square using the instructions in lines 210 through 240. What do you think **GOSUB 210** will do now?

---

Clear the screen. Enter the command **GOSUB 210**. Were you right?

10. What do you think will happen if you enter the command **GOSUB 310**?
-

Clear the screen. Then enter **GOSUB 310**. What happened?

---

### Main Routine

11. So far, you have used the **GOSUB** command in the immediate mode to tell the computer to perform a package of instructions. You can also use **GOSUB** in a program statement. Indeed, this is the usual case. If you were going to draw a house, an outline of the program might be

```
PROGRAM FRAME HOUSE
Draw house
Draw roof
END
```

You would then enter the following program skeleton:

```
100 'PROGRAM FRAME HOUSE
110     GOSUB 200 'Draw house
120     GOSUB 300 'Draw roof
130  END
```

This block (lines 100 through 130) forms the **main routine** of the program. It looks like the programs you have written previously. The main routine begins with a remark statement that names the program and ends with an **END** statement. The main difference is that the statements in the main routine are **GOSUB** statements. The **GOSUB** statements are indented to show that they are part of the main routine.

12. The program won't draw a house yet since the subroutines have not been written. Let's first work on subroutine **HOUSE**. A very simple drawing of a house is just a square. You can do this with the **DRAW** statements. Clear the memory and the screen. Enter the lines below and adjust the indentation until the program looks like this:

```
100 'PROGRAM FRAME HOUSE
110     GOSUB 200 'Draw house
120     GOSUB 300 'Draw roof
130  END
190 '
200 'SUB DRAW HOUSE
210     DRAW "R60"
220     DRAW "D60"
230     DRAW "L60"
240     DRAW "U60"
250  RETURN
290 '
300 'SUB DRAW ROOF
```

Statements 200 through 250 form a **subroutine**. The subroutine begins with a remark statement that names the subroutine and ends with a **RETURN** statement.

The four **DRAW** statements are indented to show they are the body of the subroutine. The empty remark statements in lines 190 and 290 are used to separate the subroutines in the program.

13. Even though subroutine **DRAW HOUSE** is part of a program, you can still tell the computer to perform it in the immediate mode. Clear the screen and enter

```
GOSUB 200
```

What happened?

- 
14. If everything went as planned, you should have seen a box representing the house. Now let's work on the roof subroutine. Clear the screen and list the program. Then add lines 310-340 to complete subroutine **DRAW ROOF** so that the subroutine looks like:

```
300 'SUB DRAW ROOF
310   DRAW "E30"
320   DRAW "F30"
330   DRAW "L60"
340 RETURN
```

The second subroutine contains the instructions to draw a triangle representing the roof on the house. Clear the screen and then enter the following command:

```
GOSUB 300
```

The subroutine is working correctly if you saw a triangle drawn on the screen.

15. Clear the screen and display the program. Here is what it should look like:

```
100 'PROGRAM FRAME HOUSE
110   GOSUB 200 'Draw house
120   GOSUB 300 'Draw roof
130 END
190 '
200 'SUB DRAW HOUSE
210   DRAW "R60"
220   DRAW "D60"
230   DRAW "L60"
240   DRAW "U60"
250 RETURN
290 '
300 'SUB DRAW ROOF
310   DRAW "E30"
320   DRAW "F30"
330   DRAW "L60"
340 RETURN
```



16. The program is almost finished now. However, it is a good plan to do some house-keeping to ensure that the house will be drawn on a clear, medium-resolution graphics screen. Enter the following statements in the main routine:

```
104    SCREEN 1
106    CLS
```

What do you think will happen if you run the program?

---

Enter the RUN command and see if you were right.

17. As the program stands, the roof is positioned correctly on the house. But suppose you added a statement to move the pen after the house is drawn but before the roof is drawn. Enter the following statement:

```
115    DRAW "BR70"
```

Run the program. What change did the new statement produce?

---

18. Now enter the two statements below:

```
117    GOSUB 200 'Draw house
119    DRAW "BD90"
```

Display the program. What do you think will happen if you run the program now?

---

Run the program and see if you were right.

19. Enter two more statements.


```
122    DRAW "BL70"
124    GOSUB 300 'Draw roof
```

Clear the screen and display the program. Based on what you have already seen, what do you think this new version of the program will tell the computer to do?

---

Run the program. What did happen?

---

20. Delete lines 117, 119, 122, and 124 from the program. (Remember that you can delete a line from a program by typing its line number and then pressing .) When you are through, clear the screen and display the program. It should look like this:

```

100 'PROGRAM FRAME HOUSE
104   SCREEN 1
106   CLS
110   GOSUB 200 'Draw house
115   DRAW "BR70"
120   GOSUB 300 'Draw roof
130   END
190 '
200 'SUB DRAW HOUSE
210   DRAW "R60"
220   DRAW "D60"
230   DRAW "L60"
240   DRAW "U60"
250   RETURN
290 '
300 'SUB DRAW ROOF
310   DRAW "E30"
320   DRAW "F30"
330   DRAW "L60"
340   RETURN

```

This program shows the top-down organization that will be used throughout the balance of the book. The main routine is followed by one or more subroutines.

### Subroutine Bugs

Run the program. Where are the house and roof drawn on the screen?

---

21. Remove the **RETURN** statement in line 250. What do you think will happen if the program is run now?

---

Run the program. Record what happened.

---

22. Put the **RETURN** statement back in line 250. Remove the **RETURN** statement in line 340. What do you think will happen now?

---

Run the program. What did happen?

---

23. Put the **RETURN** statement back in line 340. Now remove the **END** statement in line 130. What do you think will happen if you run this new version of the program?

---

Run the program. Explain, if you can, what actually happened.

---

24. Turn the computer off and go on to the discussion section.

### 4—3 DISCUSSION

#### Packaging Statements into Subroutines

All computer languages provide a way to package a block of statements together into a group. Then you can use a single command to tell the computer to perform all the statements in the group. In BASIC, the group of statements is called a **subroutine**. **GOSUB** is used to tell the computer to perform the group of statements.

There are several reasons why it is useful to group statements into subroutines. If the same block of statements is to be performed many times in a program, it saves program space in memory to put the statements into a subroutine rather than to repeat the statements each time they are needed. Another reason to use subroutines is that with them, top-down program organization is possible. Programs written in top-down fashion are easier to write, easier to read, easier to change, and are more likely to run correctly the first time.

A block of statements that ends with a **RETURN** statement is called a subroutine. You tell the computer to perform a subroutine with a **GOSUB** statement. The word **GOSUB** *must* be followed by the line number of the first line of the subroutine. It is a good practice to always make the first line of the subroutine a remark statement that names the task accomplished by the subroutine.

The statements between the beginning remark statement and the **RETURN** statement form the **body** of the subroutine. These statements give the computer the instructions it needs to carry out the purpose of the subroutine. The body statements should be indented to make it clear that they belong to the subroutine.

The **GOSUB** statement is used in a program to tell the computer to perform the task described in a subroutine. While the **GOSUB** statement is most often found as a statement in a program, it is also possible to use **GOSUB** in the immediate mode. This is particularly valuable as a trouble-shooting tool when you are writing complicated programs. You will be shown how to do this in the programming examples later on in this chapter.

#### Using Top-Down Organization

In the Discovery Exercises you wrote a program that was described as having **top-down organization**. Experience has shown that this kind of organization makes programs easier to write and understand. All top-down programs in this text begin with a main routine followed by one or more subroutines.

All main routines will have this general form:

```
'PROGRAM (name of program)
      (body of main routine)
END
```

You assign numbers to the lines in a program skeleton. The main routine should begin with a remark statement that identifies the program. (Remember that if the first nonspace character after the line number is an apostrophe, the computer treats the rest of the statement as a remark.) The last line in the main routine *must* be the **END** statement. The body of the main routine is usually made up of **GOSUB** statements that send the computer to subroutines. It is a good plan to place after each **GOSUB** statement a trailing remark naming the subroutine to which the **GOSUB** refers. Trailing remarks follow an apostrophe at the end of the statement. The body statements should be indented to emphasize that they belong in the main routine.

All subroutines will have this general form:

```
'SUB (name of subroutine)
      (body of subroutine)
RETURN
```

The subroutine should begin with a remark statement that names the subroutine. The last line in the subroutine *must* be the **RETURN** statement. The body of the subroutine usually contains BASIC statements that tell the computer how to carry out some task. Sometimes, if the task is complex, a subroutine may contain a **GOSUB** statement to another subroutine where another task is performed. If a subroutine contains a **GOSUB** to another subroutine, the subroutines are said to be **nested**. The body statements in the subroutine are indented to emphasize that they belong in the subroutine.

Empty remark statements should be used to separate the main routine from the first subroutine and to separate each of the subroutines that follow. An empty remark statement is just a line number followed by a space and an apostrophe. The computer ignores such lines. Their sole purpose is to improve readability. Just as blank lines between paragraphs make an English composition easier to read, blank lines in a program make it easy to see the parts of the program.

- **GOSUB sends the computer to a subroutine.**
- **RETURN ends a subroutine and returns the computer to the sending routine.**

In summary, it is quite simple to write a program using top-down organization. Begin by writing an outline of the main tasks to be performed. Then enter a program skeleton of the main routine, containing **GOSUBs** to these main tasks. Isolate each of these tasks in a subroutine. If a specific task in a subroutine gets too complicated, put in a **GOSUB** to another subroutine, where a subtask is performed. For now, you should concentrate simply on the form of top-down programming. As



you proceed through the book, you will see repeated examples of this programming form. Gradually, you will understand why this way of writing programs is preferred.

### Seeing Subroutine Bugs

In the Discovery Exercises, you were directed to place errors (or bugs) into a program containing subroutines. You saw the sometimes unpredictable results programs containing subroutine bugs can produce. If you always ended each subroutine with a **RETURN** statement and always placed an **END** statement at the end of the main routine, there would be no reason to worry about the types of bugs you have considered here. Sooner or later, however, you will surely make such errors. Some subroutine bugs are particularly bothersome since the program runs, the computer reports no errors, yet the output is nonsense. To be forewarned is to be prepared for such bugs.

It is easier to understand how subroutine bugs affect a program if you look at a specific program. The program below does nothing particularly useful except to serve as a vehicle for studying subroutine bugs.

```

100 'PROGRAM SUBROUTINE BUGS
110   GOSUB 200 'First letter
120   GOSUB 300 'Second letter
130   END
190 '
200 'SUB FIRST LETTER
210   PRINT "A"
220   RETURN
290 '
300 'SUB SECOND LETTER
310   PRINT "B"
320   RETURN

```

As you can see, this program follows the rules of top-down organization. If the program is run, the letters **A** and **B** will be printed on the screen vertically. This seems singularly unimpressive and is trivial if the output is all that matters. You could accomplish the same thing with two **PRINT** statements. However, the structure of the program, coupled with the simple output, provides an ideal way to study how bizarre results are sometimes generated by more complicated programs.

First, suppose you forgot to put the **END** statement at the end of the main routine. Here is how the program would look:

```

100 'PROGRAM SUBROUTINE BUGS
110   GOSUB 200 'First letter
120   GOSUB 300 'Second letter
190 '
200 'SUB FIRST LETTER
210   PRINT "A"
220   RETURN
290 '
300 'SUB SECOND LETTER
310   PRINT "B"
320   RETURN

```



Think about what would happen if the computer were to run this program. The first line in the main routine is a remark statement, which is ignored. The second line is a **GOSUB** statement to line 200. The task in that subroutine (printing the letter **A**) is performed. The computer returns to line 120, another **GOSUB** statement that sends the computer to the subroutine beginning in line 300. After that task (printing the letter **B**) is performed, the computer returns to line 190, another remark statement.

Here is where the problem arises. After the computer performs line 190, it goes right on to line 200. But this is the beginning of a subroutine and the computer was *not* sent there by a **GOSUB** statement. After printing the letter **A**, the computer reaches the **RETURN** statement, detects the fact that it was not sent to the subroutine by a **GOSUB** statement, indicates a **RETURN without GOSUB** error and stops. Here is what the program would output:

```
A
B
A
RETURN without GOSUB in 220
```

Compare this output to the intended output:

```
A
B
```

At least, this type of error (the missing **END** statement) generates an error message, and you are aware that something is wrong. This is not always the case, as you will soon see.

Suppose you forget to put a **RETURN** statement at the end of the first subroutine. With the **END** statement back in place, here is what the program would look like:

```
100 'PROGRAM SUBROUTINE BUGS
110   GOSUB 200 'First letter
120   GOSUB 300 'Second letter
130  END
190 '
200 'SUB FIRST LETTER
210   PRINT "A"
290 '
300 'SUB SECOND LETTER
310   PRINT "B"
320  RETURN
```

Again, think about what would happen if the computer were to run this program. **GOSUB 200** sends the computer to the subroutine beginning in line 200. But this time, due to the missing **RETURN** statement, the body of the subroutine includes the lines 210 through 310. Thus, the letters **A** and **B** are printed. Then the computer returns to the next line in the main routine. This is another **GOSUB** statement which sends the computer to the subroutine beginning in line 300. This time a **B** is printed. When the computer returns to the main routine, the next statement is **END**, which stops the program. Here is the output:

A  
B  
B

Again, the output was supposed to be:

A  
B

When the **RETURN** statement is missing, the output is incorrect, *and there is no error message*. In programs that have complicated output, this kind of error may be difficult to detect.

Other subroutine bugs involve the **GOSUB** statement itself. For example if there is no **GOSUB** statement sending the computer to a subroutine, that subroutine will not be performed. Or, if a **GOSUB** statement tries to send the computer to a nonexistent line number, the computer will complain and stop. Finally, it is perfectly legal to have a **GOSUB** statement send the computer to the same statement that contains the **GOSUB**. Here is an example:

```
150 GOSUB 150
```

When the computer performs this statement, it keeps cycling at the same statement for a while, reports an **Out of memory in 150** error message, and stops. The reason for this error is that the computer uses a memory stack to keep track of how to get back from subroutines. Each **GOSUB** statement places one line number on the stack. Each **RETURN** statement takes one line number from the stack. But you can see that the result of line 150 above is to keep putting numbers on the stack. None are removed. Sooner or later, the stack is full and can accept no more. At this point, the error message is generated.

It is easy to avoid problems with subroutines if you keep these rules in mind when writing programs: You must end the main routine with an **END** statement. You must end each subroutine with a **RETURN** statement. The line number after **GOSUB** must be a line in a program, but not the line containing the **GOSUB**. You should begin each subroutine with a remark statement that explains its purpose. You should indent the statements in the body of subroutines. You should use blank lines between subroutines to make them easier to read. It is always wise to read a program from top to bottom, checking that these rules are followed before trying to run it. As you have seen, a program with bugs in it may run, give incorrect results, but produce no error messages.

### Using the Top-Down Strategy

The programs with subroutines that you have seen so far have been quite simple. The bodies of both the main routines and the subroutines in these programs were brief and needed no special explanation. However, when programs get complicated, you need a strategy that will always work and always help you organize the program so that it is easy to write, read, and change.

Using the top-down programming strategy, you begin by writing on paper an English-language description of the tasks. These tasks you write on paper have no line numbers; they make up the comments of the main routine. *The importance of this step cannot be overemphasized.* The whole point of top-down organization is

breaking down a complicated problem into several simpler problems. Each of the simpler problems is described by an English phrase. Initially, the body of the main routine contains these phrases.

The next step is to produce a program skeleton and examine each of the phrases that describe the subtasks to be performed. If you can perform the task with one or two BASIC statements, then these one or two statements take the place of the English phrase. However, if it looks like the task will get complicated or will require many statements, replace the phrase with a **GOSUB** statement to a subroutine where the details will be written.

The subroutines are written the same way. The body of each subroutine is expressed initially in English phrases that describe the particulars to be done by that subroutine. Then, if you can perform one of the particular tasks with a BASIC statement or two, these statements take the place of the English phrase. However, if it appears that a lot of details are going to arise, or if complicated program logic will be needed, then replace the English phrase with a **GOSUB** statement to a new subroutine where the details will be written.

If you write your programs this way, the main routine reads like a directory to the complete program. If a subroutine contains a **GOSUB** statement to another subroutine (nesting of subroutines), you can trace the details to as fine a level as you wish. In other words, you can concentrate on specific parts of the program while writing it and, later on, while editing it. Most important, programs organized this way are easy for other people to read.

#### 4—4 WORKING WITH PROGRAM EXAMPLES

So far, you can use these BASIC tools: input with the **INPUT** statement, output with the **PRINT** statement, assignment with the **LET** statement, and the graphics statements. In addition, you have the top-down organizational tools that use **END**, **GOSUB**, and **RETURN**. With these tools alone, you can't write programs that decide between two alternate actions or programs that keep repeating an action. This severely limits the range of programs that you can write. Nevertheless, some simple programs will illustrate how to use the top-down programming method.

##### Example 1 - Plotting Big Letters

The letters and characters that you can write on the screen with the **PRINT** statement are fixed in size. You might want to use large letters in a title. Let's see how to use the graphic commands to produce oversize letters.

To keep the problem manageable, let's draw the letters in the name Herb in capital letters on the medium-resolution graphics screen. In top-down programming, the first step is to write an outline of the main routine, which contains English phrases describing the tasks to be done by the computer. Here is how such an outline might look:



```

PROGRAM BIG LETTERS
  Graphics screen
  Position and draw H
  Position and draw E
  Position and draw R
  Position and draw B
END

```

The tasks are self-explanatory. You'll use the **DRAW** command to form each of the letters. A good strategy is to begin drawing at the upper left corner of each letter, draw the letter, and then return to the upper left corner. At this point, you can move horizontally without plotting a line to the upper left corner of the next letter.

The next step in the program is to replace each of the English phrases in the outline of the main routine with BASIC statements. If, when you examine each phrase, you can accomplish the task it describes with one or two BASIC statements, write the statement(s). If the task seems more complicated, put in a **GOSUB** statement to a subroutine where you will write the details. Drawing each of the letters will be complicated, so you'll use a **GOSUB** statement. However, you can define the starting position of each letter with a single BASIC statement.

Here is the finished main routine with line numbers:

```

100 'PROGRAM BIG LETTERS
105 'Graphics screen
110 SCREEN 1
115 CLS
120 DRAW "BL110 BU50" 'Position for H
125 GOSUB 200 'Plot H
130 DRAW "BR50" 'Position for E
135 GOSUB 300 'Plot E
140 DRAW "BR50" 'Position for R
145 GOSUB 400 'Plot R
150 DRAW "BR50" 'Position for B
155 GOSUB 500 'Plot B
160 END

```

The **GOSUB** statements will send the computer to subroutines that carry out the tasks described in the trailing remarks. The first **DRAW** statement in line 120 moves (with pen up) from the center of the screen to the point where the upper left corner of the letter H will be located. The subsequent **DRAW** statements in the main routine each move the pen 50 units to the right, without drawing a line, to the starting point for the next letter.

At this point, the main routine is finished. Of course, the program is not finished. The subroutines named in the main routine must be written. A good technique is to write skeleton versions of each of the subroutines that contain an

English remark describing the purpose of each subroutine. Here are such skeleton versions:

```

190 '
200 'SUB H
205     PRINT "H"
210     'Draw H
280     RETURN
290 '
300 'SUB E
305     PRINT "E"
310     'Draw E
380     RETURN
390 '
400 'SUB R
405     PRINT "R"
410     'Draw R
480     RETURN
490 '
500 'SUB B
505     PRINT "B"
510     'Draw B
580     RETURN

```

You can use any line number falling between the number of the opening remark statement and the number of the closing **RETURN** statement when you write the BASIC statements to carry out the task described by the English phrase. Another useful technique is demonstrated here by lines 205, 305, 405, and 505. These lines let you run the program at this state to check if it is running correctly without subroutine bugs. You will remove these lines as you complete each subroutine. Also, note the use of the empty remark statements between subroutines to improve readability.

Enter the main routine and the skeletons of the subroutines as listed above. When finished, run the program. If the program is correct, you should see the regular-size letters

```

H
E
R
B

```

displayed down the screen.

Now, you'll fill in the body of each subroutine with appropriate BASIC statements. As before, if the details can be carried out relatively easily with one or two BASIC statements, you'll write the BASIC statements there. If the task seems too complicated, you'll use a **GOSUB** statement to "bury" the details in another subroutine. As you complete the body of each subroutine, you will remove the **PRINT** statement in the skeleton version.

Here is the finished subroutine to draw the letter H with the **PRINT** statement removed:



```

200 'SUB H
210     DRAW "D60 BU30"
220     DRAW "R40 BU30"
230     DRAW "D60 BL40 BU60"
280     RETURN

```

For this first letter, you should examine each **DRAW** statement to see what it produces on the screen. You should have no difficulty seeing how the letter is drawn. Remember that when you enter this subroutine, you can use the statement

```
GOSUB 200
```

in the immediate mode to see whether the subroutine is working correctly. Of course, you would also have to first give the command

```
SCREEN 1
```

in the immediate mode to bring up the graphics screen.

The bodies of the rest of the subroutines can be written easily.

```

300 'SUB E
310     DRAW "R40 BL40"
320     DRAW "D60 R40 BL40"
330     DRAW "BU30 R20"
340     DRAW "BL20 BU30"
380     RETURN
390 '
400 'SUB R
410     DRAW "D60 BU60 R40"
420     DRAW "D30 L40 BR30"
430     DRAW "F10 D20"
440     DRAW "BL30 BU60"
480     RETURN
490 '
500 'SUB B
510     DRAW "D60 R40 U20"
520     DRAW "H10 E10 U20 L40"
530     DRAW "BD30 R30"
580     DRAW "BL30 BU30"
550     RETURN

```

Again, take a few moments to check the details of how each letter is drawn. Here is the complete program with remark separators:

```

100 'PROGRAM BIG LETTERS
105 'Graphics screen
110     SCREEN 1
115     CLS
120     DRAW "BL110 BU50" 'Position for H
125     GOSUB 200 'Plot H
130     DRAW "BR50" 'Position for E

```

```

135   GOSUB 300 'Plot E
140   DRAW "BR50" 'Position for R
145   GOSUB 400 'Plot R
150   DRAW "BR50" 'Position for B
155   GOSUB 500 'Plot B
160   END
190   '
200   'SUB H
210   DRAW "D60 BU30"
220   DRAW "R40 BU30"
230   DRAW "D60 BL40 BU60"
280   RETURN
290   '
300   'SUB E
310   DRAW "R40 BL40"
320   DRAW "D60 R40 BL40"
330   DRAW "BU30 R20"
340   DRAW "BL20 BU30"
380   RETURN
390   '
400   'SUB R
410   DRAW "D60 BU60 R40"
420   DRAW "D30 L40 BR30"
430   DRAW "F10 D20"
440   DRAW "BL30 BU60"
480   RETURN
490   '
500   'SUB B
510   DRAW "D60 R40 U20"
520   DRAW "H10 E10 U20 L40"
530   DRAW "BD30 R30"
540   DRAW "BL30 BU30"
580   RETURN

```

Enter the balance of the program into your computer. Run the program and verify that it works correctly.

### Example 2 - An Exploration

One of the reasons for using top-down organization is that programs written that way are easy to read. It is just as important for you to learn to read programs as to write them. Indeed, perhaps the best way to learn to write programs is to spend lots of time reading well-written programs.

The program below is given without explanation. Read the program carefully and try to understand what it does. Then enter the program and run it several times. Notice that to accomplish each action in the main routine, the writer needed to use several BASIC statements. Subroutines, therefore, were used. You might

want to make some changes yourself. If you do make changes, try to anticipate the effect of each change before you run the new version of the program.

```

100 'PROGRAM EXPLORE
110   GOSUB 200 'Get data
120   GOSUB 300 'Set graphics screen
130   GOSUB 400 'Plot figures
140   GOSUB 500 'Delay
150 END
190 '
200 'SUB GET DATA
205   CLS
210   PRINT "ENTER NUMBERS IN THE"
215   PRINT "INDICATED RANGE"
220   PRINT
225   PRINT "(60 - 260) ";
230   INPUT X1
235   PRINT "(60 - 260) ";
240   INPUT X2
245   PRINT "(50 - 150) ";
250   INPUT Y1
255   PRINT "(50 - 150) ";
260   INPUT Y2
265   PRINT "(25 - 50) ";
270   INPUT R
275   CLS
280   RETURN
290 '
300 'SUB GRAPHICS SCREEN
310   SCREEN 1
320   CLS
330   RETURN
390 '
400 'SUB PLOT FIGURES
410   LINE (X1, Y1) - (X2, Y2)
420   LINE (X2, Y1) - (X1, Y2)
430   LINE (X1, Y1) - (X2, Y2), 1, B
440   LET X = (X1 + X2) / 2
450   LET Y = (Y1 + Y2) / 2
460   CIRCLE (X, Y), R
470   RETURN
490 '
500 'SUB DELAY
510   PRINT "PRESS RETURN TO"
520   PRINT "END PROGRAM"
530   INPUT A$
540   CLS
550   SCREEN 0
560   RETURN

```

## 4—5 PROBLEMS

1. Write a program with a main routine and two subroutines. The instructions in the first subroutine should draw a rectangular border around the medium-resolution graphics screen. The instructions in the second subroutine should place a large cross at the center of the screen.
2. What is wrong with the following program?

```

100 'PROGRAM PROBLEM
110   GOSUB 200 'First word
120   GOSUB 300 'Second word
130   RETURN
190 '
200 SUB FIRST WORD
210   PRINT "HOWDY"
220   RETURN
290 '
300 SUB SECOND WORD
310   PRINT "DOODY"
320   RETURN

```

3. Write a program to draw a line figure of a car. Use one subroutine to contain the instructions for the body. Use another subroutine to draw the wheels.
4. Describe exactly what will happen if the following program is run.

```

100 'PROGRAM PROBLEM
110   GOSUB 200 'Sub one
120   GOSUB 300 'Sub two
190 '
200 'SUB ONE
210   PRINT "1"
220   RETURN
290 '
300 'SUB TWO
310   PRINT "2"
320   GOSUB 200 'Sub one
330   RETURN

```

## 4—6 PRACTICE TEST

1. What statement must be used at the end of a main routine?  
\_\_\_\_\_
2. What statement should be used at the beginning of a main routine?  
\_\_\_\_\_

3. What statement must be used at the end of a subroutine?

---

4. What statement should be used at the beginning of a subroutine?

---

5. What is the purpose of empty remark statements between subroutines?

---

6. Describe the first phase in developing the main routine.

---

7. Why are the body statements of both the main routine and subroutines indented?

---

8. What will happen if you run the following program?

```

100 'PROGRAM TEST
110   SCREEN 1
120   GOSUB 300 'First sub
130   GOSUB 200 'Second sub
140   END
190 '
200 'SUB SECOND
210   PRINT "ONE"
220   GOSUB 300 'First sub
230   RETURN
290 '
300 'SUB FIRST
310   PRINT "TWO"
320   RETURN
    
```



# ARITHMETIC OPERATIONS AND PRECISION

## 5—1 OBJECTIVES

### Getting Numbers into a BASIC Program

There are only three ways to assign values to variables in a BASIC program. You will examine these ways.

### Doing Arithmetic on the Computer

Ultimately, all mathematics on the computer is done using the simplest arithmetic operations. It is essential to have a clear understanding of how the computer performs these arithmetic operations. The memory locations in which the computer stores numbers can hold numbers up to a maximum size. Thus, not all numbers can be represented exactly in BASIC. You will learn how the computer does arithmetic within these limitations.

### Using Parentheses in Computations

You must type all mathematical expressions on a single line when you enter them into the computer. Some expressions can be handled this way only by organizing parts of the expression in parentheses. Thus, the effective use of parentheses is a necessary skill.

### Learning E Notation for Numbers

The computer must deal with both very large and very small numbers. The computer uses E notation to describe such numbers. You need to be able to recognize and interpret E notation since the computer may print numbers in this form.

### Spacing the Printout

You will learn how to use the standard spacing provided by BASIC for simple output. You will also learn how to format more complex output with the **PRINT USING** statement.

**Working with Program Examples**

As in the previous chapter, you will write BASIC programs to apply what you learned.

**5—2 DISCOVERY EXERCISES****The INPUT Statement**

1. Recall that you can use the **Alt** key to help in typing some BASIC statements. For instance, hold down **Alt** and press **A** to type **AUTO**, and hold down **Alt** with **P** to type **PRINT**. In addition, you can use **Alt** with **I** to type **INPUT**. Turn on the computer and bring up BASICA. Enter the following program:

```

100 'PROGRAM VALUE ASSIGNMENT
110   INPUT A
120   INPUT B
130   INPUT C
140   LET D = A + B + C
150   PRINT D
160   END

```

What do you think will happen if you run this program?

---

Run the program. When the first question mark (the input prompt for **A**) is displayed, enter 2. When the second question mark comes up, enter 3, and finally, at the last question mark, enter 5. Record what happened below.

---

2. Note that the program in step 1 has three **INPUT** statements (lines 110, 120, and 130). Enter

```

110
120

```

What does this do to the program?

---

Clear the screen. Display the program in memory and see if you were right. Then enter

```

130   INPUT A, B, C

```

3. Run the program, and when the input prompt (?) is displayed, enter

2, 3, 5

What happened?

---

Can you input more than one variable at a time in a BASIC program?

---

4. Run the program, and when the input prompt is displayed, enter

2, 3

What happened?

---

Enter

2, 3, 5

and record below what happened.

---

5. Run the program and when the input prompt appears, enter

2, 3, 5, 1

What happened?

---

Press CtrlBreak. What happened?

---

6. Can you enter more numbers than are called for in an **INPUT** statement?

---

What will happen if you do?

---

7. Can you enter fewer numbers than are called for in an **INPUT** statement?

---

What will happen if you do?

---

**The READ Statement**

8. Enter

```
130 READ A, B, C
```

Display the program. What has happened?

---

Run the program and record what happened.

---

9. Now enter

```
135 DATA 2, 3, 5
```

and display the program. What has happened?

---

10. Run the program and record what happened.
- 

From the above, you can infer that whenever a BASIC program contains a **READ** statement, it must also contain another type of statement. What is that statement?

---

11. Name two different methods (other than using a **LET** statement) for assigning values to variables in a program. (Hint: See steps 1 and 8 with 9.)
- 

12. Display the program. Delete the **DATA** statement. Enter

```
155 DATA 2, 3, 5
```

and display the program again. What has happened?

---

13. Run the program and record the output.
- 

Does it appear to make any difference where in the program the **DATA** statement appears?

---

14. Clear the program in memory by entering **NEW** and enter the program below:

```

100 'PROGRAM DATA ERROR
110   READ A
120   PRINT A
130   READ A
140   PRINT A
150   READ A
160   PRINT A
170   DATA 2, 3
180 END

```

Note that there are only two numbers in the **DATA** statement in line 170. What will happen if you run the program?

---

Try it and see if you were correct. Record the output.

---

Is the **Out of data** message associated with the **READ** statement or the **DATA** statement?

---

Edit line 170 as follows:

```

170   DATA 2, 3, 5

```

Run the program. What happened?

---

15. Delete the **DATA** statement in line 170 from the program. Now enter the following statements:

```

105   DATA 2
115   DATA 3
125   DATA 5

```

Display the program in memory. What has taken place?

---

16. If you run the program, what do you think will be displayed?

---

Run the program and see if you were correct. Record the output below.

---



17. Can you have more than one **DATA** statement in a BASIC program?

---

Does it seem to make any difference where in the program the **DATA** statements appear?

---

### Arithmetic on the Computer

18. Recall from Chapter 1 the following symbols for arithmetic operations:

+	<i>Addition</i>
-	<i>Subtraction</i>
*	<i>Multiplication</i>
/	<i>Division</i>

To review the arithmetic operations, clear the memory and enter

```

100 'PROGRAM ARITHMETIC
110   INPUT A
120   INPUT B
130   LET C = A * B - B / 3
140   PRINT C
150   END



```

Display the program and study it briefly. If you were to run the program and enter 2 for **A** and 3 for **B**, what do you think would be displayed?

---

Run the program. Enter 2 and 3 at the input prompts, and write down what happened.

---

19. Clear the memory and enter (Note: ^ is obtained with  and .)

```

100 'PROGRAM EXPONENTS
110   LET A = 3 * 3
120   LET B = 3 ^ 2
130   PRINT A
140   PRINT B
150   END

```

Display the program and make sure it is correct. Now run the program and record what was displayed.

---

Compare the numbers printed out to the expressions in the lines where they were computed. See if you can figure out what is taking place.

20. Edit lines 110 and 120 as follows:

```
110 LET A = 3 * 3 * 3
120 LET B = 3 ^ 3
```

Run the program. What was displayed?

---

21. Now edit lines 110 and 120 as follows:

```
110 LET A = 2 * 2 * 2 * 2
120 LET B = 2 ^ 4
```

Run the program. What was displayed?

---

What is the ^ symbol used for?

---

22. Remember from your introductory algebra course (if you haven't had algebra, don't panic) that if you want to indicate that a number is to be multiplied by itself three times, you can use an exponent. If the number were 2, you would write the expression as

$$2^3$$

How would this expression be written in BASIC using the ^ symbol? (Hint: See steps 19, 20, and 21.)

---

23. Fill in the operators (symbols) that call for the following arithmetic operations:
- Division

---

Exponentiation

---

Multiplication

---

24. Clear the memory and enter

```

100 'PROGRAM ORDER OF OPERATIONS
110   LET A = 4 + 2 * 6
120   LET B = (4 + 2) * 6
130   LET C = 4 + (2 * 6)
140   PRINT A
150   PRINT B
160   PRINT C
170   END

```

The two points of this program are (1) the order in which the arithmetic is done, and (2) the effect of the parentheses. If you look closely, you will see that the same numbers and operations are involved in each of the calculations in lines 110, 120, and 130. The only difference in the lines is the grouping with parentheses. Run the program and record what was displayed.

---

Study the program and the numbers the computer displays until you see what is taking place in the program. The computer uses set rules in such situations. You will study these rules later in the chapter.

### Limitations of the Computer

25. Clear the memory and enter the following direct mode statement:

```
PRINT 2 / 3
```

What number was displayed?

---

The decimal displayed is approximately 2/3 but it is not exactly equal to 2/3.

26. Enter

```
PRINT 1234 ^ 13
```

Write down the number displayed below **Overflow**.

---

Enter

```
PRINT 4321 ^ 13
```

How does this number compare with the number you wrote above?

---

The overflow error indicates that the result is outside the limitations of BASICA. The number **1.701412E+38** (or its negative) is displayed in all such cases.

Enter

```
PRINT 1234 ^ 12
```

Record the number below.

---

## E Notation

27. Clear the memory and enter the following program:

```
100 'PROGRAM E NOTATION
110 LET A = 3 * 10000
120 LET B = 3 * 1000000
130 LET C = 3 * 1000000000
140 PRINT A
150 PRINT B
160 PRINT C
170 END
```

Run the program and record the output.

---

Can you explain the different form in which the computer displayed the numbers? (Hint: Count the number of zeros in the multipliers in lines 110, 120, and 130.)

---

Describe the meaning of E+37 in the number you recorded at the end of step 26.

---

28. Display the program. Notice the ! mark and # sign in lines 120 and 130. These marks indicate the computer is storing the numbers in single-precision format (!) or double-precision format (#). You should not concern yourself about their occurrence here. Edit lines 110, 120, and 130 as follows:

```
110 LET A = 4 / 1000000
120 LET B = 4 / 100000000
130 LET C = 4 / 10000000000
```

Run the program and record the output.

---

Again, can you see what is taking place in the output? Count the zeros in the denominators in lines 110, 120, and 130.

29. If an E shows up in a number displayed by the computer, what does it mean? Explain in your own words.
- 

Do not be concerned if you do not understand the purpose of the E notation fully. You will return to it later.

30. Clear the memory and enter the following program:

```

100 'PROGRAM SPACING OUTPUT
110   READ A, B, C, D, E, F, G
121   PRINT A
122   PRINT B
123   PRINT C
124   PRINT D
125   PRINT E
126   PRINT F
127   PRINT G
120   DATA 10, 12, 8, 9, 73, 60, 82
150   END

```

Run the program and record what happened.

---

31. Insert a comma at the end of each **PRINT** statement in lines 121-127. For example, line 121 would look like:

```

121   PRINT A,

```

Run the program and record what happened.

---

32. Use the **EDIT** command to replace the commas after the **PRINT** statements with semicolons. Run the program and record what happened.
- 

33. If a variable in a **PRINT** statement is not followed by punctuation marks, how does the computer space the output? (Hint: See step 30.)
- 

Suppose the variable is followed by a comma?

---

What will happen if the variable is followed by a semicolon?

---



34. Renumber the program in steps of 10 starting at 100 (**RENUM 100**). Add line 115 and modify the **PRINT** statements so the program looks like this:

```

100 'PROGRAM SPACING OUTPUT
110   READ A, B, C, D, E, F, G
115   LET P = 5
120   PRINT TAB(P); A
130   PRINT TAB(P + 5); B
140   PRINT TAB(P + 10); C
150   PRINT TAB(P + 15); D
160   PRINT TAB(P + 20); E
170   PRINT TAB(P + 25); F
180   PRINT TAB(P + 30); G
190   DATA 10, 12, 8, 9, 73, 60, 82
200   END

```

Run the program and record what happened.

---

35. Edit line 115 as follows:

```

115   LET P = 10

```

Run the program and record what happened.

---

36. What does the **TAB** in the print statement appear to control?
- 

37. Let's look at a special **PRINT** statement. Clear the memory. Enter the following program noting carefully the spacing in line 110:

```

100 'PROGRAM OUTPUT FORMAT
110   LET F$ = "#.# #.## #.###"
120   LET A = 1.1234
130   LET B = 2.1234
140   LET C = 3.1234
150   PRINT USING F$; A; B; C
160   END

```

Run the program. How many decimal places are printed for **A**, **B**, and **C**?

---

How many spaces are there between the output of **B** and **C**?

---

Display the program. Study line 110 to see the connection between the program output and **F\$**.

38. Insert lines 104 and 106, carefully observing the spacing in line 110 and noting that the \ is found on the key next to Z.

```
104    LET S$ = "\\ \\ \"
106    PRINT USING S$; "1ST"; "2ND"; "3RD"
```

Display the program. Run the program. Where are the column headings placed?

---

39. This concludes the computer work for now. Turn off the computer.

## 5—3 DISCUSSION

### Getting Numbers into a BASIC Program

In Chapter 2 you learned one way to assign values to variables inside a program. For example,

```
100 LET A = 6
```

assigns the value 6 to the variable **A** and stores the number under that variable name. This method has limitations, so you need to know other ways to assign values to variables in a BASIC program. Let's look first at the **INPUT** statement and how it is used. An example might be

```
260 INPUT G
```

When the computer executes this line, it displays a question mark as a prompt that input is expected. Then it waits for you to enter a number. In the case above, the number typed in is known as **G**.

A single **INPUT** statement can call for more than one variable. For instance,

```
420 INPUT A, B, C, D
```

In this case, the computer displays the same **INPUT** prompt (the question mark) but now the computer expects you to type in four numbers separated by commas. If you enter more than or less than four numbers, the computer displays **Redo from start**. If you wish to interrupt the program at this time, hold down **[Ctrl]** and press **[Break]**. Otherwise, reenter four numbers.

One last method to assign values to variables is with **READ** and **DATA** statements. The computer handles the **READ** statement

```
100 READ A, B, C, D
```

in the same way as the **INPUT** statement, with two exceptions. First, the computer does not stop. There is no need to, as you will see. Second, the computer reads the numbers to be input from **DATA** statements contained within the program rather than displaying the **INPUT** prompt and waiting for you to enter those numbers.

The following program fragment illustrates how the **READ** and **DATA** statements work:

```

100  READ A, B, C, D
110  LET E = A + B + C + D
120  PRINT E
130  DATA 25, 3, 17, 12

```

The computer reads four numbers from the **DATA** statement and prints out the sum of the numbers. It makes no difference where in the program the **DATA** statement appears. Also, there can be more than one **DATA** statement, and **DATA** statements need not be grouped together at the same place in a program. When the **READ** statement calls for numbers, the computer takes them in order from the **DATA** statements, beginning with the lowest numbered statement. If a **READ** statement still requests numbers after the computer exhausts all the numbers in the available **DATA** statements, the computer types an **Out of data** message and stops.

In summary, there are three methods for assigning values to variables in BASIC programs. They are: (1) **LET** statements, (2) **INPUT** statements, and (3) **READ** and **DATA** statements. Each of these methods can be used to advantage. You will become familiar with the advantages and disadvantages of each method as you spend more time writing programs.

■ You can assign values to variables with:

- (1) **LET**
- (2) **INPUT**
- (3) **READ and DATA**

## Doing Arithmetic on the Computer

You are concerned with five arithmetic operations. These are addition, subtraction, multiplication, division, and exponentiation (+, -, /, \*, and ^). The first four are certainly familiar to you, and the only difficulty with the last (exponentiation) is the intimidating word used to designate the process.

The exponentiation operation is represented by the ^ symbol. Exponentiation means “raising to the power.” Therefore,  $3^4$  means “3 raised to the fourth power,” which in turn means 3 multiplied by itself four times, giving 81 as the result.

You need a full understanding of the order in which the computer performs arithmetic operations. An example will illustrate the point. Consider the following expression:

$$2 + 3^2 / 5 - 1$$

If the computer simply went through the expression from left to right performing operations as it met them, the computer would add 2 and 3 (giving 5), raise 5 to the second power, (giving 25), divide 25 by 5 (giving 5), subtract 1 from that, and arrive at an answer of 4. However, suppose the computer performs addition and subtraction first, then exponentiation, then multiplication and division. This process would give 5 raised to the second power divided by 4. Then the computer would exponentiate giving 25 divided by 4, and arrive at an answer of 6.25.

You could experiment with different rules for the order of arithmetic operations and might get different answers each time. The point is that there are well-defined rules in BASIC for the order and priority of arithmetic operations. Here they are:

The computer performs operations from left to right using the priority rules below. The priority for arithmetic operations is

1st:	Exponentiation
2nd:	Multiplication and division
3rd:	Addition and subtraction

Follow the computer as it performs the familiar expression

$$2 + 3^2/5 - 1$$

The computer scans left to right for exponentiation. Since there is an exponentiation indicated ( $3^2$ ), it is done first. Now the expression is

$$2 + 9/5 - 1$$

Scanning from left to right, the computer looks for exponentiation and, finding none, scans left to right for the operations with the next highest priority (multiplication and division). The division is therefore done next, with the following result:

$$2 + 1.8 - 1$$

Since there are no more multiplications or divisions left in the expression, the computer scans again from left to right, this time for addition and subtraction. The addition gives

$$3.8 - 1$$

and the final subtraction produces the answer of 2.8.

Review the rules for order and priority of arithmetic operations until they become second nature to you. You will look at the rules again when the use of parentheses is discussed in the next section.

Numbers are stored in memory locations in the computer. In each storage location, the computer can store numbers up to some maximum size. For example,

$$1234^13$$

is too large a number for BASICA to handle. If you attempt to compute this number in BASICA, you will get an overflow error message.

Another problem can arise. Some numbers, such as  $2/3$  and  $5/9$ , cannot be stored exactly in BASICA. For example, the direct mode statements

```
LET A = 2 / 3
PRINT A
```



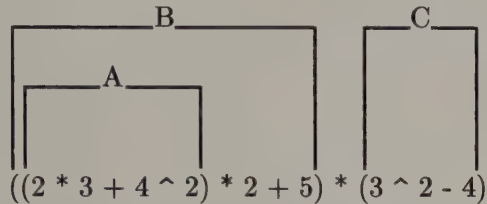
will display

**.6666667**

The exact value is the infinitely repeating decimal  $0.6666666666 \dots$ . The computer rounds off the value at the seventh place after the decimal point. This kind of round-off error can be troublesome if complex computations are required.

### Using Parentheses in Computations

The rules for order and priority of arithmetic operations are not the whole issue, however. Consider the following more complicated example:

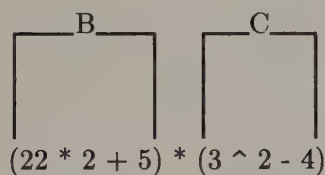


The difference between this expression and the ones you have been studying is the use of parentheses to group parts of the expression. Follow this example in detail to see how the computer attacks the arithmetic.

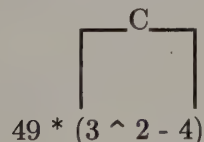
The computer starts by scanning from left to right and meets the left parenthesis of B. It then looks inside to see if there are any other left parentheses and finds the one of A. The next parenthesis met is the right parenthesis of A. At this point, the computer has isolated the first group of operations to be done, the expression in parentheses A:

$$2 * 3 + 4 ^ 2$$

The computer evaluates that expression using the order and priority rules. The result is 22 (check it). Now the problem has become



On the next scan, the computer isolates the parentheses of B, does the arithmetic inside, and reduces the problem to



Since only the parentheses of C are left, the computer does the arithmetic inside, arriving at

$$49 * 5$$

which after the final multiplication results in the answer of 245.



Thus, if parentheses are nested, the computer works out from the deepest set, working from left to right. When a set of parentheses is removed, the arithmetic operations inside are done according to the order and priority rules already given.

Follow this excellent rule of thumb: If there is any confusion about how the computer will evaluate an expression, use extra parentheses. Too many cannot hurt, but too few certainly can.

### Learning E Notation for Numbers

In BASIC, numbers are sometimes printed out in what is known as the E notation. Examples of E notation are 2.145E+06 or 6.0323E-07.

It is easy to see why a special notation is needed for either very large or very small numbers. The computer usually prints out 9 digits for a number. A problem comes up if you want the computer to print out a variable whose value is, for instance, 45612800000 (11 digits). The computer prints this number as 4.56128E+10, which means that the decimal point belongs 10 places to the right of its present position. A variable whose value is 8954000000000 is printed as 8.954E+12. The E+12 means that the decimal point belongs 12 places to the right. Very small numbers are expressed in the same way. A variable whose value is 0.0000000683 is printed as 6.83E-08. The E-08 means that the decimal point belongs 8 places to the left. The table below should help you understand how to convert from decimal to E notation or from E notation back to decimal notation.

Decimal Form	E Notation
2630000	2.63E+06
263000	2.63E+05
26300	2.63E+04
2630	2.63E+03
263	2.63E+02
26.3	2.63E+01
2.63	2.63
0.263	2.63E-01
0.0263	2.63E-02
0.00263	2.63E-03
0.000263	2.63E-04
0.0000263	2.63E-05
0.00000263	2.63E-06

To change from E notation to decimal notation, look at the sign following the E. If the sign after the E is +, move the decimal point to the right as many places as the number after the +. If the sign after the E is -, move the decimal point to the left. To convert from decimal to E notation, just reverse the process.

Actually, you shouldn't get very tense about the E notation since you will rarely use it when setting up programs on the computer. The main reason for bringing up the issue is that the computer may display numbers in the E notation. Consequently, you should be able to recognize what is happening.

## Spacing the Printout

BASICA has a “built-in” standard spacing mechanism that prints two or five numbers spaced equally on one line depending upon whether you are using a 40- or 80-column text screen. When quantities in a **PRINT** statement are separated by commas, the computer uses standard spacing. The comma signals the computer to move to the next print position on the line. If the computer is already at the last position on a line and encounters a comma in a **PRINT** statement, it does a carriage return and prints the number on the first position on the next line. Thus, for a 40-column screen

```
100 PRINT A, B, C
```

tells the computer to print the numeric values of **A** and **B** evenly spaced across a line in the two standard positions. The numeric value of **C** would be printed below the value of **A** on the next line.

- **Commas in PRINT statements produce two columns per line on a 40-column screen and five columns per line on an 80-column screen.**

When used in a **PRINT** statement, a semicolon between variables instructs the computer to space in a nonstandard manner. For instance, the statement

```
100 PRINT A; B
```

causes output to be spaced closer together than the statement

```
100 PRINT A, B
```

Finally, you can control the spacing of a line more precisely by using the **TAB** feature in **PRINT** statements. The **TAB** feature works the same way as a tabulator setting on a typewriter. There are 40 or 80 tab positions available, depending on the screen width you are using.

The statement

```
100 PRINT TAB(5); A; TAB(25); B
```

signals the computer to space to the 5th printing position, print the numeric value of **A**, space to the 25th printing position, and finally print the numeric value of **B**. If **B** has too many digits to fit on the line, the computer will place **B** at the beginning of the next line.

It is also possible to use a variable tab setting, for instance,

```
100 PRINT TAB(X); A
```

This statement causes the computer to look up the value of **X**, then space to the printing position determined by the largest integer in **X** (for example, if **X** = 23.1435, the computer will space to the 23rd printing position), then print the numeric value of **A**.

■ Use the TAB function to produce variable spacing.

You can use the empty **PRINT** statement by itself to produce vertical spacing on the output. For instance, if the computer sees

```
100 PRINT
```

it looks for the quantity to be printed and finds none. It then looks for punctuation and, finding none, orders a carriage return and advances the cursor one line. If you want two or three empty lines in a printout, use two or three empty **PRINT** statements.

The **PRINT USING** statement can be used to format and space printed numbers and strings precisely. For example, the program fragment

```
100 LET F$ = "###.##  \"
110 LET A = 234.87654
120 LET B$ = "BINGO"
130 PRINT USING F$; A; B$
140 END
```

tells the computer to print out

```
234.88 BIN
```

when it is run. Notice that numeric and string information may be called for in the same **PRINT USING** statement. Both string and numeric information is cut off to fit the format called for by the **PRINT USING** statement. The characters in the string **F\$** form a “picture” of how you want the format to appear. For example if **F\$** = “###.##” and the number 366.153 is printed using the format string **F\$**, the result is 366.15. If 1238.156 is printed in the same format, the result would be 238.16.

To format strings, use the backslash (\) symbol. If **F\$** = “\ \” and the string constant **ABCDEF** is printed using this format, the result is **ABCD**.

For more details about the **PRINT USING** statement see the BASIC reference manual for your computer.

## 5—4 WORKING WITH PROGRAM EXAMPLES

One way to improve your programming and debugging skills is to study program examples. The examples in this chapter are simple but illustrative. Study each one carefully until you are certain that you understand all the details. You might want to enter the programs into the computer and run them to verify that they work as intended.

### Example 1 - Unit Prices

Your problem is to write a program to compute unit prices on supermarket items. Let **TOTAL** stand for the total price, **NUMBER** for the number of items, and **UNIT-PRICE** for the unit price. You can compute the unit price as follows:

$$\text{UNITPRICE} = \text{TOTAL} / \text{NUMBER}$$

As an example, suppose that a case of 12 large cans of fruit juice costs \$6.96. The unit cost per can would be

$$\text{UNITPRICE} = 6.96 / 12 = \$0.58$$

The program can be designed to produce the following output:

```
WHAT IS THE TOTAL PRICE? (You enter value)
WHAT IS THE NUMBER OF ITEMS? (You enter value)
UNIT PRICE IS (Computer displays value)
```

On a piece of paper, you would create a skeleton of the main routine without line numbers. It might look as follows:

```
PROGRAM UNIT PRICES
  Enter total price
  Enter number of items
  Compute and display unit price
END
```

You can now write the program skeleton with the necessary GOSUBs.

```
100 'PROGRAM UNIT PRICES
110   GOSUB 200 'Enter total price
120   GOSUB 300 'Enter number of items
130   GOSUB 400 'Compute and display unit price
140 END
```

To flesh out the first subroutine, you can request the user to enter the total price and assign it to a variable as follows:

```
200 'SUB ENTER TOTAL PRICE
210   PRINT "WHAT IS THE TOTAL PRICE ";
220   INPUT TOTAL
230 RETURN
```

The second subroutine beginning at line 300 is much like the first. It might look like this:

```
300 'SUB ENTER NUMBER OF ITEMS
310   PRINT "WHAT IS THE NUMBER OF ITEMS ";
320   INPUT NUMBER
330 RETURN
```

The last subroutine could be

```
400 'SUB COMPUTE AND DISPLAY UNIT PRICE
410   LET UNITPRICE = TOTAL / NUMBER
420   PRINT "UNIT PRICE IS "; UNITPRICE
430 RETURN
```



The complete program with blank remark statements as separators appears below:

```

100 'PROGRAM UNIT PRICES
110   GOSUB 200 'Enter total price
120   GOSUB 300 'Enter number of items
130   GOSUB 400 'Compute and display unit price
140   END
190 '
200 'SUB ENTER TOTAL PRICE
210   PRINT "WHAT IS THE TOTAL PRICE ";
220   INPUT TOTAL
230   RETURN
290 '
300 'SUB ENTER NUMBER OF ITEMS
310   PRINT "WHAT IS THE NUMBER OF ITEMS ";
320   INPUT NUMBER
330   RETURN
390 '
400 'SUB COMPUTE AND DISPLAY UNIT PRICE
410   LET UNITPRICE = TOTAL / NUMBER
420   PRINT "UNIT PRICE IS "; UNITPRICE
430   RETURN

```

The use of top-down organization in a simple problem like this may still seem to be a case of overkill to you. However, you should get in the habit of writing programs this way. In more complicated problems, the top-down method will be a lifesaver.

### Example 2 - Converting Temperature

The relationship between temperature measured in degrees Fahrenheit and in degrees Celsius is

$$C = (5 / 9) (F - 32)$$

Let **C** stand for degrees Celsius and **F** stand for degrees Fahrenheit. If, for example, **F** is 212, then **C** is determined to be

$$C = (5 / 9) (212 - 32) = 100$$

The problem is to write a program that will produce the following output:

```

ENTER THE TEMP IN DEGREES F
?   (You enter value of F)
(value of F) DEGREES F IS (answer) DEGREES C

```

Let's begin with a paper-and-pencil outline of the tasks to be performed in the main routine.



```

PROGRAM TEMPERATURE CONVERSION
  Get value for F
  Compute the value for C
  Output the converted value
END

```

You could replace these comments with a few BASIC statements that would accomplish the tasks described. Instead, use the top-down method. Habits developed at this point will become very important later on.

Here is the completed main routine:

```

100 'PROGRAM TEMPERATURE CONVERSION
110   GOSUB 200 'Get value for F
120   GOSUB 300 'Compute the value for C
130   GOSUB 400 'Output the converted value
140   END

```

Next, you should write an outline of the first subroutine. Here is how it might look:

```

SUB F VALUE
  Print input prompt
  Input value of F
RETURN

```

With this outline, you can complete the subroutine.

```

200 'SUB F VALUE
210   PRINT "ENTER THE TEMP IN DEGREES F"
220   INPUT F
230   RETURN

```

Here is an outline of the second subroutine:

```

SUB CONVERT
  Compute degrees C
RETURN

```

Here is the completed subroutine:

```

300 'SUB CONVERT
310   LET C = (5 / 9) * (F - 32)
320   RETURN

```

Finally, here is the outline of the last subroutine:

```

SUB OUTPUT
  Output results
RETURN

```

The subroutine is easy to complete.

```

400 'SUB OUTPUT
410   PRINT F; " DEGREES F IS ";
420   PRINT C; " DEGREES C"
430   RETURN

```

Here is the complete program with blank remark statements to separate the routines:

```

100 'PROGRAM TEMPERATURE CONVERSION
110   GOSUB 200 'Get value for F
120   GOSUB 300 'Compute the value for C
130   GOSUB 400 'Output the converted value
140   END
190 '
200 'SUB F VALUE
210   PRINT "ENTER THE TEMP IN DEGREES F"
220   INPUT F
230   RETURN
290 '
300 'SUB CONVERT
310   LET C = (5 / 9) * (F - 32)
320   RETURN
390 '
400 'SUB OUTPUT
410   PRINT F; " DEGREES F IS ";
420   PRINT C; " DEGREES C"
430   RETURN

```

You could have written the same program in only a few lines. Here is a version that does exactly the same thing.

```

100 'PROGRAM TEMPERATURE CONVERSION
110   PRINT "ENTER THE TEMP IN DEGREES F"
120   INPUT F
130   LET C = (5 / 9) * (F - 32)
140   PRINT F; " DEGREES F IS ";
150   PRINT C; " DEGREES C"
160   END

```

For simple programs like this, beginners usually fail to see the point of top-down organization and prefer to write the shorter version of the program. In the shorter version, one is usually thinking of the next statement needed rather than what block is needed. As the programs get longer and more complex, the top-down method still works well. Quick-and-dirty methods of programming lead to complicated tangles of logic that are difficult to write, hard to understand, and almost impossible to change.

### Example 3 - Sum and Product of Numbers

This example computes the sum and product of any two numbers. The numbers will be assigned by **READ** and **DATA** statements.

Again, consider how you want the output to appear.

```

VALUE OF A IS  (Computer prints value)
VALUE OF B IS  (Computer prints value)
SUM OF A AND B IS  (Computer prints sum)
PRODUCT OF A AND B IS  (Computer prints product)

```

A skeleton of the main routine without line numbers might look like this:

```

PROGRAM SUM AND PRODUCT
  Read and display values
  Compute and display sum and product
END

```

The main routine can now be written with the necessary GOSUBs.

```

100 'PROGRAM SUM AND PRODUCT
110   GOSUB 200 'Read and display values
120   GOSUB 300 'Compute and display sum and product
130   END

```

You can write the first subroutine as follows:

```

200 'SUB READ AND DISPLAY VALUES
210   READ A, B
220   PRINT "VALUE OF A IS "; A
230   PRINT "VALUE OF B IS "; B
240   DATA 5, 7
250   RETURN

```

The second subroutine computes the sum and product as follows:

```

300 'SUB COMPUTE AND DISPLAY SUM AND PRODUCT
310   PRINT "SUM OF A AND B IS "; A + B
320   PRINT "PRODUCT OF A AND B IS "; A * B
330   RETURN

```

The complete program with remark separators follows:

```

100 'PROGRAM SUM AND PRODUCT
110   GOSUB 200 'Read and display values
120   GOSUB 300 'Compute and display sum and product
130   END
190 '
200 'SUB READ AND DISPLAY VALUES
210   READ A, B
220   PRINT "VALUE OF A IS "; A
230   PRINT "VALUE OF B IS "; B
240   DATA 5,7
250   RETURN
290 '

```

```

300 'SUB COMPUTE AND DISPLAY SUM AND PRODUCT
310     PRINT "SUM OF A AND B IS "; A + B
320     PRINT "PRODUCT OF A AND B IS "; A * B
330     RETURN

```

## 5—5 PROBLEMS

1. Write a program that will read four numbers 10, 9, 1, and 2, from a **DATA** statement, putting the numbers in **A**, **B**, **C**, and **D**, respectively. Add the first two numbers, assigning the sum to **S**. Then compute the product of all four numbers, putting the result in **P**. Print out the value of **S** and **P** on the same line.
2. Write a program that calls for the input of four numbers, then print back the numbers in reverse order. For example, if you enter 5, 2, 11, 12, the computer should print 12, 11, 2, 5. The program must work for any set of four numbers that you decide to enter.
3. Write a program to read values for the variables **A**, **B**, **C**, and **D** from a **DATA** statement. Put any numbers you wish in the **DATA** statement. Then the computer should print out the numbers vertically with **B** below **A**, **C** below **B**, and **D** below **C**.
4. One of the ratios used to judge the health of a business is the acid-test ratio. The acid-test ratio is the sum of cash, marketable securities, and receivables, divided by the current liabilities. Write a program to call for the input of the necessary quantities, then compute and output the acid-test ratio.
5. The volume of a box can be computed as

$$V = LWH$$

where **L**, **W**, and **H** are the length, width, and height, respectively. If these are all measured in centimeters, for example, the volume will be in cubic centimeters. Write a program that will produce the following output when run:

```

LENGTH (CM) ? (You enter value for L)
WIDTH (CM) ? (You enter value for W)
HEIGHT (CM) ? (You enter value for H)
VOLUME IS (Computer prints value for V) CUBIC CM

```

6. If an object is dropped near the surface of the earth, the distance it will fall in a given time can be determined by

$$S = 16T^2$$

where **S** is the distance fallen (in feet) and **T** is the time of fall in seconds. Write a program that will produce the following output:

```

TIME OF FALL (SEC) ? (You enter value for T)
OBJECT FALLS (Computer types out value for S) FEET

```

7. A **DATA** statement contains examination grades for a class of ten students. Write a program to compute and print out the class average. Try out the program on sample data of your choice.
8. Simple interest on an investment is computed according to the following rule:

$$I = (P) (R / 100) (T / 365)$$

where P is the principal invested at an annual interest rate R (expressed in percent) for a time T (expressed in days). Write a program that will generate the display shown below:

```

WHAT IS THE PRINCIPAL
?   (You enter value for principal)
WHAT IS THE ANNUAL INTEREST RATE (%)
?   (You enter value for interest rate)
WHAT IS THE TERM IN DAYS
?   (You enter value for term)

FOR AN INVESTMENT OF
(Computer prints value of principal)
AT AN ANNUAL INTEREST RATE OF
(Computer prints value for interest rate)
PERCENT, INVESTED FOR
(Computer prints value for term)
DAYS, THE INTEREST IS
(Computer prints value for interest)

```

9. If compound interest is paid, the true annual interest rate is higher than the nominal rate quoted for the investment. The following formula computes this true annual interest rate:

$$T = ((1 + R/(100 * M))^M - 1) * 100$$

In this expression, T is the true annual interest rate in percent, R is the nominal interest rate in percent, and M is the number of times the investment is compounded per year. Write a program that will produce the following output:

```

QUOTED INTEREST RATE (PERCENT)
?   (You enter value)
NUMBER OF TIMES COMPOUNDED PER YEAR
?   (You enter value)
TRUE ANNUAL INTEREST RATE IS
(Computer prints value)

```

10. If an amount of money, P, is left to accumulate interest at I percent compounded J times per year for N years, the value of the investment will be

$$T = P * (1 + I/(100 * J))^(J * N)$$

Write a program that calls for the input of P, I, J, and N. Run the program as needed to get the value of \$1000 invested at 8 percent for 2 years compounded annually (J = 1), semiannually (J = 2), monthly (J = 12), weekly (J = 52), and daily (J = 365). If a savings and loan company bases a big advertising campaign on the fact that it computes the interest every day instead of each week, should you get excited?



11. If an amount of money,  $P$ , is left to accumulate interest at a rate of  $I$  percent per year for  $N$  years, compounded annually, the money will grow to a total amount  $T$  given by

$$T = P * (1 + I/100)^N$$

As an example, if  $P = \$1000$ ,  $I = 6\%$ , and  $N = 5$  years,

$$T = 1000 * (1 + 6/100)^5 = 1338.23$$

Write a program that will produce the following output:

```
INITIAL INVESTMENT ? (You enter value)
ANNUAL INTEREST RATE (%) ? (You enter value)
YEARS LEFT TO ACCRUE INTEREST ? (You enter value)
TOTAL VALUE IS (Computer prints value)
```

## 5—6 PRACTICE TEST

1. List three different ways that values can be assigned to variables in a BASIC program.  
\_\_\_\_\_
2. What kind of statement must go with a **READ** statement?  
\_\_\_\_\_
3. How many standard print columns per line are provided for on a 40 character screen when the print quantities are separated by commas?  
\_\_\_\_\_
4. How many **DATA** statements may appear in a program?  
\_\_\_\_\_
5. What is the purpose of the **TAB** feature in BASIC?  
\_\_\_\_\_
6. If there are no parentheses in an arithmetic expression containing the five arithmetic operations discussed in the chapter, which operation will be done first?  
\_\_\_\_\_
7. Convert 1.23E-3 to decimal notation.  
\_\_\_\_\_

8. Convert 123,450,000,000 to E notation.

---

9. What is the difference between using commas and semicolons to separate quantities in a **PRINT** statement?

---

10. Miles can be converted to kilometers by multiplying by 1.609. Thus, 10 miles equal 16.09 kilometers, and so on. Write a program that will produce the following output when run:

**INPUT NO. OF MILES?** *(You enter value)*  
*(Computer prints value for miles)* **MILES EQUAL**  
*(Computer prints answer)* **KILOMETERS**



# THE LOOP BLOCK

## 6—1 OBJECTIVES

### Creating an Infinite Loop Block

You will be shown how to create a block of statements that the computer repeats again and again until you tell it to stop.

### Writing a Counting Loop

One of the most common applications of a loop is to count up to some point and then stop. You will learn how to write a counting loop.

### Learning the Standard Loop Block

The standard loop block provides an exit condition. The computer stays in the loop until the exit condition becomes true, at which time the computer leaves the loop. This standard loop block is one of the three fundamental blocks needed to solve any computer programming problem. You will learn how to write the standard loop block and how to use it in programs.

### Using the FOR/NEXT Abbreviation

Some loop problems are solved best by using an abbreviation of the standard loop block. You will learn how to use the **FOR/NEXT** abbreviation in programs.

### Working with Program Examples

You will apply the ideas you learn to programming problems.

## 6—2 DISCOVERY EXERCISES

### An Infinite Loop Block

1. Turn on the computer, bring up BASICA and enter the following program:

```
100 'PROGRAM LOOP DEMO
110   PRINT "ONE"
120   PRINT "TWO"
130   PRINT "THREE"
140   PRINT "FOUR"
150   END
```

What will happen if you run the program?

---

Run the program and see if you were right.

2. Now insert statement that you haven't used before. Enter the following statement:

```
105   GOTO 120
```

What effect do you think this will have on the program?

---

Run the program. What was the output this time?

---

3. Enter the following statement:

```
125   GOTO 140
```

Run the program. What was the output this time?

---

4. Explain in your own words what the **GOTO** statement does?
- 

5. Remove lines 105 and 125 from the program. Insert the following line:

```
145   GOTO 110
```

You have seen two previous cases where the **GOTO** statement told the computer to jump forward over statements. What does this statement tell the computer to do?

---

Think for a second about the implications of your answer. Will the computer stop if the program is run?

---



6. Run the program. When you get tired watching the display, interrupt the program (hold down **Ctrl** and also press **Break**). That will stop the program. How long would the program have run?
- 

7. You have just seen an infinite loop. Let's put the loop in the standard form. Display the program and change it until it looks like this:

```

100 'PROGRAM LOOP DEMO
110   'LOOP FOREVER
120     PRINT "ONE"
130     PRINT "TWO"
140     PRINT "THREE"
150     PRINT "FOUR"
160     GOTO 110
170   'END LOOP
180 END

```

Notice that the body of the main routine is an infinite loop. The loop begins and ends with remark statements in lines 110 and 170, respectively. The body statements of the loop are indented two spaces further to the right to show that they belong to the loop. The **GOTO** statement in line 160 sends the computer back to the beginning of the loop.

8. Run the program again. Confirm that it is in an infinite loop. When you have seen enough, use **CtrlBreak** to stop the computer.

### A Counting Loop

9. Use the **NEW** command to clear the program from memory. Then enter the following program:

```

100 'PROGRAM COUNTING LOOP
110   LET N = 1
120   'LOOP FOREVER
130     PRINT N
140     LET N = N + 1
150     GOTO 120
160   'END LOOP
170 END

```

Take a few minutes to examine the body of the program. What is the value of **N** when the computer enters the infinite loop?

---

What do you think the value of **N** will be when the computer reaches line 150 the first time?

---

What do you think will happen if you run the program?

---

10. Run the program and watch the output. When you have seen enough, stop the program. (Remember you can do this with `CtrlBreak`.)
11. So far this is just an infinite loop that will go on counting until you tell the computer to stop. Now change the program so that the computer will stop itself. Insert the following line (the greater than sign, `>`, is found above the period):

```
125      IF N > 100 THEN 160
```

Edit line 120 to read as follows:

```
120      'LOOP
```

Display the program and study the loop for a few moments. What do you think will happen if you run the program?

---

12. Run the program. What was the last number printed on the screen?
- 

As you saw, now the computer stops itself at some predetermined value of N.

13. Change line 125 to read as follows:

```
125      IF N > 20 THEN 160
```

What will happen if you run the program now?

---

Run the program. What happened?

---

14. In step 9, the computer stayed in the loop until you interrupted the program. In this new counting loop, the computer stayed in the loop until the value of N became greater than some specified value. The new version of the loop is called a counting loop. If you wanted the computer to stop when 35 had been printed, what statement would you put in line 125?
- 

### The Standard Loop Block

15. Let's go on to another form of the loop. Clear the program in memory with the **NEW** command. Clear the screen and enter the following program:

```

100 'PROGRAM STANDARD LOOP
110 'LOOP
120     INPUT A$
130     IF A$ = "STOP" THEN 160
140     PRINT "GOOD OLD "; A$
150     GOTO 110
160 'END LOOP
170 END

```


Note that the **INPUT** statement asks for input of a string value. The exit condition from the loop becomes true when **A\$** has the value **STOP**.

16. What would the output be if you ran the program and entered the word **CAR**?

---

Run the program and when the input prompt comes up on the screen, enter the word **CAR**. What was output?

---

17. The computer is still in the loop waiting for input. One at a time, enter words from the following list. After each word, don't forget to press .

**DOG, HORSE, TIMES**

There should be no surprises. What will happen if you enter the word **STOP**?

---

Enter the word and see if you were correct.

18. This last version of the loop is the so-called standard version. Inside the loop, something gets done before the computer comes to the exit condition. After the exit condition, something else gets done before the loop is closed. You will see many examples of this loop in the balance of the book.

### The FOR/NEXT Abbreviation

19. Use the **NEW** command to clear the program in memory. Clear the screen and enter the following program:

```

100 'PROGRAM COUNTING LOOP
110     FOR X = 1 TO 10
120     PRINT X
130     NEXT X
140 END

```

Study the program a bit and try to get a sense of what it will tell the computer to do. Now run the program. What happened?

---

20. Change line 110 to read

```
110    FOR X = 1 TO 20
```

What do you think will happen if the program is run now?

---

Run the program and see what did happen.

21. See what happens if you run the program after editing line 110 as follows:

```
110    FOR X = 10 TO 20
```

22. Now for something different. Edit line 110 as follows:

```
110    FOR X = 2 TO 20 STEP 2
```

From what you have already seen, what do you think the output will be if you run this version of the program?

---

Run the program and compare the output to your answer.

23. List the program and modify it until it looks like this:

```
100  'PROGRAM COUNTING LOOP
110    FOR X = 2 TO 20 STEP 2
120        PRINT X
130    NEXT X
135    PRINT "EXIT VALUE OF X IS "; X
140  END
```

In step 21, you ran a similar program and saw that the computer printed numbers from 2 to 20 by twos. The computer stayed in the loop until the last value was printed. As you saw, the last value of **X** was **20**. Line 135 is new. What do you think will be printed by the new line?

---

Run the program and record what did happen.

---

24. Edit line 110 as follows:

```
110    FOR X = 30 TO 0 STEP -5
```

What values will be printed out now? In particular, what do you think the exit value of **X** will be?

---

Run the program and see if you were right.

25. This concludes the Discovery Exercises. Turn off the computer and go on to the discussion that follows.

### 6—3 DISCUSSION

Before you examine the loop in detail, some general remarks about program blocks are in order. Only three kinds of program blocks are needed to solve *any* programming problem. These are the **action**, **loop**, and **branch** blocks. Each of these blocks is known as a **one-in, one-out** block. That means that the computer always enters the block at the top and always exits at the bottom.

You have already experimented with the loop block in the Discovery Exercises in this chapter. In the next chapter, you will learn about the branch block. That leaves only the action block. Actually, you have been using the action block for some time. An action block is simply a set of BASIC statements that the computer executes in line-number order. The computer performs the first statement in the block, then the second, the third, and so on all the way to the last statement in the block. All the statements in the block are performed. There are no exits from the block, save the one at the bottom.

The action block is simplicity itself and needs no further explanation. However, the loop and branch blocks require some thought and careful organization.

#### Creating an Infinite Loop Block

In the Discovery Exercises, you created an infinite loop by adding a **GOTO** statement that sent the computer back to an earlier statement in the program. When you glance at a program, you should be able to see immediately that a loop is present. Often, programmers create an infinite loop with just the **GOTO** statement. That practice is bad since if the body of the loop is several pages of code, it takes a lot of reading to discover that there is a loop in the program.

Problems like this can be avoided by using a standard form for all infinite loops. Here is how it looks:

```

XXX 'LOOP FOREVER  ←
    first body statement
    second body statement
    etc.
    GOTO XXX
'END LOOP

```



This is only an outline of the infinite loop. When you use one in a program, line numbers are needed. In the outline above, **XXX** stands for a line number (more about that later).

The first line in the outline tells the reader that an infinite-loop block is beginning. The body statements are all indented. The final remark statement lets the reader know where the loop ends. Even if the body of the infinite loop is several pages long, the remark statement at the beginning tells the reader that a loop is present.

The outline contains a new feature. It is the arrow from the **GOTO** statement back to the first remark statement. This is called a **jump arrow**. It reminds you of something important: **XXX**, which stands for the number after the word **GOTO** *must* be the same as the line number of the remark statement at the beginning. You will see jump arrows in the loop outlines in this chapter and in the branch outlines in the next chapter.

The line that the jump arrow points to is called the **target** of the **GOTO**. In this book, the targets will always be remark statements. The words after the apostrophes in the remark statements explain the reason for the jump. In the infinite-loop outline, the reason for the jump is to repeat the body of the loop forever. *Always use remark statements to explain the jumps in your programs.*

In a surprisingly large number of applications, an infinite loop is just what is needed. Examples might include a navigational program for a jet aircraft or a program to monitor and control an oil refinery. In cases like this, you want the program to keep running, carrying out whatever its purpose is, until a human operator intervenes. If the computer is to stop the loop, there must be some **condition** that tells when the loop is to be exited.

### Writing a Counting Loop

In the Discovery Exercises, you ran the following program:

```

100 'PROGRAM COUNTING LOOP
110   LET N = 1
120   'LOOP
125   IF N > 100 THEN 160
130     PRINT N
140     LET N = N + 1
150     GOTO 120
160   'END LOOP
170 END

```

A condition is stated in line 125. When this condition became true (i.e., when the value of **N** became greater than **100**), the computer left the loop. In this case the condition involves the **comparison operator** **>** which means *greater than*. The complete set of comparison operators is: **=**, **<**, **>**, **<>**, **<=**, and **>=**. Each of these comparison operators can be used in loops, the subject of this chapter, and in branches, the subject of the next. Each of the comparison operators has a specific meaning when used to compare numbers (and also strings).

For numbers, here are the meanings:

### Comparison Operator    Meaning When Applied to Numbers

=	is equal to
<>	is not equal to
>	is greater than
>=	is greater than or equal to
<=	is less than or equal to
<	is less than

The table below gives the meaning of the same six comparison operators when used to compare strings. The words “earlier” and “later” refer to dictionary order.

### Comparison Operator    Meaning When Applied to Strings

=	is equal to
<>	is not equal to
>	is later than
>=	is later than or equal to
<=	is earlier than or equal to
<	is earlier than

The condition in an **IF** statement can have more than one comparison. Such a condition is called a **compound** condition. The conditions are separated by the words **OR** or **AND**. The compound condition

*comparison 1 OR comparison 2*

is true if comparison 1 is true, or if comparison 2 is true, or if both comparisons are true. Otherwise, the compound condition is false. On the other hand, the compound condition

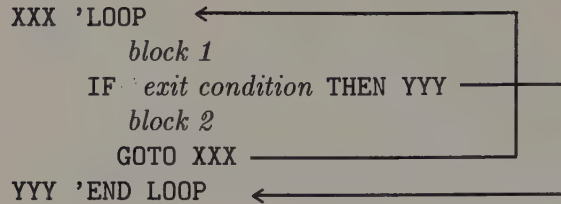
*comparison 1 AND comparison 2*

is true only if both comparisons are true and is false otherwise.

You can use compound conditions that have comparisons separated by both **OR** and **AND**. You can have more than two comparisons in the same compound condition. However, compound conditions like this can be hard to understand. Consult your *BASIC* reference manual for further information.

## Learning the Standard Loop Block

The infinite loop puts the computer into a loop that lasts forever, or at least until a human operator stops the program. But most often, the requirement is to keep performing the loop again and again until some condition becomes true. At that point, the computer should leave the loop automatically without any human action. We need a new structure for this kind of loop. Here it is:



The *exit condition* in the **IF** statement tells the computer when to stop looping and jump out of the loop. When the computer enters the loop, it performs the statements included in block 1. Then it tests the exit condition to see whether it is true or false. If it is true, the computer jumps to the **END LOOP** remark statement and leaves the loop. If the exit condition is false, the computer performs the statements included in block 2 and then goes back to the beginning of the loop.

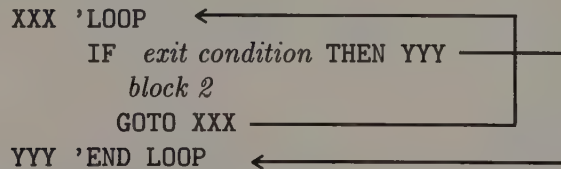
#### ■ Use **IF/THEN** to exit a standard loop.

The loop works this way only if it is written correctly. There is nothing magical about the phrases **LOOP** and **END LOOP** in the remark statements. They are there to help the reader. The computer ignores them. But the numbers after **THEN** and **GOTO** are important: *They must match the line numbers of the two remark statements in the outline.* If they are wrong, the computer will not perform the loop correctly. It is up to you to get the numbers right.

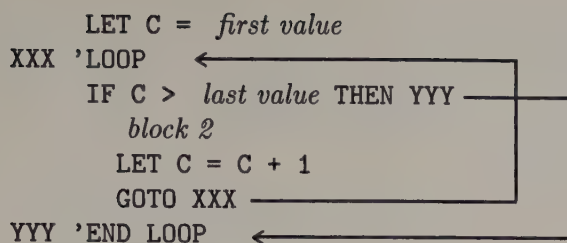
There is one important difference between the two subblocks in the loop-block outline. The computer must perform the first block at least once. However, it may never get to the second one at all. This happens if the exit condition is true at the start.

In some loop blocks, one or the other of the blocks inside may not be there at all. In other words, the **IF** statement may be the first statement inside the loop. Or, it may be the last, except for the **GOTO** statement. You will see some examples of this later.

If one of the blocks is missing, the loop block is called an **abbreviated loop block**. Here is a loop block with the first interior block missing:



A counting loop is an example of an abbreviated loop block. The purpose of a counting loop is to perform some action (to “do something” as described in the interior program block) a fixed number of times. The counting loop needs one statement before the loop block. That statement sets the counter to its first value. Here is the outline of any counting loop:



Let's see how to use the counting loop to tell the computer to print the numbers from 50 to 100. In the outline, the words *first value* stand for 50. The words *last value* stand for 100. The words *block 2* stand for **PRINT C**. Here is a program segment for this example:

```

210 LET C = 50
220 'LOOP
230 IF C > 100 THEN 270
240 PRINT C
250 LET C = C + 1
260 GOTO 220
270 'END LOOP
  
```

Line numbers have been added. The jump arrows are present only in the outline. They are not in the BASIC programs.

The first line in the program says to assign **50** to **C**. Inside the loop, the **IF** statement tests **C** to see if it is already greater than **100**. It is not, so the computer performs the **PRINT** statement. Thus, **50** appears on the screen. Next, the **LET** statement first adds **1** to **C** and then assigns this value, **51**, as the new value of **C**. Finally, the **GOTO** statement sends the line pointer back to the beginning of the loop. Each time the computer goes through the loop, it tests to see if **C** has become greater than **100**. Eventually, the condition becomes true. As a result, the loop stops. Until that happens, the **PRINT** statement prints each number through **100**.

### Using the FOR/NEXT Abbreviation

Here is a program that asks you to enter a first value and a last value. It then uses a counting loop to print a list of numbers that starts with the first value and ends with the last value.

```

100 'PROGRAM COUNT
110 PRINT "FIRST AND LAST VALUES"
120 INPUT F, L
130 LET C = F
140 'LOOP
150 IF C > L THEN 190
160 PRINT C
170 LET C = C + 1
180 GOTO 140
190 'END LOOP
200 END
  
```



There is a simpler way to write the same program. Here it is:

```

100 'PROGRAM COUNT
110     PRINT "FIRST AND LAST VALUES"
120     INPUT F, L
140     FOR C = F TO L
160         PRINT C
190     NEXT C
200 END

```

The new **FOR** statement is a shorthand way of writing lines 130, 140, and 150 in the original version of program **COUNT**. The new **NEXT** statement replaces lines 170, 180, and 190 in the original version. Notice also that the variables **F** and **L** appear in the **FOR** statement.

In the example above **C** is known as a **loop variable**. For this case, the loop variable begins at the value **F** and is increased by 1 each time around the loop. If there is a **STEP** clause, the value of the loop variable is increased by the number after **STEP** each time around the loop. For example, the statement

```
140     FOR C = F TO L STEP 2
```

would cause the loop variable **C** to go from **F** to **L** in steps of 2. The loop variable keeps increasing until its value is greater than **L**. At that point, the computer leaves the loop and goes on to the statement after the **NEXT** statement.

As you saw in the Discovery Exercises, the **FOR/NEXT** abbreviation can also decrease the value of the loop variable each time around the loop. If the value of **F** is greater than the value of **C**, then the statement

```
140     FOR C = F TO L STEP -1
```

would cause the computer to count down from **F**, decreasing the value of **C** by 1 each time around the loop, until the value of **C** is less than **L**. At that point, the computer would leave the loop.

A word of caution is in order with regard to the **FOR/NEXT** abbreviation of the loop block. Anything that can be done with the **FOR/NEXT** abbreviation can be done with the standard loop block. However, you can do things with the standard loop block that you cannot do with the **FOR/NEXT** abbreviation. It is a mistake to attempt to shoehorn all loop requirements into the **FOR/NEXT** abbreviation. Use the abbreviation when convenient, but learn to rely on the standard form of the loop block, which will take care of *all* loop needs.

## 6—4 WORKING WITH PROGRAM EXAMPLES

Let's go through several examples in detail to illustrate how you can use different forms of loops in programs. While learning about applications of loops, you will also learn more about how to write programs using a top-down approach.

### Example 1 - Finding the Average of a Group of Numbers

Your problem is to write a program to compute the average of a set of numbers. When the program is run, you will enter the numbers at the keyboard, one at a



time. When the last number in the list to be averaged has been entered, you tell the computer you are finished by entering the **flag variable -999**. Then the computer should compute the average and print out the results.

A paper-and-pencil outline of the tasks to be done might look like this:

```

PROGRAM AVERAGE
  Initialize for computation
  Get data from keyboard
  Compute average and print results
END

```

Next, add line numbers and convert the phases to remark statements.

```

100 'PROGRAM AVERAGE
110 'Initialize for computation
120 'Get data from keyboard
130 'Compute average and print results
140 END

```

Now, look at each of the English phrases. The rule is that if you can perform whatever is called for with a few BASIC statements, do so. If not, or if you suspect complications may arise, bury the details in a subroutine. For the three phrases above, it seems appropriate to replace each with a **GOSUB** statement to a subroutine where the details will be located.

```

100 'PROGRAM AVERAGE
110   GOSUB 200 'Initialization
120   GOSUB 400 'Get data
130   GOSUB 600 'Results
140 END

```

Note that trailing remarks have been put after each **GOSUB** to explain the purpose of the subroutine. Now the main routine is finished, and you can turn to the subroutines.

As before, the best way to begin is with an skeleton of each subroutine, describing the specific tasks to be performed. It is wise also to put a dummy **PRINT** statement to identify each subroutine. Later, this **PRINT** statement will be removed. For subroutine **INITIALIZATION**, here is how such a skeleton might look:

```

200 'SUB INITIALIZATION
205 ' PRINT "INITIALIZATION"
210 'Clear screen
220 'Set variables to zero
230 'Print instructions
380 RETURN

```

The skeleton for subroutine **GET DATA** is easy to write.

```

400 'SUB GET DATA
405   PRINT "GET DATA"
410   'Loop and input data until flag variable is input
580   RETURN

```

Finally, subroutine **RESULTS** must perform two tasks.

```

600 'SUB RESULTS
605   PRINT "RESULTS"
610   'Compute average
620   'Print results
780   RETURN

```

Here is what the program looks like at this point:

```

100 'PROGRAM AVERAGE
110   GOSUB 200 'Initialization
120   GOSUB 400 'Get data
130   GOSUB 600 'Results
140   END
190 '
200 'SUB INITIALIZATION
205   PRINT "INITIALIZATION"
210   'Clear screen
220   'Set variables to zero
230   'Print instructions
380   RETURN
390 '
400 'SUB GET DATA
405   PRINT "GET DATA"
410   'Loop and input data until flag variable is input
580   RETURN
590 '
600 'SUB RESULTS
605   PRINT "RESULTS"
610   'Compute average
620   'Print results
780   RETURN

```

Lines 190, 390, and 590 are blank remark statements to separate the blocks of the program. Notice the line numbers assigned to the **RETURN** statements. It's good practice to leave room for many intervening statements between the first statement of the subroutine and **RETURN**. When you replace the English phrases in each subroutine, the line numbers of the body statements may change.

Enter this version of the program and run it now. The result should be the following three phrases:

```

INITIALIZATION
GET DATA
RESULTS

```

These phrases tell you that the computer is performing the subroutines in the correct order. In this very simple program, that doesn't help much. However, in more complicated programs, it is sometimes crucial to be sure that the computer is going through the subroutines in the correct sequence.

The next task is to replace the English statements in the body of each subroutine. As before, if the task can be accomplished with a few BASIC statements, write them. If not, use a **GOSUB** to a new subroutine, which will contain the new details.

Here is the first subroutine:

```
200 'SUB INITIALIZATION
205   PRINT "INITIALIZATION"
210   'Clear screen
220   'Set variables to zero
230   'Print instructions
380   RETURN
```

The phrase *Clear screen* can be replaced with the **CLS** statement. The phrase *Set variables to zero* needs a little thought. When computing the average, you will need two variables. Use **SUM** to stand for the sum of the numbers to be averaged and **N** for the number of numbers to be averaged. Initially, both of these must be set to zero. Finally, several **PRINT** statements can be used to give the instructions referred to in the phrase *Print instructions*.

After these details are supplied, subroutine **INITIALIZATION** looks like this:

```
200 'SUB INITIALIZATION
210   CLS
220   LET SUM = 0
230   LET N = 0
240   PRINT "ENTER NUMBERS ONE AT A TIME"
250   PRINT "AFTER LAST NUMBER, ENTER"
260   PRINT "FLAG VALUE -999"
270   RETURN
```

The dummy **PRINT** statement has been removed. The lines have also been renumbered.

Next, finish subroutine **GET DATA**. Here is how it looks now:

```
400 'SUB GET DATA
405   PRINT "GET DATA"
410   'Loop and input data until flag variable is input
580   RETURN
```

As before, the dummy **PRINT** statement will be removed when the body of the subroutine is finished. What kind of loop is needed here? The **FOR/NEXT** abbreviation won't work, since you don't know in advance how many numbers are to be averaged. The standard form of the loop will always work. Here is the subroutine with a loop outline in the body. The loop outline contains English descriptions of what must be done.

```

400 'SUB GET DATA
410 'LOOP
420 'Get a number from the keyboard
430 IF the number is the flag THEN exit the loop
440 'Add the number to SUM
450 'Add 1 to N
460 GOTO 410
470 'END LOOP
480 RETURN

```

It should be easy for you to translate the remaining English phrases into BASIC. Here is the result:

```

400 'SUB GET DATA
410 'LOOP
420 INPUT X
430 IF X = -999 THEN 470
440 LET SUM = SUM + X
450 LET N = N + 1
460 GOTO 410
470 'END LOOP
480 RETURN

```

Only subroutine RESULTS is left. Here is how it looks now:

```

600 'SUB RESULTS
605 PRINT "RESULTS"
610 'Compute average
620 'Print results
780 RETURN

```

The average is computed by dividing SUM by N. The results can be printed out with several PRINT statements. If the dummy PRINT statement in line 605 is removed, and the remaining details are inserted, the completed subroutine looks like this:

```

600 'SUB RESULTS
610 LET AVE = SUM / N
620 PRINT
630 PRINT N; " NUMBERS WERE INPUT"
640 PRINT "THEIR AVERAGE IS "; AVE
650 RETURN

```

Here is the completed program:

```

100 'PROGRAM AVERAGE
110 GOSUB 200 'Initialization
120 GOSUB 400 'Get data
130 GOSUB 600 'Results
140 END
190 '
200 'SUB INITIALIZATION

```



```

210    CLS
220    LET SUM = 0
230    LET N = 0
240    PRINT "ENTER NUMBERS ONE AT A TIME"
250    PRINT "AFTER LAST NUMBER, ENTER"
260    PRINT "FLAG VALUE -999"
270    RETURN
390 '
400 'SUB GET DATA
410 'LOOP
420     INPUT X
430     IF X = -999 THEN 470
440     LET SUM = SUM + X
450     LET N = N + 1
460     GOTO 410
470 'END LOOP
480    RETURN
590 '
600 'SUB RESULTS
610     LET AVE = SUM / N
620     PRINT
630     PRINT N; " NUMBERS WERE INPUT"
640     PRINT "THEIR AVERAGE IS "; AVE
650    RETURN

```

Run this final version of the program with several lists of numbers. Verify that it does work as intended. (Incidentally, there is a bug in the program. Can you find it?)

At this point, you may still have the feeling that you have been using cannons to go mouse hunting, that very heavy-duty tools have been used to solve a simple problem. It is quite true that you can write a much shorter program to average a list of numbers. However, these rough-and-ready solutions often fail when applied to more complicated problems.

### Example 2 - Depreciation Schedule

When a company invests in equipment, the investment is depreciated over a number of years for tax purposes. This means that the value of the equipment is decreased each year (due to use, wear, and tear), and the amount of decrease is a tax-deductible item. One of the methods used to compute depreciation is the "sum-of-the-years-digits" schedule. To illustrate, suppose a piece of equipment has a lifetime of 5 years. The sum of the years' digits would be

$$1 + 2 + 3 + 4 + 5 = 15$$

The depreciation the first year will be 5/15 of the initial value. The depreciation fraction the second year will be 4/15, and so on. If the equipment had an initial value of \$3000, the depreciation schedule would be:



End of Year	Depreciation Fraction	Current Depreciation	Asset Value
1	5/15	1000	2000
2	4/15	800	1200
3	3/15	600	600
4	2/15	400	200
5	1/15	200	0

The problem is to write a program to ask for the necessary data from the keyboard and then print out a depreciation table like the one above.

To solve this problem, the program must perform three tasks: 1) getting data about the value of the equipment and the expected lifetime, 2) computing the sum-of-the-digits, and 3) computing and printing out the depreciation table. Here is a first pass at the main routine with line numbers included:

```

100 'PROGRAM DEPRECIATION
110 'Get data about depreciation
120 'Compute sum of the years
130 'Compute and print schedule
140 END

```

Each of the three tasks will require more than one or two BASIC statements, so each phrase will be replaced by a GOSUB statement.

```

100 'PROGRAM DEPRECIATION
110 GOSUB 200 'Get data
120 GOSUB 400 'Sum of years
130 GOSUB 600 'Print schedule
140 END

```

That completes the main routine. Next, let's write a skeleton outline of each of the three subroutines. As in Example 1, you'll include dummy PRINT statements in each subroutine.

```

190 '
200 'SUB GET DATA
205 PRINT "GET DATA"
210 'Get asset value
220 'Get asset life
380 RETURN
390 '
400 'SUB SUM OF YEARS
405 PRINT "SUM OF YEARS"
410 'Compute sum of the years
580 RETURN
590 '
600 'SUB PRINT SCHEDULE
605 PRINT "PRINT SCHEDULE"
610 'Print table heading
620 'Compute and print current asset values
780 RETURN

```

Enter the main routine and the skeleton outline of the subroutines into the computer. Run the program and check that the subroutines are being performed in the right order.

Now, turn to the body of the first subroutine. First, remove the dummy **PRINT** statement. Then replace the English phrases with the necessary BASIC statements. The details follow quite easily and are shown below:

```

200 'SUB GET DATA
210   PRINT "WHAT IS THE INITIAL ASSET VALUE";
220   INPUT VALUE
230   PRINT "WHAT IS THE ASSET LIFE IN YEARS";
240   INPUT LIFE
250   RETURN

```

A **FOR/NEXT** loop is ideal to compute the sum of years in the second subroutine. Here is subroutine **SUM OF YEARS** with the body completed:

```

400 'SUB SUM OF YEARS
410   LET SUM = 0
420   FOR J = 1 TO LIFE
430     LET SUM = SUM + J
440   NEXT J
450   RETURN

```

Printing the schedule heading in subroutine **PRINT SCHEDULE** can be done with several **PRINT** statements. Computing and printing the table is an ideal application of a **FOR/NEXT** loop. The actual computation may be involved, so you'll put it in another subroutine. Notice the continuation of line 640.

```

600 'SUB PRINT SCHEDULE
610   PRINT
620   PRINT "END OF"; TAB(10); "DEPRECIATION";
630   PRINT TAB(24); "DEPRECIATION"; TAB(38); "CURRENT"
640   PRINT "YEAR"; TAB(10); "FRACTION"; TAB(38);
      "ASSET VALUE"
650   PRINT
660   LET CURRENT = VALUE
670   FOR J = 1 TO LIFE
680     GOSUB 800 'Compute data and print results
690   NEXT J
700   RETURN

```

This is an example of discovering the need for a new subroutine while completing the body of another. If things look like they are going to get complicated, always bury the details in another subroutine.

The last task is to write the subroutine beginning in line 800. The new subroutine will compute the actual depreciation and print it on the screen. The subroutine is shown to you without explanation. Go through each statement and compare the computations with the table at the beginning of this example. Convince yourself that the computations do compute the desired depreciation values.

```

800 'SUB COMPUTE DATA
810   LET FRACTION = (LIFE - J + 1) / SUM
820   LET DEPRECIATION = VALUE * FRACTION
830   LET CURRENT = CURRENT - DEPRECIATION
840   PRINT J; TAB(10); FRACTION;
850   PRINT TAB(24); DEPRECIATION; TAB(38); CURRENT
860   RETURN

```

Here is the complete program:

```

100 'PROGRAM DEPRECIATION
110   GOSUB 200 'Get data
120   GOSUB 400 'Sum of years
130   GOSUB 600 'Print schedule
140   END
190
200 'SUB GET DATA
210   PRINT "WHAT IS THE INITIAL ASSET VALUE";
220   INPUT VALUE
230   PRINT "WHAT IS THE ASSET LIFE IN YEARS";
240   INPUT LIFE
250   RETURN
390 '
400 'SUB SUM OF YEARS
410   LET SUM = 0
420   FOR J = 1 TO LIFE
430     LET SUM = SUM + J
440   NEXT J
450   RETURN
590 '
600 'SUB PRINT SCHEDULE
610   PRINT
620   PRINT "END OF"; TAB(10); "DEPRECIATION";
630   PRINT TAB(24); "DEPRECIATION"; TAB(38); "CURRENT"
640   PRINT "YEAR"; TAB(10); "FRACTION"; TAB(38);
      "ASSET VALUE"
650   PRINT
660   LET CURRENT = VALUE
670   FOR J = 1 TO LIFE
680     GOSUB 800 'Compute data and print results
690   NEXT J
700   RETURN
790 '
800 'SUB COMPUTE DATA
810   LET FRACTION = (LIFE - J + 1) / SUM
820   LET DEPRECIATION = VALUE * FRACTION
830   LET CURRENT = CURRENT - DEPRECIATION
840   PRINT J; TAB(10); FRACTION;
850   PRINT TAB(24); DEPRECIATION; TAB(38); CURRENT
860   RETURN

```

If you are using a 40-column display, the output will be folded. You can, however, use a different format so that the table fits the 40-character width. Enter the bodies of the subroutines. Check out the program with several sets of data. Since you have not rounded dollar amounts off to the nearest penny, the current value at the end of the last year may not be exactly zero. (In Chapter 8, you will learn how to use functions to take care of this problem.)

### Example 3 - Table of Numbers and Their Cubes

Your final problem is to compute and print a table of numbers and their cubes. In particular, the table should begin with 1 and continue in steps of 1 as long as the cube of the number is less than 5000. Here are the first few entries in the table:

Number	Cube of Number
1	1
2	8
3	27
4	64

What kind of structure is needed to solve this problem? At first glance, a **FOR/NEXT** loop seems ideal. The problem is that you don't know at the outset what the last number in the list is (the one whose cube is closest to but does not exceed 5000). However, the standard form of the loop will work.

This program is quite simple, so let's put it in the main routine and not rely on subroutines. Here is an outline of the main routine:

```

100 'PROGRAM CUBES
110 'Print table heading
120 'Set value of X to 1
130 'LOOP
140 'compute cube of X
150 IF cube > 5000 THEN 190
160 'Print value of X and cube
170 'Add 1 to X
180 GOTO 130
190 'END LOOP
200 END

```

With this outline, it is easy to fill in the necessary BASIC statements.

```

100 'PROGRAM CUBES
110 PRINT "NUMBER", "CUBE OF NUMBER"
115 PRINT
120 LET X = 1
130 'LOOP
140 LET CUBE = X * X * X
150 IF CUBE > 5000 THEN 190
160 PRINT X, CUBE
170 LET X = X + 1
180 GOTO 130
190 'END LOOP
200 END

```



Enter the program and run it. See what the last number is whose cube is less than 5000.

In the discussion section, you learned that anything you can do with a **FOR/NEXT** loop can be done with the standard form of the loop block, but not the other way around. This problem is a good example. The **FOR/NEXT** abbreviation of the standard loop block is very useful. However, remember that it is just an abbreviation of a more general structure.

## 6—5 PROBLEMS

1. You can convert from degrees Fahrenheit to degrees Celsius with the following equation:

$$C = (5/9) * (F - 32)$$

Write a program to generate a conversion table. Begin with 0 degrees **F** and go up to 100 degrees **F** in steps of 5 degrees.

2. Write a program to count from 0 to 500 by tens and print out the results.
3. What will be printed out if the following program is run:

```

100 'PROGRAM PROBLEM 3
110   LET N = 50
120   'LOOP
130   PRINT N
140   IF N < 10 THEN 170
150   LET N = N - 5
160   GOTO 120
170   'END LOOP
180   END

```

4. Write a program to accept the input of a number **N**. Then print out the even numbers greater than 0 but less than or equal to **N**.
5. **N!** is read “**N** factorial” and means the product of all the counting numbers from 1 to **N** inclusive. For example

$$\begin{aligned}
 3! &= (1)(2)(3) = 6 \\
 5! &= (1)(2)(3)(4)(5) = 120
 \end{aligned}$$

and so on. Write a program to call for the input of **N**. Then compute and print out **N!** If you try out this program on the computer, you may be surprised to find that there are values of **N** less than 100 that produce factorials too large for the computer to handle. The factorial of **N** is an *extremely* rapidly increasing function.

6. Look at the following sequence of numbers:

1, 1, 2, 3, 5, 8, 13, 21, and so on.

The sequence starts with the two numbers 1 and 1. Then each new number is the sum of the two previous ones. These are called Fibonacci numbers. Write a program



to compute and print out all Fibonacci numbers in this sequence that are less than 1000.

7. Write a program to print out a conversion table from inches to centimeters. There are 2.54 centimeters in 1 inch. Include the appropriate headings. Start the table at 0 and continue to 10 inches in steps of 0.5 inches.
8. Suppose you decide to invest \$1000 on the first of each year for 10 years at an annual interest rate of 6 percent. At the end of the tenth year, the value of the investment will be \$13,971.64. To see how this could be computed, use the following formula:

$$\text{NewDollars} = (\text{OldDollars} + \text{Investment})(1 + \text{Rate} / 100)$$

In this formula, *Rate* is the annual interest rate expressed as a percentage, *Investment* is the annual investment, *OldDollars* is the value of the investment at the beginning of each year, and *NewDollars* is the value of the investment at the end of the year. Thus, *NewDollars* becomes *OldDollars* for the next year. Write a BASIC program that will produce the following output:

```
WHAT IS THE ANNUAL INVESTMENT? (You enter)
WHAT IS THE ANNUAL INTEREST RATE (%)? (You enter)
HOW MANY YEARS? (You enter)
AT THE END OF THE LAST YEAR, THE VALUE OF THE
INVESTMENT WILL BE (Computer prints answer)
```

## 6—6 PRACTICE TEST

1. If the computer is put in an infinite loop, how can you stop it?  
\_\_\_\_\_
2. Write the outline of the standard loop block.  
\_\_\_\_\_
3. How do you begin a **FOR/NEXT** abbreviation for a standard loop block?  
\_\_\_\_\_
4. If no **STEP** size is specified in a **FOR/NEXT** abbreviation, what interval is used for the loop variable?  
\_\_\_\_\_
5. Explain why some problems can be solved with the standard loop block but not the **FOR/NEXT** abbreviation.  
\_\_\_\_\_
6. Miles can be converted to kilometers by multiplying the number of miles by 1.609. Write a program to produce the following output when run.

Miles	Kilometers
-----	-----
10	16.09
15	24.135
20	32.18
etc.	
100	160.9

- 
7. What is wrong with the following program?

```

100 'PROGRAM PRACTICE TEST 7
110   LET Y = 20
120   'LOOP
130   IF Y = 30 THEN 160
140   LET Y = Y + 2
150   PRINT Y
160   'END LOOP
170 END

```

- 
8. What will be printed if the following program is run?

```

100 'PROGRAM PRACTICE TEST 8
110   LET X = 1
120   'LOOP
130   IF X = 4 THEN 170
140   GOSUB 200 'Y loop
150   LET X = X + 1
160   GOTO 120
170   'END LOOP
180 END
190 '
200 'SUB Y LOOP
210   LET Y = 1
220   'LOOP
230   IF Y = 4 THEN 270
240   PRINT X * Y
250   LET Y = Y + 1
260   GOTO 220
270   'END LOOP
280 RETURN

```

- 
9. If A\$ = "KITTY" and B\$ = "KITTYCAT", then A\$ > B\$. True or false?

# THE BRANCH BLOCK

## 7—1 OBJECTIVES

### Learning the IF/THEN/ELSE Statement

The **IF/THEN/ELSE** statement is necessary to tell the computer to choose between two options. You will use an **IF/THEN/ELSE** statement in a BASIC program to let the computer make a choice.

### Learning the Standard Branch Block

The standard branch block, along with the action block and the loop block, is one of the three blocks needed to solve any programming problem. You will learn the standard form for this block and apply it in programs.

### Learning an Abbreviated Branch Block

Sometimes, you want the computer to do something if a condition is true; otherwise you want it to do nothing special. You need an abbreviated branch block, a shortened version of the standard branch block, to handle this situation. You will write programs using the abbreviated branch block.

### Learning a Case Block

The case block instructs the computer to choose from among a number of options. You will learn the standard form for the case block and apply it to programming problems.

### Working with Programming Examples

You will apply the ideas you learn in programming problems.

## 7—2 DISCOVERY EXERCISES

### The IF/THEN/ELSE Statement

1. You have already used the IF/THEN statement to get out of a loop. You need to look at this statement in more detail. Turn on the computer, bring up BASICA, and enter the following program:

```

100 'PROGRAM DEMO
110   INPUT A$
120   IF A$ = "TRUE" THEN 150
130   PRINT "A"
140   GOTO 180
150   PRINT "B"
180   END

```

2. Read the program carefully. Depending on what you enter in response to the prompt generated by line 110, the program prints either **A** or **B** on the screen. What do you think will happen if you run the program and enter **TRUE**?

---

Run the program, enter **TRUE**, and record what was printed.

---

3. What will happen if you enter **FALSE**?

---

Try it and see if you were correct.

4. Change line 120 so that it looks like this:

```

120   IF A$ = "TRUE" THEN 150 ELSE 130

```

Run the program, enter **TRUE**, and record what happened.

---

Run the program, enter **FALSE**, and record what happened.

---

Apparently, the **ELSE** clause at the end of line 120 made no difference in the output of the program.

5. Change line 120 so that it looks like this:

```
120    IF A$ = "TRUE" THEN 130 ELSE 150
```

Display the program. Run the program, enter **TRUE**, and record what happened.

---

Run the program, enter **FALSE**, and record what happened.

---

Compare the results with those in step 4.

6. Add line 115 and change line 120 as follows:

```
115    INPUT B$
120    IF A$ = "TRUE" AND B$ = "TRUE" THEN 130 ELSE 150
```

Here is how the complete program should look:

```
100 'PROGRAM DEMO
110    INPUT A$
115    INPUT B$
120    IF A$ = "TRUE" AND B$ = "TRUE" THEN 130 ELSE 150
130    PRINT "A"
140    GOTO 180
150    PRINT "B"
180    END
```

Now the program asks for two inputs. The **IF** statement in line 120 tests both the value of **A\$** and **B\$**. The word **AND** separates the two conditions tested.

7. What do you think will be happen if you run the program and enter **TRUE** for both **A\$** and **B\$**?
- 

See if you are right.

8. Suppose you enter **TRUE** for **A\$** and **FALSE** for **B\$**. What would be printed?
- 

What about **FALSE** for **A\$** and **TRUE** for **B\$**?

---

Check out both cases and see what happens.

9. There is only one possibility left. What will happen if you enter **FALSE** for both **A\$** and **B\$**.
-



See if your answer is correct.

10. Now change the **AND** in line 120 to **OR**. Repeat the experiments you carried out in steps 7, 8, and 9. Think about the results you observe until you can explain what is happening.

### The Standard Branch Block

11. Clear out the program in memory. Enter the following program:

```

100 'PROGRAM BRANCH
110   INPUT A$
120   IF A$ = "TRUE" THEN 130 ELSE 160
130   'THEN CLAUSE
140       PRINT "TRUE WAS INPUT"
150       GOTO 180
160   'ELSE CLAUSE
170       PRINT "TRUE WAS NOT INPUT"
180   'END IF
190 END

```

Take a few minutes to study the program. Depending on the value you input for **A\$** in line 110, what do you think will happen?

---

12. Run the program. At the input prompt, enter **TRUE**. What happened?
- 

13. Run the program again. This time, enter **ABC**. What happened?
- 

All **branch blocks** can be put into this form. Either one thing or another gets done depending upon the condition in the **IF** statement.

14. The **INPUT** statement in line 110 can be modified to give clear directions on what input is expected. Change line 110 as follows:

```

110   INPUT "ENTER A TRUE OR FALSE "; A$

```

Run the program. Enter one of the words requested. What happened this time?

---

This same effect can be achieved with a **PRINT** and **INPUT** statement as you know. This form of the **INPUT** statement makes the program easier to follow.

### An Abbreviated Branch Block

15. Suppose that you want something to happen if a condition is true and, otherwise, nothing in particular to happen. An **abbreviated branch block** handles this situation. Delete lines 150 through 170 and display the program. It should look like this:

```

100 'PROGRAM BRANCH
110   INPUT A$
120   IF A$ = "TRUE" THEN 130 ELSE 180
130     'THEN CLAUSE
140       PRINT "TRUE WAS INPUT"
180   'END IF
190   END

```

Line numbers 150, 160, and 170 (the **ELSE** clause) are missing. Also, the last line number in line 120 has been changed. What will happen if you run the program and enter **TRUE**?

---

What will happen if you run the program and enter **GOOD**?

---

Try it and see if you were correct.

---

What will happen if you run the program and enter anything but **TRUE**?

---

Run the program twice and find out if you were correct.

16. In cases like this, when the **THEN** block in an abbreviated branch block is a single statement, you can abbreviate the block further. Change the program presently in memory to look like this:

```

100 'PROGRAM BRANCH
110   INPUT A$
120   IF A$ = "TRUE" THEN PRINT "TRUE WAS INPUT"
190   END

```

Statements 130 through 180 have been deleted and the **IF** statement in line 120 has been modified. Again, what do you think will happen if you run the program and enter **TRUE**?

---

What about any other entry?

---

Run the program twice and see if you were correct.

## A Case Block

17. The standard branch block is just what you need if you want the computer to choose between two possibilities. But what if the computer is to choose from many possibilities? Although you can deal with this situation by nesting standard branch blocks inside one another, a new structure, the **case block** makes life easier. Clear out the program presently in the computer and enter the following program:

```

100 'PROGRAM CASE DEMO
110   INPUT N
120   'CASE
130     IF N = 1 THEN PRINT "ONE"
140     IF N = 2 THEN PRINT "TWO"
150     IF N = 3 THEN PRINT "THREE"
160     IF N = 4 THEN PRINT "FOUR"
170   'END CASE
180 END

```

This **case block** structure takes advantage of the one-line form of the branch block you saw in step 15. In this instance, there are several of these one-line branches, one after the other.

What do you think will happen if you run the program and enter 3 at the input prompt?

---

What about 4?

---

What about any number other than 1, 2, 3, or 4?

---

Take a few minutes to check your answers.

18. That concludes the computer work for now. Turn off the computer and go on to the next section.

## 7—3 DISCUSSION

In the previous chapter, you learned about action blocks and loop blocks. These, together with the branch block, are the three fundamental blocks needed to solve any programming problem. The action block is merely a series of action statements performed one after the other. The loop and branch blocks are the ones that need special attention.

Recall from Chapter 6 that the loop block is used whenever something is to be done again and again until some specified exit condition becomes true. The branch block, by contrast, is used when the computer must choose between two possibilities.

## Learning the IF/THEN/ELSE Statement

Often, there is some confusion about the role of the **IF** statement, which appears in both the loop and branch blocks. In the loop block, the exit condition has the following form:

```
IF  exit condition THEN XXX
```

The only purpose of the **IF** statement in a loop block is to provide a way out of the loop. Each time around the loop, the computer checks whether the exit condition is true. If it is not, the computer stays in the loop. If it is true, the computer exits the loop and jumps to line number **XXX**.

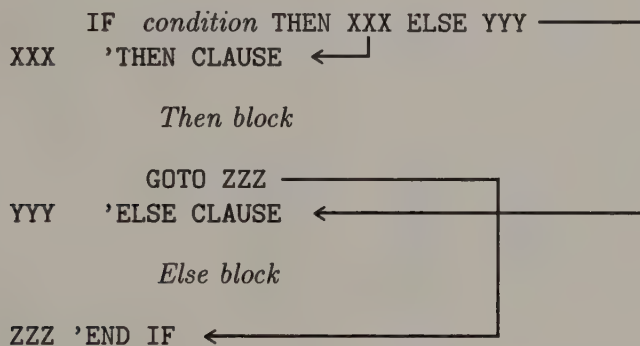
The **IF** statement in the branch block plays a fundamentally different role and looks like this:

```
IF  condition THEN XXX ELSE YYY
```

This **IF** statement has a new word, **ELSE**, at the end. If the condition is true, the computer jumps to the line number after **THEN**. If the condition is false, the computer jumps to the line number after **ELSE**. Thus, the purpose of the **IF/THEN/ELSE** statement in a branch block is to choose between two actions.

## Learning the Standard Branch Block

Here is the structure of the standard branch block:



**XXX**, **YYY**, and **ZZZ** stand for line numbers. When you write a program containing a branch block, it is up to you to see that the proper line numbers are used in place of **XXX**, **YYY**, and **ZZZ**. The line number following **THEN** must be the same as the statement containing the remark **THEN CLAUSE**. This line number is indicated by **XXX**. The line number following **ELSE** must be the same as the statement containing the remark **ELSE CLAUSE**. This line number is indicated by **YYY**. The remarks as well as the jump arrows remind you of this.

Notice that the remark **THEN CLAUSE** is followed by a block of statements called the *Then block*. Also, the remark **ELSE CLAUSE** is followed by a block of statements called the *Else block*. These blocks can be any of the three fundamental blocks; action, loop, or branch. It is important to think of which block is needed, not which statement is needed.



### Learning an Abbreviated Branch Block

Quite often, you want the computer to do something if a condition is true and otherwise do nothing in particular. This is equivalent to removing the **ELSE CLAUSE** from the standard branch block. The result, an abbreviated branch block, is shown below:

```

      IF condition THEN XXX ELSE YYY
XXX  'THEN CLAUSE ←
      Then block
      YYY 'END IF ←

```

As you can see, the **ELSE CLAUSE** remark and the following *Else block* are missing. Also, the line number following **ELSE** must be the same as the statement containing the remark **END IF**.

If the condition in the **IF** statement is true, the *Then block* is performed. If the condition is false, nothing at all happens.

### Learning a Case Block

The standard branch block works well if the computer must choose between two actions to perform. Sometimes, however, you want the computer to choose among many different things. You can handle this situation by nesting standard branch blocks inside one another. However, this often results in programs that are difficult to read and understand. A better solution, if each action can be done with a single statement, is the **case block**. The outline of the case block is shown below:

```

CASE
  IF condition 1 THEN statement 1
  IF condition 2 THEN statement 2
  IF condition 3 THEN statement 3
  IF condition 4 THEN statement 4
  IF condition 5 THEN statement 5
  etc.
END CASE

```

This isn't a strictly standard case block. In a standard case block, the statement after just one of the conditions is executed. In the form above, *all* the statements would be performed if all the conditions were true. The point is that for this structure to act like a standard case block, only one of the conditions should be true each time the case block is executed. It is up to you to make sure that only one condition is true.

If only one condition is true, then this structure allows the computer to perform just one of the statements in the case block, depending on the conditions in the block.

No matter what kinds of program blocks are needed, there is a well-defined way to write programs. Always begin with a top-level description of the main routine. This leads immediately to a program skeleton with the subroutines blocked



in using dummy **PRINT** statements. At this point, you can run the program and see if the messages in the dummy **PRINT** statements are displayed in the right order. Then you complete each of the subroutines in stages. First describe what tasks are to be performed and which program block is needed to accomplish each task. Then write an outline of each program block. Finally, insert line numbers and the details needed for each block.

In the next two example programs, you will follow each step in the process described above. Pay particular attention to the way the program evolves in very simple steps from the initial English-language description of the problem to the finished BASIC program.

## 7—4 WORKING WITH PROGRAM EXAMPLES

The solution to any programming problem can be written using only action, loop, and branch blocks. Of course, knowing that it is possible and doing so are two different things. As always, it is best to study examples in detail to see how to proceed from a written description of the problem to the final solution.

### Example 1 - Automobile License Fees

Let's assume that in an attempt to force consumers to use lower-horsepower cars and conserve energy, the state adopts a set of progressive annual license fees based on the power rating of the car. The criteria and fees are listed below:

Horsepower	License Fee
Up to 50 hp	0
More than 50 but 100 hp or less	30
More than 100 but 200 hp or less	70
More than 200 but 300 hp or less	150
More than 300 hp	500

The goal is to write a program that will produce the following output when executed:

```
INPUT AUTO HP?  (You type in horsepower)
LICENSE FEE IS  (Computer types fee) DOLLARS
```

### Main Routine

The first task is to make a pencil-and-paper outline of the main routine. This problem is simple, and the outline follows easily:

```
PROGRAM LICENSES
  Get data
  Compute license fee
  Print results
END
```

Since each of these English phrases will be translated into several BASIC statements, it is best to put the details in subroutines. Here is the finished main routine.

```

100 'PROGRAM LICENSES
110   GOSUB 200 'Get data
120   GOSUB 300 'Compute license fee
130   GOSUB 400 'Print results
140   END

```

### Program Skeleton

Now you know how many subroutines are needed and where they begin. The next step is to block out an outline of the main routine and the subroutines. Include dummy **PRINT** statements in each subroutine so that you can enter and run the skeleton version of the program. Here is the skeleton program:

```

100 'PROGRAM LICENSES
110   GOSUB 200 'Get data
120   GOSUB 300 'Compute license fee
130   GOSUB 400 'Print results
140   END
190 '
200 'SUB GET DATA
205   PRINT "DATA"
280   RETURN
290
300 'SUB COMPUTE LICENSE FEE
305   PRINT "COMPUTE FEE"
380   RETURN
390 '
400 'SUB PRINT RESULTS
405   PRINT "RESULTS"
480   RETURN

```

At this point, you can enter the skeleton version of the program and run it. You should see the following phrases if you do this:

```

DATA
COMPUTE FEE
RESULTS

```

### Data Subroutine

Now that the skeleton program is finished, you can complete each of the subroutine bodies. However, concentrate on only one subroutine at a time, thereby making the task easier.

The data subroutine is first. Here is an outline of the subroutine, showing the simple task it performs and the kind of program block that is needed:

```
200 'SUB GET DATA
```

*Get data  
(Action block)*

```
280 RETURN
```

With this outline, you can fill in the line numbers and the necessary BASIC statements easily.

```
200 'SUB GET DATA
210 CLS
220 PRINT "INPUT AUTO HP ";
230 INPUT HP
280 RETURN
```

### Fee Subroutine

The next subroutine determines what the license fee is. Again, here is an outline of the task to be done and the program block that can do it:

```
300 'SUB COMPUTE LICENSE FEE
```

*Decide what fee is  
(Case block)*

```
380 RETURN
```

The case block is the best structure for computing the fee. However, the point has been made several times that only the action, loop, and branch blocks are needed to do any programming task. You could use nested branch blocks to compute the fee; the case block is convenient but not necessary.

Here is a fleshed-out outline of the subroutine:

```
300 'SUB COMPUTE LICENSE FEE
    'CASE
    IF horsepower is 50 or less THEN fee is 0
    IF horsepower is 100 or less THEN fee is 30
    IF horsepower is 200 or less THEN fee is 70
    IF horsepower is 300 or less THEN fee is 150
    IF horsepower is more than 300 THEN fee is 500
    'END CASE
380 RETURN
```

Now it is easy to complete the remaining details in the subroutine. Here is the finished subroutine:

```
300 'SUB COMPUTE LICENSE FEE
310 'CASE
320 IF HP > 0 AND HP <= 50 THEN LET FEE = 0
```

```

330      IF HP > 50 AND HP <= 100 THEN LET FEE = 30
340      IF HP > 100 AND HP <= 200 THEN LET FEE = 70
350      IF HP > 200 AND HP <= 300 THEN LET FEE = 150
360      IF HP > 300 THEN LET FEE = 500
370      'END CASE
380      RETURN

```

### Results Subroutine

The purpose of the final subroutine is to print out the results. Here is the outline of the subroutine:

```

400      'SUB PRINT RESULTS

          Print results
          (Action block)

480      RETURN

```

There is nothing particularly complicated about this subroutine. Here is the completed version:

```

400      'SUB PRINT RESULTS
410      PRINT "LICENSE FEE IS ";
420      PRINT FEE;
430      PRINT " DOLLARS"
480      RETURN

```

### Final Program

Now you can put the complete program together. Here it is:

```

100      'PROGRAM LICENSES
110      GOSUB 200 'Get data
120      GOSUB 300 'Compute license fee
130      GOSUB 400 'Print results
140      END
190      '
200      'SUB GET DATA
210      CLS
220      PRINT "INPUT AUTO HP ";
230      INPUT HP
280      RETURN
290      '
300      'SUB COMPUTE LICENSE FEE
310      'CASE
320      IF HP > 0 AND HP <= 50 THEN LET FEE = 0
330      IF HP > 50 AND HP <= 100 THEN LET FEE = 30
340      IF HP > 100 AND HP <= 200 THEN LET FEE = 70

```

```

350      IF HP > 200 AND HP <= 300 THEN LET FEE = 150
360      IF HP > 300 THEN LET FEE = 500
370      'END CASE
380      RETURN
390      '
400      'SUB PRINT RESULTS
410      PRINT "LICENSE FEE IS ";
420      PRINT FEE;
430      PRINT " DOLLARS"
480      RETURN

```

This example was quite simple. The next one is more challenging.

### Example 2 - Averaging the Positive and Negative Numbers in a List

Suppose you want to average a list of numbers but don't know in advance how many numbers are in the list. You do know, however, that after the final number, the flag value 999 will be entered. The numbers in the list can be positive, negative, or zero.

The problem is to write a program that will allow you to enter the numbers, one by one, until the flag value is entered. Then, the computer should compute the average of the positive numbers in the list as well as the average of the negative numbers.

#### The Main Routine

Begin by making a paper-and-pencil outline of the main routine. Here is how such an outline might look:

```

PROGRAM AVERAGE
  Initialize
  Get data and compute sums
  Print results
END

```

The three English phrases describe the three main tasks to be done. Each of these tasks will require a subroutine. Here is the finished main routine:

```

1000 'PROGRAM AVERAGE
1010   GOSUB 2000 'Initialize
1020   GOSUB 3000 'Get data and compute sums
1030   GOSUB 4000 'Print results
1040   END

```

#### Skeleton Program

After finishing the main routine, you can write a skeleton version of the program that contains the main routine and empty subroutines.

```

1000 'PROGRAM AVERAGE
1010   GOSUB 2000 'Initialize

```



```

1020      GOSUB 3000 'Get data and compute sums
1030      GOSUB 4000 'Print results
1040      END
1990      '
2000      'SUB INITIALIZE
2005          PRINT "INITIALIZE"
2980      RETURN
2990      '
3000      'SUB GET DATA AND COMPUTE SUMS
3005          PRINT "DATA AND SUM"
3980      RETURN
3990      '
4000      'SUB PRINT RESULTS
4005          PRINT "RESULTS"
4980      RETURN

```

As usual, blank lines have been used to separate the routines. Also, dummy **PRINT** statements have been put in each subroutine so that the skeleton program can be entered and run. If you do this, you should see the three phrases

```

INITIALIZE
DATA AND SUM
RESULTS

```

This confirms that the computer is going through the subroutines in the right order.

### Initialization Subroutine

Now you can concentrate on each of the subroutines in turn. Again, it is wise to do a little paper-and-pencil planning before getting too deep into writing statements. The trick is to ask yourself, "What program block do I need?" Don't get bogged down at this stage thinking about individual statements.

What sorts of tasks must the initialization subroutine do? If it is to compute averages, you must initialize variables to hold the necessary sums. Also, you must initialize variables to keep track of the number of positive numbers, negative numbers, and zeros in the list. Here is an outline of the tasks to be performed and the program blocks necessary to do each task:

```

2000      'SUB INITIALIZE

           Set counting variables to zero
           (Action block)

           Set summing variables to zero
           (Action block)

2980      RETURN

```

Let's use the variables **COUNTPOS**, **COUNTNEG**, and **COUNTZER** to hold the number of positive numbers, negative numbers, and zeros, respectively. Similarly,

SUMPOS and SUMNEG will hold the sum of the positive numbers and the sum of the negative numbers in the list. With these definitions in place, you can complete the body of the subroutine.

```

2000 'SUB INITIALIZE
2010     LET COUNTPOS = 0
2020     LET COUNTNEG = 0
2030     LET COUNTZER = 0
2040     LET SUMPOS = 0
2050     LET SUMNEG = 0
2980     RETURN

```

### Data Subroutine

The next subroutine must carry out two main tasks. Here is an outline:

```

3000 'SUB GET DATA AND COMPUTE SUMS

    Enter data
    (Loop)

    If data is input, decide what to do with it
    (Subroutine)

3980     RETURN

```

It is clear that a loop is needed to handle input of the numbers in the list. The top part of the loop will contain the INPUT statement. When it executes the bottom part of the loop, the computer must decide what to do with the data. This task probably will get complicated, so let's bury the details in another subroutine.

A refined outline of the subroutine is shown below:

```

3000 'SUB GET DATA AND COMPUTE SUMS
    'LOOP
        Enter a value
        IF value is flag THEN leave loop
        Go to a subroutine to check value
    'END LOOP
3980     RETURN

```

Now the subroutine can be completed easily. Here is the finished version:

```

3000 'SUB GET DATA AND COMPUTE SUMS
3010 ' 'LOOP
3020     INPUT X
3030     IF X = 999 THEN 3060
3040     GOSUB 5000 'Check X and sum
3050     GOTO 3010
3060     'END LOOP
3980     RETURN

```

**Results Subroutine**

When the results are computed and output, you must consider a number of possibilities.

1. All the numbers could be zero.
2. All the numbers could be positive.
3. All the numbers could be negative.
4. The list could contain a combination of zeros, positive numbers, and negative numbers.

If all the numbers in the list are positive, then **COUNTNEG** will be zero. In this case, when the computer attempts to compute the average of the negative numbers, it tries to divide by zero. Division by zero causes an error message, and the program will stop. The same thing applies if there are no positive numbers, or if the numbers are all zero. These conditions must be anticipated in the body of the subroutine.

Here is an outline of the subroutine that shows the tasks to be done and the program structure needed to accomplish each task.:

**4000 'SUB PRINT RESULTS**

*Check for no positive and no negative numbers  
(Abbreviated branch block)*

*If there are positive numbers, compute average  
(Standard branch block)*

*If there are negative numbers, compute average  
(Standard branch block)*

*Print number of zeros in list  
(Action block)*

**4980 RETURN**

Next, flesh out the outline by inserting the skeleton of each of the program blocks.

**4000 'SUB PRINT RESULTS**

**IF no pos or neg numbers THEN message ELSE leave branch**  
*Print message*  
**'END IF**

**IF no pos numbers THEN message ELSE compute average**  
**'THEN CLAUSE**  
*Print message*  
**'ELSE CLAUSE**  
*Compute average of pos numbers*  
**'END IF**

```

    IF no neg numbers THEN message ELSE compute average
    'THEN CLAUSE
        Print message
    'ELSE CLAUSE
        Compute average of neg numbers
    'END IF

```

*Print number of zeros*

```
4980 RETURN
```

At this point, it is easy to put in line numbers and complete the subroutine.

```

4000 'SUB PRINT RESULTS
4010     IF COUNTPOS = 0 AND COUNTNEG = 0 THEN 4020
        ELSE 4030
4020     PRINT "NO NON-ZERO NUMBERS IN LIST"
4030     'END IF
4040 '
4050     IF COUNTPOS = 0 THEN 4060 ELSE 4090
4060     'THEN CLAUSE
4070         PRINT "NO POS. NUMBERS"
4080         GOTO 4120
4090     'ELSE CLAUSE
4100         PRINT "AVE. OF POS. NUMBERS = ";
4110         PRINT SUMPOS / COUNTPOS
4120     'END IF
4130 '
4140     IF COUNTNEG = 0 THEN 4150 ELSE 4180
4150     'THEN CLAUSE
4160         PRINT "NO NEG. NUMBERS"
4170         GOTO 4210
4180     'ELSE CLAUSE
4190         PRINT "AVE. OF NEG. NUMBERS = ";
4200         PRINT SUMNEG / COUNTNEG
4210     'END IF
4220 '
4230     PRINT "NUMBER OF ZEROS = ";
4240     PRINT COUNTZER
4980 RETURN

```

The **ELSE 4030** below line 4010 is part of line 4010. Program line 4010 is wrapped around in this way because it is too wide to fit on one line of the text. However, it will fit on your 80 column screen. You will see such occurrences several times in the remainder of the book.

## Checking Subroutine

The purpose of this subroutine is to tell the computer what to do with each piece of data as it is entered. You know that  $X$  must be either positive, zero, or negative. There are no other possibilities. Each of these possibilities must be accounted for in the body of the subroutine. Here is an outline listing each task to be performed and the program block necessary to do the task.

5000 'SUB CHECK X AND SUM

*Take care of negative X*  
*(Abbreviated branch block)*

*Take care of zero X*  
*(Abbreviated branch block)*

*Take care of positive X*  
*(Abbreviated branch block)*

5980 RETURN

As before, each of these task descriptions can be replaced with an outline of the appropriate program block.

5000 'SUB CHECK X AND SUM

IF *negative X* THEN *process X* ELSE *leave branch*  
'THEN CLAUSE  
*Increment sum and count*  
'END IF

IF *zero X* THEN *process X* ELSE *leave branch*  
'THEN CLAUSE  
*Increment count*  
'END IF

IF *positive X* THEN *process X* ELSE *leave branch*  
'THEN CLAUSE  
*Increment sum and count*  
'END IF

5980 RETURN

The final step is to add the line numbers and replace all English words and phrases with appropriate BASIC words and line numbers.

```
5000 'SUB CHECK X AND SUM
5010   IF X < 0 THEN 5020 ELSE 5050
5020   'THEN CLAUSE
5030       LET SUMNEG = SUMNEG + X
5040       LET COUNTNEG = COUNTNEG + 1
5050   'END IF
```



```

5060 '
5070   IF X = 0 THEN 5080 ELSE 5100
5080   'THEN CLAUSE
5090       LET COUNTZER = COUNTZER + 1
5100   'END IF
5110 '
5120   IF X > 0 THEN 5130 ELSE 5160
5130   'THEN CLAUSE
5140       LET SUMPOS = SUMPOS + X
5150       LET COUNTPOS = COUNTPOS + 1
5160   'END IF
5980 RETURN

```

### Complete Program

That finishes the program. The complete program looks like this:

```

1000 'PROGRAM AVERAGE
1010   GOSUB 2000 'Initialize
1020   GOSUB 3000 'Get data and compute sums
1030   GOSUB 4000 'Print results
1040 END
1990 '
2000 'SUB INITIALIZE
2010   LET COUNTPOS = 0
2020   LET COUNTNEG = 0
2030   LET COUNTZER = 0
2040   LET SUMPOS = 0
2050   LET SUMNEG = 0
2980 RETURN
2990 '
3000 'SUB GET DATA AND COMPUTE SUMS
3010   'LOOP
3020       INPUT X
3030       IF X = 999 THEN 3060
3040       GOSUB 5000 'Check X and sum
3050       GOTO 3010
3060   'END LOOP
3980 RETURN
3990 '
4000 'SUB PRINT RESULTS
4010   IF COUNTPOS = 0 AND COUNTNEG = 0 THEN 4020
4020       PRINT "NO NON-ZERO NUMBERS IN LIST"
4030   'END IF
4040 '
4050   IF COUNTPOS = 0 THEN 4060 ELSE 4090
4060   'THEN CLAUSE
4070       PRINT "NO POS. NUMBERS"

```

```

4080      GOTO 4120
4090      'ELSE CLAUSE
4100      PRINT "AVE. OF POS. NUMBERS = ";
4110      PRINT SUMPOS / COUNTPOS
4120      'END IF
4130      '
4140      IF COUNTNEG = 0 THEN 4150 ELSE 4180
4150      'THEN CLAUSE
4160      PRINT "NO NEG. NUMBERS"
4170      GOTO 4210
4180      'ELSE CLAUSE
4190      PRINT "AVE. OF NEG. NUMBERS = ";
4200      PRINT SUMNEG / COUNTNEG
4210      'END IF
4220      '
4230      PRINT "NUMBER OF ZEROS = ";
4240      PRINT COUNTZER
4980      RETURN
4990      '
5000      'SUB CHECK X AND SUM
5010      IF X < 0 THEN 5020 ELSE 5050
5020      'THEN CLAUSE
5030      LET SUMNEG = SUMNEG + X
5040      LET COUNTNEG = COUNTNEG + 1
5050      'END IF
5060      '
5070      IF X = 0 THEN 5080 ELSE 5100
5080      'THEN CLAUSE
5090      LET COUNTZER = COUNTZER + 1
5100      'END IF
5110      '
5120      IF X > 0 THEN 5130 ELSE 5160
5130      'THEN CLAUSE
5140      LET SUMPOS = SUMPOS + X
5150      LET COUNTPOS = COUNTPOS + 1
5160      'END IF
5980      RETURN

```

You should take a few minutes to enter this program and test it. Try various lists of numbers that will exercise all parts of the program.

## 7—5 PROBLEMS

1. Write a BASIC program that calls for the input of two numbers and then prints out the larger.
2. Write a BASIC program that reads three numbers from a **DATA** statement and then prints out the smallest.
3. Suppose a program has a **DATA** statement that contains a list of numbers. You don't know the length of the list. However, the end of the list is marked with the flag

variable 999. Write a BASIC program to compute and print out the sum of the numbers between  $-10$  and  $+10$ , inclusive.

4. Usually supermarkets use a different markup for different items depending upon the unit cost of the item. Suppose this markup is based on the following schedule:

Unit Cost	Mark up
0 to \$1.00	20%
\$1.01 to \$2.00	10%
over \$2.00	5%

The unit cost is determined by dividing the case price by the number of items in the case. Write a program to compute the label price which is the unit cost plus the markup. Arrange for prompts and input any way you desire.

5. Consider the series

$$1 + (1/2)^2 + (1/3)^2 + (1/4)^2 + \dots$$

Write a program to compute the sum of the series until the difference between the present sum and the sum obtained by including just one more term is less than a value **EPSILON**. Run the program with values of **EPSILON** equal to 0.001, 0.0001, 0.00001, and 0.000001. What do you think the sum of the series would be if the terms went on forever?

6. Write a program that will accept input from the keyboard until the flag value 999 is entered. Print out both the largest and smallest values entered up to but not including the flag value.
7. Write a program to accept the input of two numbers. If both the numbers are greater than or equal to 10, print out their sum. If both the numbers are less than 10, print out their product. If one number is greater than or equal to 10 and the other is less than 10, print out the difference between them.
8. An instructor decides to award letter grades on an examination as follows:

90 to 100	A
80 to 89	B
60 to 79	C
50 to 59	D
0 to 49	F

Write a program to produce the following output when executed:

```
INPUT EXAM GRADE ?   (You enter number grade)
YOUR GRADE IS       (Computer types out A, B, C, D, or F)
```

9. A set of integers (whole numbers) is chosen at random from the set 1, 2, 3, 4 and put in a **DATA** statement. The end of the set is marked with the flag 999. Write a program that computes and prints out the numbers of 1s, 2s, 3s, and 4s in the set. Test your program on the following **DATA** statement:

```
DATA 3, 1, 2, 1, 4, 4, 1, 2, 2, 2, 3, 999
```

10. Write a BASIC program that calls for N grades to be input. Compute and print out (1) the highest grade, (2) the lowest grade, and (3) the average of the grades.

## 7—6 PRACTICE TEST

1. What will be output if the following program is executed?

```

100 'PROGRAM TEST
110   FOR J = 1 TO 3
120       GOSUB 200 'You decide
130   NEXT J
140 END
190 '
200 'SUB YOU DECIDE
210   'CASE
220       IF J = 3 THEN PRINT "BEST"
230       IF J = 2 THEN PRINT "BETTER"
240       IF J = 1 THEN PRINT "GOOD"
250   'END CASE
260 RETURN

```

---

2. Suppose that you decide to buy a number of widgets. The manufacturer is pushing sales and will give reduced prices for large orders. The price schedule is as follows:

Number Purchased	Price per Widget
20 or less	\$2.00
21 to 50	1.80
51 or more	1.50

Write a program that will produce the following output when executed:

```

HOW MANY WIDGETS ? (You enter purchase quantity)
PRICE PER WIDGET IS (Computer prints unit price)
TOTAL COST OF ORDER IS (Computer prints total)

```

Then keep looping through the program.

---

3. Write a program that calls for the input of three numbers, then computes and prints out the average of the two smaller numbers.
-

4. Write a program that prints out a table of whole numbers beginning with 1 and the sum of all the whole numbers up to the current one. For example:

1 = 1  
1 + 2 = 3  
1 + 2 + 3 = 6  
1 + 2 + 3 + 4 = 10  
and so on.

Keep printing the sums as long as the last one printed is less than 100.

---

5. Write a program that calls for the input of a positive whole number from the keyboard and then computes the product of all the whole numbers from 1 up to the number entered. As the product is being computed, abandon the calculation if the running product exceeds 10,000. If not, print out the product.
-





# FUNCTIONS

## 8—1 OBJECTIVES

### Seeing Functions in Action

A function is defined to perform a specific task. You will learn the characteristics of functions and how they are used.

### Using Built-in String and Numeric Functions

Some ready-made functions in BASICA perform tasks that result in a string value. You will learn to use these functions. Also, other ready-made functions in BASICA perform tasks that result in a numeric value. You will learn to use these functions as well.

### Creating User-Defined Functions

Users can define functions that meet their special needs. You will use the **DEF** statement to create such functions.

### Working with Program Examples

You will apply what you learn to programming problems.

## 8—2 DISCOVERY EXERCISES

### Built-in Functions

1. Turn on the computer and bring up BASICA.
2. Enter the following program:

```

100 'PROGRAM EXPLORING FUNCTIONS
110 'LOOP FOREVER
120     INPUT "ENTER FUNCTION PARAMETER: "; A$
130     PRINT LEN(A$)
140     GOTO 110
150 'END LOOP
160 END

```

3. Use the **[F2]** key to run the program. At the input prompt, enter your first name. What was displayed?

---

How many letters are in your first name?

---

4. Now enter **TURN** at the input prompt. What number was displayed?

---

Enter **TURNOVER** at the input prompt. What number was displayed this time?

---

5. Now for another variation. At the input prompt, enter **R O B E R T**, including all the spaces between the letters. What number was displayed?

---

How many letters and spaces are there in **R O B E R T**?

---

What does the **LEN** function do?

---

6. Look at another function. Use `Ctrl|Break` to interrupt the program. Edit lines 120 and 130 as follows:

```
120    INPUT "ENTER FUNCTION PARAMETER :"; X
130    PRINT SQR(X)
```

Run the program and at the input prompt enter 4. What happened?

---

Enter 9 and record the result.

---

Now enter 25 and record the result below.

---

Finally, enter 10. What number was displayed?

---

7. What do you think the **SQR** function does?

---

8. **INT** is another built-in function. Interrupt the program. Edit line 130 as follows:

```
130    PRINT INT(X)
```

Run the program for the following values of **X**. In each case, record in the table below what was displayed on the screen:

X	VALUE
1	_____
3.4	_____
256.78	_____
0	_____
-1.2	_____
-2.3	_____

Examine the values you have recorded above and compare each value with the corresponding value of **X**. What do you think the **INT** function does?

---

9. Interrupt the program. Edit line 130 as follows:

```
130 PRINT SGN(X)
```

Display the program and review its structure to refresh your memory about how the program works. Run the program for each of the following values of **X**. In each case, record the output in the table below:

X	Value
1.5	_____
43	_____
128.3	_____
0	_____
-1	_____
-1.2	_____
-345.7	_____
4.7	_____
-5.8	_____

Examine carefully the values you recorded above. What do you think the **SGN** function does?

\_\_\_\_\_

10. Go on to another function. Interrupt the program. Edit line 130 as follows:

```
130 PRINT ABS(X)
```

Use the values of **X** shown below and fill in the following table:

X	Value
3.4	_____
0	_____
-3.4	_____
-2	_____
-8.45	_____
8.45	_____

Examine the values you recorded. What do you think the **ABS** function does?

\_\_\_\_\_

11. The previous functions you explored have all returned numeric values. The function **MID\$** returns a string value. Interrupt the program. Edit lines 120 and 130 as follows:

```
120 INPUT "ENTER 3 FUNCTION PARAMETERS: "; A$, X, Y
130 PRINT MID$(A$, X, Y)
```

Run the program and enter the parameters **MISSISSIPPI**, **4**, **3** at the input prompt. What was displayed?

\_\_\_\_\_



Enter the following parameters: **PAIRINGS**, **2**, **3**. What word was displayed?

---

Now enter the following parameters: **PAIRINGS**, **4**, **4**. What word was displayed this time?

---

- 12.** Enter the following parameters: **PAIRINGS**, **6**, **1**. How many letters were displayed?

---

What position does the letter **N** occupy in **PAIRINGS**?

---

When you run the program, what parameters would you enter to have the computer display the **G** in **PAIRINGS**.

---

Try those parameters and see if you were correct.

- 13.** You will now explore two functions that are closely related. Interrupt the program. Edit lines 120 and 130 as follows:

```
120    INPUT "ENTER FUNCTION PARAMETER: "; X
130    PRINT CHR$(X)
```

Run the program and at the input prompt enter **65**. What was displayed?

---

Enter **66** at the input prompt? What letter was displayed?

---

Experiment with this program; try various numeric inputs. Keep the numbers in the range **32** to **255**. Record the results below.

---

As you can see, you can refer to a character either by the character itself or by a number, its position in a set of characters called the American Standard Code for Information Interchange (ASCII).

14. The function related to **CHR\$** is **ASC**. **ASC** is the function that converts a character to its equivalent position number in the ASCII character set. Interrupt the program. Edit lines 120 and 130 as follows:

```
120    INPUT "ENTER FUNCTION PARAMETER: "; A$
130    PRINT ASC(A$)
```

Run the program and at the input prompt enter **Z**. What was displayed?

---

Enter **A**. What number was displayed this time?

---

How does this relate to the value of **CHR\$(65)** you explored in step 13?

---

15. Check one last detail about this function. Enter each of the following parameters: **P**, **PEA**, **POTATO**. What happened?

---

16. Another pair of related functions is **VAL** and **STR\$**. Recall that a number can be considered as a string of digits. Thus a number can be entered as a string of digits. **VAL** is the function that converts such strings of digits to their numeric value. Interrupt the program. Edit lines 120 and 130 as follows:

```
120    INPUT "ENTER TWO STRINGS OF DIGITS: "; A$; B$
130    PRINT VAL(A$) + VAL(B$)
```

Run the program. Enter the two strings of digits **123** and **456** at the input prompt. What was displayed?

---

Is this result the joining of the two strings or the arithmetic sum of the numbers represented by the strings?

---

17. The **STR\$** function converts a number to its string of digits. Interrupt the program. Edit lines 120 and 130 as follows:

```
120    INPUT "ENTER TWO NUMBERS: "; X, Y
130    PRINT STR$(X) + STR$(Y)
```

Run the program. Enter the numbers -111 and -222. What happened?

---

Were the numbers added or joined together?

---

Functions that return string values end in a \$, while functions that return numeric values do not.

18. Interrupt the program. Clear the memory and the screen. Enter the following program:

```
100 'PROGRAM GENERATING RANDOM NUMBERS
110  RANDOMIZE(TIMER) 'Seed RND generator
120  FOR K = 1 TO 9
130    PRINT RND
140  NEXT K
150  END
```

Run the program. Record the largest and smallest numbers that were displayed.

---

Run the program again. How do the largest and smallest numbers compare with those of the previous run?

---

19. Edit line 110 as follows:

```
110  RANDOMIZE
```

Run the program. When asked to enter a seed, enter 257. What happened?

---

Run the program again. Enter -374 as the seed. How do the two sets of numbers compare?

---

Run the program one more time. Enter the seed -374 again. How do the two sets of numbers on the screen compare this time?

---

20. Edit line 110 as follows:

```
110    RANDOMIZE(-374)
```

Run the program. How do the two sets of numbers on the screen compare now?

---

If you use a constant seed, the random number sequence is the same each you run a program. This can be useful in debugging programs.

### User-Defined Functions

21. Clear the memory and the screen. Enter the following program.

```
100 'PROGRAM CREATING YOUR OWN FUNCTIONS
110    DEF FN A(FILLIN) = 5 * FILLIN + 4
120    INPUT "ENTER FUNCTION PARAMETER: "; X
130    PRINT FN A(X), 5 * X + 4
140    END
```

Before running the program, complete the following table for the two expressions in the program above:

FILLIN	$5 * \text{FILLIN} + 4$	X	$5 * X + 4$
1	$5 * 1 + 4 = 9$	1	$5 * 1 + 4 = 9$
2		2	
5		5	

Use **[F2]** to run the program. Enter **2** at the input prompt. How do the two values displayed compare?

---

Run the program again. Enter **5** at the input prompt. What numbers are displayed this time?

---

In line 130, **FN A(X)** determines the value of the function you defined. It does so by replacing the variable **FILLIN** in line 110 with the number you entered for **X** in line 120. Thus, both the numbers displayed by the program are the same.

22. Now change the function. Edit lines 110 and 130 as follows:

```
110    DEF FN A(FILLIN) = FILLIN ^ 2 + 1
130    PRINT FN A(X)
```

Display the program. Run the program and enter **2** at the input prompt. What number is displayed?

---

If you were to run the program and enter 5 at the input prompt, what value do you think would be displayed?

---

Try it and see if you were correct.

23. Clear the memory and the screen. Enter the following program:

```
100 'PROGRAM DOUBLE ACTION
110   DEF FN TRIPLE(PARAMETER) = 3 * PARAMETER
120   INPUT "ENTER FUNCTION PARAMETER: "; X
130   PRINT FN TRIPLE(SQR(X))
140   END
```

Display the program. Look at line 130 and try to determine what value will be returned if **X** is 4. Run the program and enter 4 at the input prompt. What number was displayed?

---

Run the program again and enter 25 at the input prompt. What number was displayed this time?

---

24. As you saw with **MID\$**, functions can have more than one parameter. Edit the program in memory as follows:

```
100 'PROGRAM DOUBLE PARAMETER
110   DEF FN AREA(BASE, HEIGHT) = .5 * BASE * HEIGHT
120   INPUT "ENTER 2 FUNCTION PARAMETERS: "; B, H
130   PRINT FN AREA(B, H)
```

Display the program and check it to make sure it is correct. Run the program and enter the two numbers 4 and 6 at the input prompt. What area is displayed?

---

25. Clear the memory and the screen. Enter the following program:

```
100 'PROGRAM STRING FUNCTION
110   DEF FN ABBREV$(F$,LAST$) = MID$(F$,1,1) + ". "
      + LAST$
120   INPUT "ENTER FIRST & LAST NAME: "; F$, LAST$
130   PRINT FN ABBREV$(F$, LAST$)
140   END
```

Run the program. Enter your first and last name, in that order (remember to separate them by a comma). How was your name abbreviated?

---



Run the program again and enter **ALOYSIUS, CARPENTER**. How was this name abbreviated?

---

26. This ends the Discovery Exercises. Turn off the computer and go on to the discussion.

## 8—3 DISCUSSION

### Seeing Functions in Action

Recall that a subroutine is a package of instructions created to perform some particular task or process. Depending on the task, the subroutine can return several values, one value, or no value. A function is similar to a subroutine in that it is also defined to perform a specified task. However, a function always returns one value. The computer replaces the function with its value in the expression that contains the function. For example in

```
LET B$ = "BOOK"
LET A$ = MID$(B$,1,3)
```

**MID\$(B\$,1,3)** returns the value **B00** derived from the string function **MID\$** using the parameters **B\$**, **1** and **3**. The returned value **B00** replaces the function and is then assigned to the variable **A\$**.

Every function has a name, a type, and a value. The name of the function is the set of letters used to identify the function. The value of the function is whatever the function returns. The type of the function is determined by the returned value. If the returned value is numeric, the type of the function is numeric. If the returned value is string, the type of the function is string. For example, in the direct-mode statement

```
PRINT SQR(4)
```

**SQR** is the name of the square root function. The value of the **SQR** function is the square root (2) of the function parameter (4). The type of the function is therefore numeric.

Functions and variables are similar in that they each have a name, a type, and a value. However, a function's value (string or numeric) is determined by its definition and parameter(s). By contrast, a variable's value is the contents of its memory location and is determined by a **LET**, **READ**, or **INPUT** statement.

- **Functions are defined. Variables are assigned.**

### Using Built-in String and Numeric Functions

BASICA has numerous built-in functions. The string functions are:

**CHR\$:** A string function that returns the ASCII character associated with the numeric parameter.

**MID\$:** A three-parameter string function that returns characters from the first (string) parameter. **MID\$** returns the number of characters given by the third (numeric) parameter starting at the position given by the second (numeric) parameter.

**STR\$:** A string function that returns the string representation of the numeric parameter.

The numeric functions are:

**ABS:** A numeric function that returns the absolute value of the parameter. If the parameter is positive or zero, it returns the parameter. If the parameter is negative, it returns the parameter times  $-1$ .

**ASC:** A numeric function that returns the ASCII ordinal code of the first character of the string parameter.

**INT:** A numeric function that returns the greatest integer less than or equal to the parameter.

**LEN:** A numeric function that returns the length of the string parameter.

**RND:** A numeric function that returns a random number in the range 0 to 1.

**SGN:** A numeric function that returns  $-1$ , 0, or 1. If the parameter is negative,  $-1$  is returned. If the parameter is 0, 0 is returned. If the parameter is positive, 1 is returned.

**SQR:** A numeric function that returns the square root of the parameter. If the parameter is not positive or zero, an error message is displayed.

**VAL:** A numeric function that converts a string representation of a number to its numeric value. The numeric value is returned.

Built-in functions are predefined to have a specified name, type, and rule for determining the returned value. If the value returned by the function is a string, the function name ends in a \$ sign. Otherwise, the function is a numeric function, which means it returns a numeric value.

All built-in functions act in essentially the same way. An example of a numeric function is the direct-mode statement

```
LET P = SQR(36)
```

which returns 6, the square root of 36, and assigns it to P. 36 is the parameter and **SQR** is the name of the numeric function whose rule has been predefined in BASICA as "take the square root."

The string function in the direct-mode statement

```
LET A$ = MID$("ABCDE",3,2)
```

returns 2 characters from the string **ABCDE** starting at character position 3. The value of the **MID\$** function in this case is therefore the string **CD**, which is assigned to the variable **A\$**. All three parameters can be variables or expressions. The first parameter must be a string variable or expression, while the second and third parameters must be numeric.

Most function have one or more parameters. However, the **RND** function is special; its parameter is implied if it is not specified. The computer keeps a count of how many times the **RND** function has been used since the last **RANDOMIZE** statement was performed. This count, along with the seed number you saw in the computer work, is used to generate the value of the **RND** function. For example, when you run the program

```

100 'PROGRAM RND EXAMPLE
110   RANDOMIZE(347)
120   PRINT RND
130   PRINT RND
140   PRINT RND
150   END

```

three random numbers are displayed. They are the first three numbers in the random number list whose seed is 347. If you entered and ran the program above and then entered the direct-mode statement

```
PRINT RND
```

the fourth random number in the list whose seed is 347 would be displayed. In this sense, the random-number counter is the implied parameter of the **RND** function.

If you change the seed number in the **RND EXAMPLE** program above to 544, then the three numbers displayed are different; they are the first three numbers of the list of random numbers associated with the new seed, 544. If you use **TIMER** as the parameter of the **RANDOMIZE** statement, then the seed is based on the number of seconds on the internal clock of the computer. If you use **RANDOMIZE** without a parameter, the computer asks you to input a seed value.

### Creating User-Defined Functions

The **DEF** (for “define”) statement permits user-defined functions in BASIC programs in addition to those functions (**SQR**, **MID\$**, **INT**, etc.) already built into the language. The form of all **DEF** statements is the same.

```
< line #> DEF FN < name>(< parameter(s)>) = < rule>
```

where *<name>* is a valid BASICA name, *<parameter(s)>* is a variable name or a list of variable names, and *<rule>* is a valid BASICA expression based on the parameter(s). For example,

```
110   DEF FN P(X) = X ^ 2 - 3 * X
```

defines a numeric function **P**. If you use this line in a program, and later use the expression **FN P(2)**, the computer would “look up” the definition of **FN P** in line 110 and then substitute 2 for **X** on the right side of the equal sign in the **DEF** statement, with this result

```
FN P(2) = -2
```

Likewise, if **TIME** = 5, then

```
FN P(TIME) = 10
```

You can use the built-in functions of BASICA in **DEF** statements. For example,

```
110 DEF FN B(Y) = SQR(Y ^ 1.5) + 3 * Y
```

is legal. Furthermore, you may use other defined functions in a **DEF** statement. For example,

```
110 DEF FN B(Y) = FN A(Y) + SQR(Y)
```

is permitted provided you have defined the function **FN A** previously.

You can also define string functions and functions with more than one parameter. For example,

```
110 DEF FN CUT$(A$,X) = MID$(A$,1,X)
```

defines a function that returns the first **X** characters of the string **A\$**.

The primary purpose of the user-defined functions is to simplify programming by avoiding repeated use of complicated expressions. You should be alert for opportunities to use the **DEF** statement.

■ Use the **DEF** statement to define your own functions.

## 8—4 WORKING WITH PROGRAM EXAMPLES

### Example 1 - Exact Division

The problem is to write a program that computes all the integers (whole numbers) that divide exactly into another integer. For example, if 8 is the test integer, you want to find all the integers (from 1 to 8) that divide exactly into 8 with no remainder. Those divisors are 1, 2, 4, and 8.

If **N** is the test integer, then **X** is an exact divisor of **N** when **N/X** is an integer. The rule to use is if **N / X = INT(N / X)**, then **X** is an exact divisor.

The first step is to create an outline. Here is a possible outline:

```
PROGRAM EXACT DIVISORS
  Input test number
  Compute & display divisors
END
```

Since you can express the request for a test number in one BASIC line, the main routine skeleton might look like this:



```

100 'PROGRAM EXACT DIVISORS
110   INPUT "ENTER A WHOLE NUMBER: "; N
120   GOSUB 200 'Compute & display divisors
130   END

```

Here is a possible skeleton outline of the subroutine needed to compute and display the divisors.

```

200 'SUB COMPUTE & DISPLAY DIVISORS
    Check all integers from 1 to N
    (Loop)
    Test whether each integer divides or not
    (One-way branch block)
280 RETURN

```

Now flesh out the outlines of the two structures needed in the subroutine.

```

200 'SUB COMPUTE & DISPLAY DIVISORS
    FOR variable = 1 TO N
        IF exact divisor THEN print variable ELSE leave branch
    'THEN CLAUSE
        print variable
    'END IF
    NEXT variable
280 RETURN

```

At this point, it is easy to complete the subroutine. Here is the finished version:

```

200 'SUB COMPUTE & DISPLAY DIVISORS
210   FOR X = 1 TO N
220     IF N / X = INT(N / X) THEN 230 ELSE 250
230     'THEN CLAUSE
240       PRINT X 'Exact divisor
250     'END IF
260   NEXT X
280 RETURN

```

Here is the finished program:

```

100 'PROGRAM EXACT DIVISORS
110   INPUT "ENTER A WHOLE NUMBER: "; N
120   GOSUB 200 'Compute & display divisors
130   END
190 '
200 'SUB COMPUTE & DISPLAY DIVISORS
210   FOR X = 1 TO N
220     IF N / X = INT(N / X) THEN 230 ELSE 250
230     'THEN CLAUSE
240       PRINT X, 'Exact divisor
250     'END IF

```



```

260     NEXT X
280     RETURN

```

Enter the program and test with fairly large values of N. Can you think of a way to make the program run in half the time?

## Example 2 - String Reversal

The task is to write a program that calls for the input of a string and then prints it back in reverse order. If you think about the solution away from the computer, you might arrive at the following outline:

```

PROGRAM STRING REVERSAL
  Input string
  Reverse and display
END

```

The main routine skeleton would then be

```

100 'PROGRAM STRING REVERSAL
110   'Input string
120   'Reverse and display
130   END

```

You can now concentrate on each task separately. You might arrive at the following stage next:

```

100 'PROGRAM STRING REVERSAL
110   INPUT "ENTER A STRING "; A$
120   GOSUB 200 'Reverse and display
130   END

```

The last task of reversing and displaying the string is the heart of the problem. A subroutine that will accomplish this follows:

```

200 'SUB REVERSE AND DISPLAY
210   FOR X = LEN(A$) TO 1 STEP -1
220     PRINT MID$(A$,X,1);
230   NEXT X
240   RETURN

```

You can study the subroutine and see that the **MID\$** function chooses the last character in the string first and displays it. Then it chooses the next-to-last character, since **X** is decreased by one ( $-1$ ), and displays it. The characters are placed next to each other because of the semicolon at the end of the **PRINT** statement. The completed program is listed below:

```

100 'PROGRAM STRING REVERSAL
110   INPUT "ENTER A STRING "; A$
120   GOSUB 200 'Reverse and display
130   END

```

```

190 .'
200 'SUB REVERSE AND DISPLAY
210   FOR X = LEN(A$) TO 1 STEP -1
220     PRINT MID$(A$,X,1);
230   NEXT X
240   RETURN

```

You may wish to run this program using your own name as input.

### Example 3 - Word Count

The number of words in a phrase can be determined from the number of spaces (assuming that the only purpose of a space is to separate words). The task in this example is to compute and print the number of words in an input string. Working with paper and pencil, you might arrive at the following outline:

```

PROGRAM WORD COUNT
  Input phrase
  Count words
  Display word count
END

```

Since the approach to the solution of a problem is becoming more familiar, let's combine the skeleton program with the stage that produces the main routine. The following is one possible result:

```

100 'PROGRAM WORD COUNT
110   INPUT "ENTER PHRASE "; PHRASE$
120   GOSUB 200 'Count words
130   PRINT "WORD COUNT = "; COUNT
140   END

```

All that remains is the subroutine that counts the words. This is the toughest part of the problem and requires some thought. A possible subroutine follows:

```

200 'SUB COUNT WORDS
210   LET COUNT = 1
220   FOR K = 1 TO LEN(PHRASE$)
230     IF MID$(PHRASE$, K, 1) = " " THEN 240 ELSE 260
240     'THEN CLAUSE
250     LET COUNT = COUNT + 1
260   'END IF
270   NEXT K
280   RETURN

```

In the **FOR/NEXT** loop in the subroutine, the **MID\$** function is used to search for a blank in the phrase. When one is found, the computer branches to 240 and increases **COUNT** by 1. This counts the blanks and, thus, the words. Can you see why **COUNT** is assigned the value 1 to begin with?

The final program is listed below:

```

100 'PROGRAM WORD COUNT
110   INPUT "ENTER PHRASE "; PHRASE$
120   GOSUB 200 'Count words
130   PRINT "WORD COUNT = "; COUNT
140   END
190 '
200 'SUB COUNT WORDS
210   LET COUNT = 1
220   FOR K = 1 TO LEN(PHRASE$)
230     IF MID$(PHRASE$, K, 1) = " " THEN 240 ELSE 260
240     'THEN CLAUSE
250       LET COUNT = COUNT + 1
260     'END IF
270   NEXT K
280 RETURN

```

Enter the program and verify that it works correctly by typing in a sentence without commas in response to the input prompt.

#### Example 4 - Graphbox

The problem is to draw a box of a specified size at a given location. A possible outline for the main routine follows:

```

PROGRAM GRAPHBOX
  Enter position
  Determine scaling
  Draw box at position
END

```

Again, combining the program skeleton with the main routine, you might arrive at the following:

```

100 'PROGRAM GRAPHBOX
110   INPUT "ENTER POSITION NUMBERS "; X, Y
120   GOSUB 200 'Determine scaling
130   GOSUB 300 'Draw box at position
140   END

```

Focusing on the scaling subroutine, you might get

```

200 'SUB DETERMINE SCALING
210   INPUT "ENTER SCALE FACTOR "; S
220   LET BOX$ = " U = S; R = S; D = S; L = S;"
230   RETURN

```

**BOX\$** is assigned a string of **DRAW** substatements that can draw a box with side **S**. This string variable will be used in drawing the box to scale **S**.

The final subroutine draws the scaled box at the position (**X,Y**). This can be accomplished as follows:

```

300 'SUB DRAW BOX AT POSITION
310   DEF FN DOBOX$(X,Y) = "BM" + STR$(X) + ", "
      + STR$(Y) + BOX$
320   CLS
330   SCREEN 1
340   DRAW FN DOBOX$(X,Y)
350   RETURN

```

In line 310, the string function **DOBOX\$** is defined. The value of the function is the combination of a string that does a blank move to the position (X,Y) and the string variable **BOX\$** assigned in the previous subroutine. The function **DOBOX\$** is then used in line 340 to draw the desired box.

The complete program follows:

```

100 'PROGRAM GRAPHBOX
110   INPUT "ENTER POSITION NUMBERS "; X, Y
120   GOSUB 200 'Determine scaling
130   GOSUB 300 'Draw box at position
140   END
190 '
200 'SUB DETERMINE SCALING
210   INPUT "ENTER SCALE FACTOR "; S
220   LET BOX$ = "U = S; R = S; D = S; L = S;"
230   RETURN
290 '
300 'SUB DRAW BOX AT POSITION
310   DEF FN DOBOX$(X,Y) = "BM" + STR$(X) + ", "
      + STR$(Y) + BOX$
320   CLS
330   SCREEN 1
340   DRAW FN DOBOX$(X,Y)
350   RETURN

```

This problem has many other solutions using other graphics statements. This solution demonstrates the use of a string function. You may wish to run this program to see how it works.

## 8—5 PROBLEMS

1. What will be displayed if you run the following program?

```

100 'PROGRAM PROB 1
110   DEF FN A(X) = 2 + X
120   DEF FN B(Y) = 10 * Y
130   DEF FN C(Z) = Z ^ 2
140   LET R = 2
150   LET S = 3
160   LET T = 5

```

```

170 PRINT FN C(T), FN A(S), FN B(R)
180 LET S = S + T
190 PRINT FN A(R) + FN B(S) + FN C(T)
200 END

```

2. What will be displayed if you run the following program?

```

100 'PROGRAM PROB 2
110 DEF FN X(A) = 6 * A
120 DEF FN Y(B) = B + 10
130 DEF FN Z(C) = C ^ 3
140 READ P, Q, R
150 DATA 1, 2, 3
160 PRINT FN X(R); FN Z(P); FN Y(Q)
170 PRINT FN Y(P+Q) + FN X(R)
180 END

```

3. The area of a circle is pi times R squared, and the volume of a sphere is  $4/3$  times pi times R cubed. Pi is 3.14159, and R is either the radius of the circle or the radius of the sphere. Define two DEF statements, one to find the area of a circle and one to find the volume of the sphere. Set up a FOR/NEXT loop for R from 1 to 10 in steps of 0.5. Use the defined functions to print out a table of areas and volumes for each of the values of R.
4. Study the following program. What does it output?

```

100 'PROGRAM PROB 4
110 FOR K = 1 TO 5
120 READ A
130 LET B = INT(A) - SGN(A) * 2
140 PRINT B
150 NEXT K
160 DATA 2, 2, -3, 10, 0, -1.5
170 END

```

5. Explain what the following program does.

```

100 'PROGRAM PROB 5
110 FOR X = 1 TO 5
120 READ Y
130 LET Z = INT(100 * Y + .5) / 100
140 PRINT Z
150 NEXT X
160 DATA 1.06142, 27.5292, 138.021
170 DATA .423715, 51.9132
180 END

```



6. The following program won't work. What's wrong?

```

100 'PROGRAM WHAT ERROR
110   FOR X = -10 TO +10 STEP 2
120     PRINT X, SQR(X)
130   NEXT X
140 END

```

7. Write a program that calls for the input of a string and then prints the characters of the string in a column.
8. Write a program that calls for the input of a string and then prints the string diagonally so that each character is one line below and one character to the right of the previous character.
9. Write a program that counts the number of vowels in an input string.
10. Write a program that calls for the input of a string and then prints the words in the string in a column.
11. Write a program that asks the user to input a sentence. Use this sentence to generate a new string with all the spaces removed. Then print the new string.
12. Write a program that calls for the input of a string in uppercase letters, converts all the letters to lowercase, and prints the string back. You may need to refer to the ASCII character set in your computer reference manual.
13. Assume that five sentences are to be entered one at a time. Write a program that counts the number of times the word *the* appears in the five sentences.
14. If in response to the input prompt, you enter the string **ABCDEFGH**, what will the following program display?

```

100 'PROGRAM PROB 14
110   INPUT "ENTER A STRING "; A$
120   FOR K = 1 TO LEN(A$) STEP 2
130     PRINT MID$(A$, K, 1);
140   NEXT K
150 END

```

15. Write a program that calls for the input of a string and counts the number of times the character **I** is followed by the character **N**.
16. Count how frequently each of the 26 letters of the alphabet (you may assume that they are all uppercase) occur in ten sentences you enter. Do not count spaces or punctuation marks. Write a program to compute and print a table consisting of each of the letters and the number of times it occurred in the sentences. Do you think such a table could help you identify an author?

**8—6 PRACTICE TEST**

1. Fill in the blanks.

- a.  $\text{SQR}(36) = \underline{\hspace{2cm}}$
- b.  $\text{INT}(7.13) = \underline{\hspace{2cm}}$
- c.  $\text{ABS}(-22.8) = \underline{\hspace{2cm}}$
- d.  $\text{SGN}(-1.3) = \underline{\hspace{2cm}}$

2. How are string variables identified in BASICA?

\_\_\_\_\_

3. If  $\text{A\$} = \text{"HOW NOW BROWN COW"}\text{"}$ , write a function that will extract **NOW BROWN**.

\_\_\_\_\_

4. Write a program that calls for the input of a string and then prints the string again and again, deleting one character each time, until nothing is left. If, for example, you typed in **PIECE OF CAKE**, the computer should print out

```
PIECE OF CAKE
PIECE OF CAK
PIECE OF CA
PIECE OF C
PIECE OF
PIECE O
PIECE
PIEC
PIE
PI
P
```



# LISTS OF DATA

## 9—1 OBJECTIVES

### Using Single- and Double-Subscripted Variables

You will learn what subscripted variables are and how to use them.

### Saving Space for Arrays

Before you enter a collection of numbers into the computer, the computer must know how large the collection will be. You will learn to use the **DIM** statement to define the size of the collection.

### Using Subscripted Variables and **FOR/NEXT** Loops

When you use a collection of numbers in a program, that program will almost certainly call for repetition. You will discover how **FOR/NEXT** loops can help you handle the collection of numbers.

### Working with Program Examples

You will study BASIC programs that take advantage of the power of subscripted variables.

## 9—2 DISCOVERY EXERCISES

Since you will work with collections of numbers, you need to add two important words to your computer vocabulary. Although you could use the word *collection* to describe a group of numbers, two other words, *matrix* and *array*, are more commonly used. In this book, they both mean the same thing: a “collection of numbers.”

When you work with arrays, you need to distinguish one number in the array from the other numbers. Subscripts serve this purpose.

To see how this works, look at the array below:

$$Y_1 = 9$$

$$Y_2 = 10$$

$$Y_3 = 7$$

$$Y_4 = 14$$

$$Y_5 = 12$$

$$Y_6 = 15$$

The name of the array is Y. Its size is six, since there are six elements (or numbers) in it. The numbers 9, 10, 7, 14, 12, and 15 are the elements in the array. The numbers printed to the right and slightly below the Ys are called *subscripts*. Each subscript merely points to one element in the array. In BASICA, subscripts are written in parentheses. Thus, Y(4) means the fourth number in the array, which in this case is 14. Y(4) is read as "Y sub four," the third number in the array is "Y sub three," and so on. This array is one-dimensional, since it takes only a single subscript to locate a given element in the array.

Now look at a more complicated example.

$$Z_{1,1} = 4 \quad Z_{1,2} = 9 \quad Z_{1,3} = 5$$

$$Z_{2,1} = 3 \quad Z_{2,2} = 8 \quad Z_{2,3} = 7$$

In this example, there are six elements in the array Z. However, this array is two-dimensional, since you must specify an element in the array both by row and by column. The first subscript gives the row number; the second, the column. Z(2,1) is read as "Z sub two one" and means the element of Z at the second row and first column. Likewise, the element at row 1, column 3 would be identified as Z(1,3) ("Z sub one three").

In summary, you will work with two kinds of matrices or arrays. You need only one subscript to locate an element in a one-dimensional array, but two subscripts (a row number and a column number) to locate an element in a two-dimensional array.

## ■ Matrix and array mean collections of numbers.

### One Dimensional Arrays

1. Turn on the computer and bring up BASICA. Enter the following program:



```
100 'PROGRAM ONE DIMENSIONAL ARRAYS
110 LET X(1) = 21
120 LET X(2) = 13
130 LET X(3) = 16
140 LET X(4) = 8
150 LET X(5) = 11
160 PRINT X(1)
170 END
```

What do you think will be displayed if you run the program?

---

Run the program and record what happened.

---

2. Now modify the program to display the fourth value of **X**. Run the program. How does the answer displayed compare with what you thought it should be?

---

3. Edit line 160 as follows:

```
160 PRINT X(3) + X(4)
```

Display the program and study it briefly. What do you think will happen if you run the program?

---

Run the program and see if you were right. Record below what happened.

---

4. Edit line 160 and add lines 162 and 164 as follows:

```
160 FOR K = 1 TO 5
162 PRINT X(K)
164 NEXT K
```

Display the program. What do you think this program instructs the computer to print?

---

See if you were right. Record below what happened when you ran the program.

---

5. Modify the program to print out only the first three values of the array **X**. Record below what happened when you tried this.
- 

6. Again modify the program in line 160, but this time so that the first value of the array, then every other one, is printed. What do you need to add to the **FOR** statement?
- 

### Two-Dimensional Arrays

7. Clear the memory and enter the following program:

```

100 'PROGRAM TWO DIMENSIONAL ARRAYS
110   LET Y(1,1) = 2
120   LET Y(1,2) = 5
130   LET Y(1,3) = 1
140   LET Y(2,1) = 2
150   LET Y(2,2) = 4
160   LET Y(2,3) = 3
170   PRINT Y(1,3)
180   END

```

Display the program and make sure you have entered it correctly. What do you think this program does?

---

Run the program and record what appeared on the screen.

---

8. Edit line 170 as follows:

```

170   PRINT Y(2,2) + Y(1,3) + Y(1,1)

```

Display the program. What does this program do?

---

Run the program and see if you were right.

9. Edit line 170 and add lines 172-178 as follows:

```

170   LET S = 0
172   FOR J = 1 TO 3
174       LET S = S + Y(1,J)
176   NEXT J
178   PRINT S

```

Display the program and study it carefully. What will happen if you run this program? Run the program and record what was displayed.

---

Explain in your own words what is taking place in the program.

---

10. Edit lines 172 and 174 as follows:

```
172    FOR J = 1 TO 2
174      LET S = S + Y(J,2)
```

Display the program. What is the program doing now?

---

Run the program and write down what was displayed.

---

Again, explain in your own words what is happening.

---

11. Now change lines 174-180 and add lines 182 and 190 as follows:

```
174      FOR K = 1 TO 3
176        LET S = S + Y(J,K)
178      NEXT K
180    NEXT J
182    PRINT S
190  END
```

Display the program and think a minute about it. In particular, compare what you see now to what was going on in steps 9 and 10. What does this program do?

---

Run the program and record what was displayed.

---

**The DIM Statement**

12. Clear the memory and enter the following program:

```

100 'PROGRAM THE DIM STATEMENT .
110   DIM X(12), Y(12)
120   GOSUB 200 'What does it do?
130   PRINT X(1) + Y(4)
140   END
190 '
200 'SUB WHAT DOES IT DO?
205   FOR K = 1 TO 12
210     READ X(K), Y(K)
215   NEXT K
220   DATA 2, 1
225   DATA -1, 3
230   DATA 5, 6
235   DATA 2, 4
240   DATA 3, 1
245   DATA 8, 4
250   DATA 5, 1
255   DATA 3, 4
260   DATA 6, 2
265   DATA 1, 1
270   DATA 7, 7
275   DATA 5, 3
280   RETURN

```

Display the program and check to see that you have entered it correctly. Study the program carefully. If you run the program, what will be displayed?

---

Run the program and see whether or not you were right. Record below what was displayed.

---

13. Enter

```

110

```

Now display the first several lines of the program by typing **LIST 100-140**. What has happened?

---

Run the program and record what happened.

---

What can you say about the necessity of the **DIM** statement that was originally in the program.

---

14. Insert line 110 and edit line 205 as follows:

```
110    DIM X(9) , Y(9)
205    FOR K = 1 TO 9
```

Display the program. What will happen if you run the program?

---

Try it and see if you were correct.

15. Enter

```
110
```

Will the program work now that the **DIM** statement has been taken out?

---

Try it and record what was displayed.

---

Compare the results of step 13 with those of step 15. Sometimes the **DIM** statement must be present and other times it need not be. You will return to this question later.

16. Clear the memory and enter the following program:

```
100 'PROGRAM ARRAYS
110   DIM ARRAY(4,3)
120   GOSUB 200 'Study it
130   GOSUB 300 'Study again
140   END
190 '
200 'SUB STUDY IT
205   FOR J = 1 TO 4
210     FOR K = 1 TO 3
215       READ ARRAY(J,K)
220     NEXT K
225   NEXT J
230   DATA 1, 3, 1
235   DATA 4, 2, 5
240   DATA 1, 4, 2
245   DATA 3, 2, 5
250   RETURN
290 '
```



```

300 'SUB STUDY AGAIN
310   FOR J = 1 TO 4
320     FOR K = 1 TO 3
330       PRINT ARRAY(J,K) ,
340     NEXT K
350   PRINT
360   PRINT
370 NEXT J
380 RETURN

```

Make sure you have entered the program correctly, then take a few minutes to study it. Can you see what will be displayed if you run the program?

---

Run the program and record the output.

---

17. Clear the memory and then enter the following program:

```

100 'PROGRAM ARRAYS AGAIN
110   DIM ARRAY(2,2)
120   GOSUB 200 'Study it
130   GOSUB 300 'Study again
140 END
190 '
200 'SUB STUDY IT
210   FOR K = 1 TO 2
220     INPUT ARRAY(K,1) , ARRAY(K,2)
230   NEXT K
240 RETURN
290 '
300 'SUB STUDY AGAIN
310   FOR J = 1 TO 2
320     FOR K = 1 TO 2
330       PRINT ARRAY(J,K) ,
340     NEXT K
350   PRINT
360   PRINT
370 NEXT J
380 RETURN

```

Run the program. At the input prompts, enter the following lines, one at a time:

```

2, 5
3, 8

```

What happened?

---

Compare the output to the numbers you entered.

18. This concludes the computer work for this chapter. Turn off the computer and go on to the next section.

### 9—3 DISCUSSION

It is common to be a bit confused about arrays at this point. The discussion material should clear up any questions that might have arisen in the computer work.

#### Using Single- and Double-Subscripted Variables

The need for subscripted numbers becomes obvious when you handle large collections of numbers. If, for example, you wrote a program that involved only 4 numbers, you would have no difficulty naming them. You might call the numbers X, Y, U, and V. But suppose you needed to work with 100 numbers? For this reason, it is often very useful to have subscripted variables. Fortunately, BASICA lets you use subscripts to name elements.

Consider the following set of numbers.

$i$	$Y_i$
1	14
2	8
3	9
4	11
5	16
6	20
7	5
8	3

You can refer to the entire set of numbers with the single name Y. Thus, Y is a collection of numbers, a matrix, or an array—all of which mean roughly the same thing, for the purpose of this book. To locate a number in an array, you must have the array name (in this case Y) and the number's position within the array. The  $i$  column gives a number's position. In BASIC, the subscript is written enclosed in parentheses just after the name of the array. Thus, the BASIC variable  $Y(3)$  ("Y sub three") names the third number in the array Y. In this case,  $Y(3)$  has the value 9. Similarly,  $Y(7)$  is 5,  $Y(1)$  is 14, and so on. You can assign the general name  $Y(I)$  ("Y sub I") to an as yet unspecified number in the array.  $Y(I)$  denotes any element of the array depending on the value of I. In the example above, if I were 8, then the value of  $Y(I)$  would be 3. This collection of numbers is one dimensional, and you need only one number (subscript) to locate any element in the array.

Now look at a two-dimensional array.

$Y_{i,j}$	1	2	3	4
1	3	-1	10	8
2	2	4	5	6
3	1	-2	9	3

You need two numbers to locate an element in this array. Given a row number and a column number, you can find any element of the array. For example,  $Y(1,3)$  is the BASIC name of the element of Y located at row 1, column 3. In the example above, the element has the value 10. A general way to denote an element in a two-dimensional array is  $Y(I,J)$ . The first subscript (I) is the row number, and the second subscript (J) is the column number.

To make sure you understand how the double subscripts are used, refer to the

two-dimensional array in the table above and verify that the following statements are correct.

```

Y(3,2) = -2
Y(1,4) = 8
Y(3,3) = 9
Y(2,1) = 2

```

The computer can interpret subscripts in the form of expressions. Thus,  $X(M - N + 3, S * T)$  is legal provided that  $M - N + 3$  and  $S * T$  can be converted by the computer into numbers. Even  $Y(Y(1,1), Y(2,3))$  is all right as long as the computer can locate the numbers in  $Y(1,1)$  and  $Y(2,3)$ . However, suppose you specify a number in an array as  $X(A + B)$ , and the computer searches the memory, and suppose  $A$  has the value 2.6 and  $B$  has the value 1.1. Thus,  $A + B = 3.7$ . But it doesn't make any sense to try to look up the 3.7th number in the array  $X$ . Accordingly, the computer will take the integer part of 3.7 (or 3) and interprets  $X(A + B)$  as  $X(3)$ , the third element in the array  $X$ .

- Double subscripts define row and column numbers.

### Saving Space for Arrays

The computer must know how big an array is for two reasons. First, it must know how much space to save in memory to hold the array. Second, the computer must know the size of the array to carry out operations properly. Actually, BASICA saves space automatically for small arrays. When you use a one-dimensional array in a program without a **DIM** statement, the maximum value of the subscript is 10. If you use a two-dimensional array without a **DIM** statement, BASICA automatically saves enough space so that both subscripts can have a maximum value of 10. Even so, it is wise to use dimension statements in all programs regardless of the size of the arrays.

- Save space with a **DIM** statement.

Once you have dimensioned an array with a **DIM** statement, the subscript can take on any value from 0 up to the number in the **DIM** statement. Negative numbers are not allowed for subscripts. While you can use 0 as a subscript, it tends to be somewhat confusing. Consult the reference manual for further information.

An example of a **DIM** (for "dimension") statement is

```
100 DIM B(5,20), Y(8), Z(34), X$(3,6)
```

Four arrays are dimensioned in line 100. **B** is a two-dimensional numeric array having 5 rows and 20 columns. **Y** is a one-dimensional numeric array with 8 elements. **Z** is one-dimensional numeric array with 34 elements. Finally, **X\$** is a two-dimensional string array with 3 rows and 6 columns. It's a good practice to make the **DIM** statement the first one in the body of the main routine. That way you need only glance at the top of the main routine to see the sizes of the arrays that will be used. The **DIM** statement *must* precede any statements that refer to arrays.

## Using Subscripted Variables and FOR/NEXT Loops

Since subscripts locate numbers in a collection, and since operations with collections of numbers almost always involve repetition, it is reasonable to use **FOR/NEXT** statements to handle arrays. You can use **FOR/NEXT** loops to define the subscripts used in the arrays. For example, the following program segment will set up a six-by-four array, then load 5s into all the elements.

```

100 DIM A(6,4)
110 FOR R = 1 TO 6
120   FOR C = 1 TO 4
130     LET A(R,C) = 5
140   NEXT C
150 NEXT R

```

The details of the process become clear when you study this program segment. When line 130 in the program is reached the first time, **R** = 1 and **C** = 1. Then **R** is held constant while **C** goes to 2, 3, and 4. At each step in this process, the corresponding element of the array is set equal to 5. Then **R** is set equal to 2, and **C** takes on the values 1, 2, 3, and 4. The process goes on until all the elements of the array have been set equal to 5.

You can use subscripts in this manner to handle either one- or two-dimensional arrays. In most applications, **FOR/NEXT** loops are the best tools to carry out operations on arrays.

## 9—4 WORKING WITH PROGRAM EXAMPLES

### Example 1 - Examination Grades

To illustrate the concept of a one-dimensional array, let's use a set of examination grades. The following are the results of an examination given to a class of fifteen students:

	Student Number														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Grade	67	82	94	75	48	64	89	91	74	71	65	83	72	69	72

The problem is to write a BASIC program that allows you to enter the class grades. The format is:

```

HOW MANY STUDENTS? (You enter the number)
STUDENT GRADE
1 (You enter first grade, etc.)
2
3
(etc.)

```

The program should compute the class average, find the highest grade and lowest grade, and print this information as follows:

```

CLASS AVERAGE IS (Computer prints average)
HIGHEST GRADE IS (Computer prints highest grade)

```



**LOWEST GRADE IS** (*Computer prints lowest grade*)

An outline of the program might be as follows:

```

PROGRAM EXAM GRADES
  Dimension arrays
  Request number of students
  Print headers
  Enter grades
  Compute high & low grades
  Display output
END

```

The main routine could be:

```

100 'PROGRAM EXAM GRADES
110   DIM G(50)
120   GOSUB 200 'Enter grades
130   GOSUB 400 'Compute average
140   GOSUB 600 'High & low grade
150   GOSUB 800 'Display output
160   END

```

The request for the student's grades is handled in the **ENTER GRADES** subroutine. An outline of the subroutine could be:

```

SUB ENTER GRADES

  Input the student's grades
  (Loop block)

RETURN

```

The finished subroutine follows:

```

200 'SUB ENTER GRADES
210   INPUT "HOW MANY STUDENTS "; N
220   PRINT
230   PRINT "STUDENT", "GRADE"
240   FOR K = 1 TO N
250     PRINT K,
260     INPUT G(K)
270   NEXT K
280   PRINT
290   RETURN

```

Computing an average is something you have done before and can be accomplished with the following subroutine:



```

400 'SUB COMPUTE AVERAGE
410   LET SUM = 0
420   FOR K = 1 TO N
430     LET SUM = SUM + G(K)
440   NEXT K
450   LET AVE = SUM / N
460   RETURN

```

Notice that the sum of the grades is accumulated in line 430, and, when all of them have been added, the average is obtained by dividing the sum by the total number of students (shown in line 450).

Next comes the task of determining the highest and lowest grade. This can be accomplished by searching through the array of grades and continually placing the highest and lowest grade in designated variables. An outline of the subroutine follows:

```

SUB HIGH & LOW GRADE
  Initialize high and low grade
  Search the grade array for the high & low grade
    (Loop block and case block)
RETURN

```

The following subroutine accomplishes this task:

```

600 'SUB HIGH & LOW GRADE
605   LET HIGH = G(1)
610   LET LOW = G(1)
615   FOR K = 2 TO N
620     'CASE
625       IF LOW > G(K) THEN LET LOW = G(K)
630       IF HIGH < G(K) THEN LET HIGH = G(K)
635     'END CASE
640   NEXT K
645   RETURN

```

Displaying the results is fairly easy and is incorporated in the final program. The complete program follows:

```

100 'PROGRAM EXAM GRADES
110   DIM G(50)
120   GOSUB 200 'Enter grades
130   GOSUB 400 'Compute average
140   GOSUB 600 'High & low grade
150   GOSUB 800 'Display output
160   END
190 '
200 'SUB ENTER GRADES
210   INPUT "HOW MANY STUDENTS "; N
220   PRINT
230   PRINT "STUDENT", "GRADE"
240   FOR K = 1 TO N

```

```

250     PRINT K,
260     INPUT G(K)
270     NEXT K
280     PRINT
290     RETURN
390 '
400 'SUB COMPUTE AVERAGE
410     LET SUM = 0
420     FOR K = 1 TO N
430         LET SUM = SUM + G(K)
440     NEXT K
450     LET AVE = SUM / N
460     RETURN
590 '
600 'SUB HIGH & LOW GRADE
605     LET HIGH = G(1)
610     LET LOW = G(1)
615     FOR K = 2 TO N
620         'CASE
625             IF LOW > G(K) THEN LET LOW = G(K)
630             IF HIGH < G(K) THEN LET HIGH = G(K)
635         'END CASE
640     NEXT K
645     RETURN
790 '
800 'SUB DISPLAY OUTPUT
810     PRINT
820     PRINT "CLASS AVERAGE IS "; AVE
830     PRINT "HIGHEST GRADE IS "; HIGH
840     PRINT "LOWEST GRADE IS "; LOW
850     RETURN

```

Run this program with the data given at the beginning of the example. Verify that the program works correctly

### Example 2 - Course Grades

You can easily extend the ideas in Example 1 to a two-dimensional array. Suppose a class has ten students, and the course grade is based on five examinations. Examination results for such a class might be:

		Student Number									
		1	2	3	4	5	6	7	8	9	10
Exam	1	92	71	81	52	75	97	100	63	41	75
	2	85	63	79	49	71	91	93	58	52	71
	3	89	74	80	61	79	88	97	55	51	73
	4	96	68	84	58	80	93	95	61	47	70
	5	82	72	82	63	73	92	93	68	86	74

The desired output is as follows:

```

STUDENT          COURSE AVE
1                (Computer prints average, etc.)
2
3
etc.

```

```

TEST            CLASS AVE
1              (Computer prints average, etc.)
2
3
etc.

```

With this in mind, you can give some thought to the solution and perhaps come up with the following outline:

```

PROGRAM COURSE GRADES
  Dimension arrays
  Load test grades
  Compute and display student averages
  Compute and display class averages
END

```

The main routine follows nicely:

```

100 'PROGRAM COURSE GRADES
110   DIM G(5,10)
120   GOSUB 200 'Load test grades
130   GOSUB 400 'Compute & display student averages
140   GOSUB 600 'Compute & display class averages
150   END

```

You can load the array with the student's test scores by using a combination of the **READ/DATA** statements and a **FOR/NEXT** loop. The following subroutine accomplishes the task:

```

200 'SUB LOAD TEST GRADES
210   DATA 92,71,81,52,75,97,100,63,41,75
220   DATA 85,63,79,49,71,91,93,58,52,71
230   DATA 89,74,80,61,79,88,97,55,51,73
240   DATA 96,68,84,58,80,93,95,61,47,70
250   DATA 82,72,82,63,73,92,93,68,56,74
260   FOR R = 1 TO 5
270     FOR C = 1 TO 10
280       READ G(R,C)
290     NEXT C
300   NEXT R
310   RETURN

```

Again, computing averages is familiar. The complete program follows:

```

100 'PROGRAM COURSE GRADES
110   DIM G(5,10)
120   GOSUB 200 'Get data
130   GOSUB 400 'Compute and display student averages
140   GOSUB 600 'Compute and display class averages
150 END
190 '
200 'SUB LOAD TEST GRADES
210   DATA 92,71,81,52,75,97,100,63,41,75
220   DATA 85,63,79,49,71,91,93,58,52,71
230   DATA 89,74,80,61,79,88,97,55,51,73
240   DATA 96,68,84,58,80,93,95,61,47,70
250   DATA 82,72,82,63,73,92,93,68,56,74
260   FOR R = 1 TO 5
270     FOR C = 1 TO 10
280       READ G(R,C)
290     NEXT C
300   NEXT R
310 RETURN
390 '
400 'SUB STUDENT AVERAGES
410   PRINT "STUDENT", "COURSE AVE"
420   PRINT
430   FOR C = 1 TO 10
440     LET SUM = 0
450     FOR R = 1 TO 5
460       LET SUM = SUM + G(R,C)
470     NEXT R
480     PRINT C, SUM / 5
490   NEXT C
500   PRINT
510   PRINT
520 RETURN
590 '
600 'SUB CLASS AVERAGES
610   PRINT "TEST", "CLASS AVE"
620   PRINT
630   FOR R = 1 TO 5
640     LET SUM = 0
650     FOR C = 1 TO 10
660       LET SUM = SUM + G(R,C)
670     NEXT C
680     PRINT R, SUM / 10
690   NEXT R
700   PRINT
710   PRINT
720 RETURN

```

This program illustrates valuable programming techniques involving arrays. It is worth studying and executing on your computer.

### Example 3 - Array Operations

A series of short programs, presented without explanation, follow. Study each program until you are sure you understand what is taking place.

a. Write a program using FOR/NEXT loops to load a three-by-four array with 1s.

```

100 'PROGRAM LOAD ARRAY WITH ONES
110   DIM X(3,4)
120   FOR R = 1 TO 3
130     FOR C = 1 TO 4
140       LET X(R,C) = 1
150     NEXT C
160   NEXT R
170 END

```

b. Write a program to generate and load the numbers

2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048

into a one-dimensional array.

```

100 'PROGRAM LOAD ARRAY
110   DIM Z(11)
120   LET Z(1) = 2
130   FOR K = 2 TO 11
140     LET Z(K) = 2 * Z(K - 1)
150   NEXT K
160 END

```

c. Write a program to read in the array

$$\begin{pmatrix} 2 & 3 & 5 \\ 1 & 4 & 2 \end{pmatrix}$$

from DATA statements and then print out the array.

```

100 'PROGRAM READ & PRINT ARRAY
110   DIM A(2,3)
120   FOR R = 1 TO 2
130     FOR C = 1 TO 3
140       READ A(R,C)
150     NEXT C
160   NEXT R
170   FOR R = 1 TO 2
180     FOR C = 1 TO 3
190       PRINT A(R,C),
200     NEXT C

```



```

210     PRINT
220     PRINT
230     NEXT R
240     DATA 2, 3, 5
250     DATA 1, 4, 2
260     END

```

#### Example 4 - A Sales Chart as a String Array

The following is a chart of a retail store's sales of computers. The sales are given by salesperson for each of four quarters.

		Quarter			
Name	JACK	7	9	11	17
	FRED	5	10	15	20
	MARY	8	7	21	14

This table can be viewed as a string array of three rows and six columns. Your task is to write a program that asks for the name of a salesperson, computes the total unit sales for that person, and prints out the results. An outline of the program might be:

```

PROGRAM SALES USING STRINGS
  Dimension arrays
  Load data
  Request salesperson
  Compute & display sales
END

```

The main routine corresponding to this outline follows:

```

100 'PROGRAM SALES USING STRINGS
110   DIM UNITSALES$(3,5)
120   GOSUB 200 'Load data
130   GOSUB 400 'Request salesperson
140   GOSUB 600 'Compute & display sales
150   END

```

You can load the data as in Example 2. The subroutine to accomplish this follows:

```

200 'SUB LOAD DATA
210   FOR R = 1 TO 3
220     FOR C = 1 TO 5
230       READ UNITSALES$(R,C)
240     NEXT C
250   NEXT R
260   DATA JACK, 7, 9, 11, 17
270   DATA FRED, 5, 10, 15, 20
280   DATA MARY, 8, 7, 21, 14
290   RETURN

```

Notice that the numbers in array `UNITSALE$` are loaded in string form so that the names and numbers can be in the same array.

The `REQUEST SALESPERSON` subroutine should allow you to input the name of the person whose total unit sales you want displayed. The program then searches the first column of the array for a match, arriving at the row containing the desired information. The following subroutine accomplishes that task:

```

400 'SUB REQUEST SALESPERSON
410 'LOOP
420     LET FLAG = 0
430     INPUT "SALESPERSON'S NAME: "; N$
440     FOR K = 1 TO 3
450         IF UNITSALE$(K,1) = N$ THEN 460 ELSE 490
460         'THEN CLAUSE
470             LET FLAG = 1
480             LET J = K
490         'END IF
500     NEXT K
510     IF FLAG = 0 THEN 520 ELSE 540
520     'THEN CLAUSE
530         PRINT "NO SUCH NAME "; N$
540     'END IF
550     GOTO 410
560 'END LOOP
570 RETURN

```

Line 450 is the heart of the search for the name matching the one input. If the input name is not spelled correctly or is not the name of a salesperson, the value of `FLAG` is 0 when the computer leaves the `FOR/NEXT` loop. If `FLAG` is 0, the computer prints a message and loops back to line 410. If `FLAG` is 1, the input name matches one of the names in the array. The value of `J` identifies the proper row in the array.

Once you have a match, you can sum up the sales for that person and display the results. However, since the data in the array are string values, you must use the `VAL` function to convert the strings to numbers before you add them together. The following subroutine takes care of that:

```

600 'SUB COMPUTE & DISPLAY SALES
610     LET SUM = 0
620     FOR C = 2 TO 5
630         LET SUM = SUM + VAL(UNITSALE$(J,C))
640     NEXT C
650     PRINT N$; " SOLD"; SUM; " COMPUTERS"
660 RETURN

```

The complete program follows:

```

100 'PROGRAM SALES USING STRINGS
110     DIM UNITSALE$(3,5)
120     GOSUB 200 'Load data
130     GOSUB 400 'Request salesperson

```

```

140   GOSUB 600 'Compute & display sales
150   END
190   '
200   'SUB LOAD DATA
210     FOR R = 1 TO 3
220       FOR C = 1 TO 5
230         READ UNITSALE$(R,C)
240       NEXT C
250     NEXT R
260     DATA JACK, 7, 9, 11, 17
270     DATA FRED, 5, 10, 15, 20
280     DATA MARY, 8, 7, 21, 14
290   RETURN
390   '
400   'SUB REQUEST SALESPERSON
410     'LOOP
420       LET FLAG = 0
430       INPUT "SALESPERSON'S NAME: "; N$
440       FOR K = 1 TO 3
450         IF UNITSALE$(K,1) = N$ THEN 460 ELSE 490
460         'THEN CLAUSE
470           LET FLAG = 1
480           LET J = K
490         'END IF
500       NEXT K
510       IF FLAG = 0 THEN 520 ELSE 540
520       'THEN CLAUSE
530         PRINT "NO SUCH NAME "; N$
540       'END IF
550       GOTO 410
560     'END LOOP
570   RETURN
590   '
600   'SUB COMPUTE & DISPLAY SALES
610     LET SUM = 0
620     FOR C = 2 TO 5
630       LET SUM = SUM + VAL(UNITSALE$(J,C))
640     NEXT C
650     PRINT N$; " SOLD"; SUM; " COMPUTERS"
660   RETURN

```

There is a valuable lesson to be learned from this example. At times, It is very useful to use a string array to store data that you wish to examine. In this way, you can store numbers and strings in the same array since the numbers are stored as strings also. Whenever desired information requires arithmetic, you simply convert the numbers before doing the arithmetic.

## 9—5 PROBLEMS

1. Write a program which uses the following DATA statements:

```
250 DATA 12
260 DATA 2,1,4,3,2,4,5,6,3,5,4,1
```

The program should read the size of an array from the first DATA statement, then read the elements of the array from the second DATA statement, load the elements into a one-dimensional numeric array **X**, and print the array.

2. Write a program to fill a three-by-four array with 1s.
3. Write a program to call for the input of a square N-by-N matrix where N is a whole number no larger than 10. Then program the computer to calculate and print the sum of the entries on the main diagonal of the array.
4. Write a BASIC program using a **READ** statement to read 25 numbers from DATA statements into a one-dimensional array named **A**. Include instructions in your program to search the array and print the number of elements in the array that are greater than 50. Fill in the required DATA statements with any numbers you choose.
5. Write a program to call for the input of an M-by-N matrix. Assume that both M and N are no larger than 15. Then program the computer to calculate and print the sum of all the elements in the matrix.
6. The program below is supposed to compute and print out the sum of the elements in a one-dimensional array that are positive but not greater than 10. As it stands the program is incorrect. What's wrong?

```
100 'PROGRAM PROB 6
110 DIM A(6)
120 GOSUB 200 'Input data
130 GOSUB 300 'Add pos numbers < 10
140 END
190 '
200 'SUB INPUT DATA
210 FOR K = 1 TO 6
220 INPUT A(K)
230 NEXT K
240 RETURN
290 '
300 'SUB ADD POS NUMBERS < 10
310 FOR K = 6 TO 1 STEP -1
320 LET S = 0
330 IF A(K) > 0 AND A(K) <= 10 THEN 340 ELSE 360
340 'THEN CLAUSE
350 LET S = S + A(K)
360 'END IF
370 NEXT K
380 PRINT S
390 RETURN
```

7. What will be output if you run the following program?

```

100 'PROGRAM PROB 7
110   DIM Y(6)
120   GOSUB 200 'Load array
130   GOSUB 300 'You determine
140   LET X = S1 - S2
150   PRINT X
160   END
190 '
200 'SUB LOAD ARRAY
210   FOR K = 1 TO 6
220     READ Y(K)
230   NEXT K
240   DATA 2, 1, 3, 1, 2, 1
250   RETURN
290 '
300 'SUB YOU DETERMINE
310   LET S1 = 0
320   LET S2 = 0
330   FOR K = 1 TO 6
340     LET S1 = S1 + Y(K)
350     LET S2 = S2 + Y(K) ^ 2
360   NEXT K
370   RETURN

```

8. What will be output if you run the following program?

```

100 'PROGRAM PROB 8
110   DIM A(10)
120   GOSUB 200 'Load array
130   LET S = A(1)
140   GOSUB 300 'You determine
150   LET A(10) = S
160   GOSUB 400 'Display array
170   END
190 '
200 'SUB LOAD ARRAY
210   FOR K = 1 TO 10
220     READ A(K)
230   NEXT K
240   DATA 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
250   RETURN
290 '
300 'SUB YOU DETERMINE
310   FOR K = 1 TO 9
320     LET A(K) = A(K+1)
330   NEXT K
340   RETURN
390 '
400 'SUB DISPLAY ARRAY
410   FOR K = 1 TO 10

```



```

420     PRINT A(K)
430     NEXT K
440     RETURN

```

9. What will be printed if you run the following program?

```

100 'PROGRAM PROB 9
110   DIM X(4,4)
120   GOSUB 200 'Load array
130   LET S = 0
140   GOSUB 300 'You determine
150   PRINT S
160   END
190 '
200 'SUB LOAD ARRAY
210   FOR J = 1 TO 4
220     FOR K = 1 TO 4
230       READ X(J,K)
240     NEXT K
250   NEXT J
260   DATA 1, 2, 3, 4, 2, 3, 4, 5
270   DATA 3, 4, 5, 6, 4, 5, 6, 7
280   RETURN
290 '
300 'SUB YOU DETERMINE
310   FOR K = 1 TO 4
320     LET S = S + X(K,5-K)
330   NEXT K
340   RETURN

```

10. What will be printed if you run the following program?

```

100 'PROGRAM PROB 10
110   DIM Y(4,4)
120   GOSUB 200 'Load array
130   GOSUB 300 'You determine
140   GOSUB 400 'Display array
150   END
190 '
200 'SUB LOAD ARRAY
210   FOR R = 1 TO 4
220     FOR C = 1 TO 4
230       LET Y(R,C) = 0
240     NEXT C
250   NEXT R
260   RETURN
290 '
300 'SUB YOU DETERMINE
310   FOR R = 1 TO 4
320     FOR C = 1 TO 4 STEP 2

```

```

330      LET Y(R,C) = R * C
340      NEXT C
350      NEXT R
360      RETURN
390 '
400 'SUB DISPLAY ARRAY
410      FOR R = 1 TO 4
420          FOR C = 1 TO 4
430              PRINT Y(R,C);
440          NEXT C
450          PRINT
460      NEXT R
470      RETURN

```

11. Write a BASIC program that calls for the input of N (assumed to be a whole number between 1 and 100), then input a one-dimensional array with N elements. The program should instruct the computer to sort the array into descending order and finally print the sorted array.
12. Assume that the first number in the DATA statements gives the number of pieces of data to follow. Also assume that the pieces of data are all whole numbers between 1 and 10 inclusive. Write a program that will compute the number of 1s, 2s etc., in the data and then print the results. (Hint: Use the data as a subscript to increment an element of an array used to count the numbers.)
13. What will be printed if you run the following program?

```

100 'PROGRAM PROB 13
110      DIM Z(6,6)
120      GOSUB 200 'Load array
130      GOSUB 300 'You determine
140      GOSUB 400 'Display array
150      END
190 '
200 'SUB LOAD ARRAY
210      FOR R = 1 TO 6
220          FOR C = 1 TO 6
230              LET Z(R,C) = 0
240          NEXT C
250      NEXT R
260      RETURN
290 '
300 'SUB YOU DETERMINE
310      FOR R = 1 TO 5 STEP 2
320          FOR C = R TO 6
330              LET Z(R,C) = 1
340          NEXT C
350      NEXT R
360      RETURN
390 '
400 'SUB DISPLAY ARRAY

```

```

410   FOR R = 1 TO 6
420     FOR C = 1 TO 6
430       PRINT Z(R,C);
440     NEXT C
450   PRINT
460 NEXT R
470 RETURN

```

14. What will be output if you run the program below?

```

1000 'PROGRAM PROB 14
1010   DIM A(5,5)
1020   GOSUB 2000 'Load array
1030   GOSUB 3000 'You determine
1040   GOSUB 4000 'Display array
1050 END
1990 '
2000 'SUB LOAD ARRAY
2010   FOR R = 1 TO 5
2020     FOR C = 1 TO 5
2030       READ A(R,C)
2040     NEXT C
2050   NEXT R
2060   DATA 2, 2, 2, 2, 2, 2, 2, 2, 2, 2
2070   DATA 2, 2, 2, 2, 2, 2, 2, 2, 2, 2
2080   DATA 2, 2, 2, 2, 2
2090 RETURN
2990 '
3000 'SUB YOU DETERMINE
3010   FOR C = 5 TO 1 STEP -1
3020     FOR R = 1 TO C
3030       LET A(R,C) = 3
3040     NEXT R
3050   NEXT C
3060 RETURN
3990 '
4000 'SUB DISPLAY ARRAY
4010   FOR R = 1 TO 5
4020     FOR C = 1 TO 5
4030       PRINT A(R,C);
4040     NEXT C
4050   PRINT
4060 NEXT R
4070 RETURN

```

15. Write a program to read the following array from DATA statements, then print the array.

$$\begin{pmatrix} 2 & 1 & 0 & 5 & 1 \\ 3 & 2 & 1 & 3 & 1 \end{pmatrix}$$

16. Write a program to read the following array from DATA statements, then print the array.

$$\begin{pmatrix} 5 & 3 \\ 2 & 0 \\ -1 & 1 \\ 4 & 2 \\ 2 & 6 \end{pmatrix}$$

17. Write a BASIC program that will call for the input of an M-by-N array, then compute and print out the sum of the elements in each row and the product of the elements in each column.
18. Write a BASIC program that will read two two-by-three arrays from **DATA** statements, then compute a third two-by-three array such that each element is the sum of the corresponding elements in the first two arrays. Finally, have the computer print the third array.
19. The data below represent the totals of sales made by four salespersons during one week.

		Mon	Tue	Wed	Thu	Fri	Sat
	1	48	40	73	120	100	90
Sales	2	75	130	90	40	110	85
Person	3	50	72	140	125	106	92
	4	108	75	92	152	91	87

Write a program that will compute and print out

- a. The daily sales totals
  - b. The weekly sales totals for each salesperson
  - c. The total weekly sales
20. Write a program to call for the input of a four-by-four array, then compute a new array from the first with the rows and columns interchanged. That is, row 1 of the input array becomes column 1 of the new array. Row 2 of the input array becomes column 2 of the new array, and so on. Then have the computer print the new array.
  21. Consider the two arrays below.

P	X
1	28
5	2
3	14
6	3
4	17
2	9

Each element of **P** "points" to an element of **X**. **P**(1) = 1 and **X**(1) = 28. **P**(2) = 5 and **X**(5) = 17. If you continue this process, you can see that the values of **X** are listed in descending order. Write a program to set up two arrays **X**, and **P**, to some convenient length. Then call for the input of arbitrary values for the array **X** from the keyboard. Construct the array **P** so that its elements point to the array **X** in

descending order as illustrated above. Then use array **P** to print out the values of array **X** in descending order.

## 9—6 PRACTICE TEST

1. What is the purpose of the **DIM** statement?

---

2. In the BASIC array named **X**, what variable name in BASIC locates the element in row 3, column 4?

---

3. Use an array in a program to input a list of numbers, then find and print out the sum of the positive numbers in the list. The printout should look as follows.

HOW MANY NUMBERS? *(You type in the number)*

WHAT ARE THE NUMBERS? *(You type them in)*

THE SUM OF POSITIVE NUMBERS IS *(Computer types answer)*

---

4. Write a program using **FOR/NEXT** statements to load a four-by-six array with 4s.

---

5. What will be printed if you run the following program?

```

100 'PROGRAM PRACTICE TEST 5
110   DIM A(5,5)
120   GOSUB 200 'Initialize array
130   GOSUB 300 'You determine
140   GOSUB 400 'Display array
150   END
190 '
200 'SUB INITIALIZE ARRAY
210   FOR J = 1 TO 5
220     FOR K = 1 TO 5
230       LET A(J,K) = 0
240     NEXT K
250   NEXT J
260   RETURN

```



```

290 '
300 'SUB YOU DETERMINE
310   FOR K = 1 TO 5
320     LET A(K,K) = 2
330   NEXT K
340 RETURN
390 '
400 'SUB DISPLAY ARRAY
410   FOR J = 1 TO 5
420     FOR K = 1 TO 5
430       PRINT A(J,K);
440     NEXT K
450   PRINT
460 NEXT J
470 RETURN

```

---

6. The following array is named **A**.

$$\begin{pmatrix} 1 & 3 & 5 \\ 6 & 2 & 4 \end{pmatrix}$$

- a. Write a **DIM** statement for **A**.

---

- b. What is the value of **A(2,3)**?

---

- c. If **X = 1** and **Y = 2**, what is **A(X,Y)**?

---

- d. What is **A(A(1,1),A(2,2))**?

---

# SIMULATIONS WITH RANDOM NUMBERS

## 10—1 OBJECTIVES

### Understanding Random-Number Generators

The random-number generator function is the heart of all programs involving the element of chance or randomness. You will learn about these random-number generators.

### Designing Sets of Random Numbers

Generally, a random-number generator is used to produce sets of random numbers with characteristics specified by the programmer. You will see how to generate desired sets of random integers.

### Working with Program Examples

You will study programming examples that simulate games of chance and random events.

## 10—2 DISCOVERY EXERCISES

### Setting up the Random-Number Generator

Before beginning the computer work, you need to know some important characteristics of random-number generators. By their very nature, these generators produce sequences of numbers that appear to have no pattern or relationship. For a random-number generator to be useful, it must return a different sequence of numbers each time you run a program that uses it. Suppose, however, a program that uses random numbers is not working correctly. If the problem lies with the random numbers, it might be extremely difficult to correct since different random numbers are generated each time the program is run. As you saw in Chapter 8, BASICA allows a sequence of random numbers to be repeated each time the program is run. You should use this feature of BASICA only when you are troubleshooting a program.

1. Turn on the computer and bring up BASICA. Enter the following program:

```

100 'PROGRAM LARGEST & SMALLEST
110   RANDOMIZE(TIMER) 'Set seed
120   GOSUB 200 'Initialize variables
130   GOSUB 300 'Generate & find largest & smallest
140   GOSUB 400 'Display largest & smallest
150   END
190 '
200 'SUB INITIALIZE VARIABLES
210   LET L = .5
220   LET S = .5
230   RETURN
290 '
300 'SUB GENERATE & FIND LARGEST & SMALLEST
310   FOR K = 1 TO 100
320     LET X = RND
330     'CASE
340     IF X > L THEN LET L = X
350     IF X < S THEN LET S = X
360     'END CASE
370   NEXT K
380   RETURN
390 '
400 'SUB DISPLAY LARGEST & SMALLEST
410   PRINT "LARGEST = "; L
420   PRINT "SMALLEST = "; S
430   RETURN

```

This program examines all the numbers generated by the **RND** function and keeps track of the largest and smallest numbers generated. As the program stands, it will generate 100 random numbers. Run the program. Record what was displayed.

---

Recall that, in E notation, a negative integer following the E means "move the decimal to the left."

2. Edit line 310 as follows:

```

310   FOR K = 1 TO 1000

```

Now the program will generate 1000 random numbers. The program will take about 10 seconds to run. Run the program and record what was printed out.

---

From what you have seen so far, what do you estimate is the largest number the **RND** function will generate?

---

What about the smallest?

---

3. You can easily modify this program to find the average of the randomly generated numbers. Enter the following new lines:

```

225     LET SUM = 0
325     LET SUM = SUM + X
425     PRINT "AVERAGE = "; SUM / 1000

```

What do you estimate would be the average of a set of randomly generated numbers between 0 and 1?

---

Clear the screen and run the program. Write down the average.

---

Run the program again. How does the new average compare with your estimate and the average you wrote down?

---

4. Now go on to generating random integers. Clear the memory and enter the following program:

```

100 'PROGRAM RANDOM INTEGERS
110  RANDOMIZE(TIMER)
120  FOR K = 1 TO 10
130    PRINT INT(2 * RND)
140  NEXT K
150  END

```

Run the program and record the output.

---

What were the only two numbers in the output?

---

5. Edit line 130 as follows:

```
130      PRINT INT(3 * RND)
```

Display the program. If this program is run, what numbers do you think will be displayed?

---

Run the program several times. Can you predict anything about the sequence or pattern in which the numbers will appear?

---

6. Edit line 130 as follows:

```
130      PRINT INT(2 * RND) + 1
```

What do you think the program will do now?

---

Run the program and record the output.

---

7. Edit line 130 as follows:

```
130      PRINT INT(4 * RND) + 4
```

If the program is run, what do you think the computer will display?

---

Run the program and describe the output.

---

Is there any pattern to the output?

---

8. Now let's look at how well the random integers between 1 and 10 are distributed. Clear the memory and screen and enter the following program:



```

100 'PROGRAM DISTRIBUTION
110   DIM X(10)
120   RANDOMIZE(TIMER)
130   GOSUB 200 'Initialize counter variables
140   GOSUB 300 'Generate & count
150   GOSUB 400 'Display distribution
160   END
190 '
200 'SUB INITIALIZE COUNTER VARIABLES
210   FOR K = 1 TO 10
220     LET X(K) = 0
230   NEXT K
240   RETURN
290 '
300 'SUB GENERATE & COUNT
310   FOR K = 1 TO 1000
320     LET Y = INT(10 * RND) + 1
330     LET X(Y) = X(Y) + 1
340   NEXT K
350   RETURN
390 '
400 'SUB DISPLAY DISTRIBUTION
410   FOR K = 1 TO 10
420     PRINT K, X(K)
430   NEXT K
440   RETURN

```

How many random numbers are generated by the **GENERATE & COUNT** subroutine?

---

9. Clear the screen and run the program. The program will take about 12 seconds to run. Are there more 2s than 3s?
- 

Run the program again. How does the distribution displayed this time compare with the previous one?

---

How evenly are the numbers distributed in the two distributions displayed?

---

If you decrease the number of integers generated in the **GENERATE & COUNT** subroutine, the chances for even distribution of integers diminishes. On the other hand, if you generate more random numbers, the probability of getting a nearly even distribution increases.

10. Clear the memory and enter the following simulation program:

```

100 'PROGRAM COIN TOSSING
110  RANDOMIZE(TIMER)
120  PRINT "TOSS", "OUTCOME"
130  GOSUB 200 'Heads or tails?
140  END
190 '
200 'SUB HEADS OR TAILS?
210  FOR K = 1 TO 10
220    LET FLIP = INT(2 * RND)
230    'CASE
240      IF FLIP = 0 THEN PRINT K, "T"
250      IF FLIP = 1 THEN PRINT K, "H"
260    'END CASE
270  NEXT K
280  RETURN

```

Study the program briefly. What are the possible random numbers that can be assigned to **FLIP** in line 220?

---

What random number represents a head in this simulation?

---

Run the program. How many heads were tossed?

---

Run the program again. How many heads were tossed this time?

---

11. Delete line 120 and edit line 110 as follows:

```

110  RANDOMIZE(751)

```

Run the program several times. How do the lists from each run compare?

---

12. Edit line 110 as follows:

```
110 RANDOMIZE(752)
```

Run the program. How do the results in this run compare with the results in step 11?

---

Run the program several more times. How do the lists from each run compare?

---

13. Turn off the computer and go on to the discussion.

## 10—3 DISCUSSION

### Understanding Random-Number Generators

You need not be concerned with the details of how random numbers are generated. It is enough to say that several mathematical methods can produce these numbers. The random-number generator is called on by the **RND** function. This function is used like other built-in functions in BASIC but differs in one important respect. The difference is that there seems to be no pattern or rule used to generate these numbers. Of course, this is precisely the point of the function. **RND** generates numbers between 0 and 1 at random. All the numbers in the interval have an equal chance of showing up. (Actually, the range of numbers generated is from 0.0000000 to 0.9999999. Zero can show up occasionally, but the number 1 never occurs.)

As you saw in the chapter on functions, the **RANDOMIZE** statement is used to seed the random number generator. For example,

```
RANDOMIZE(TIMER)
```

uses the number of seconds on the internal clock in the computer to seed the random number generator. If you wish to repeat the same set of random numbers each time you run a program, then you would use a fixed number in the **RANDOMIZE** statement. For example,

```
RANDOMIZE(62)
```

seeds the random number generator the same way each time it is performed.

- **RND generates numbers in the range 0 to 0.9999999.**

### Designing Sets of Random Numbers

Most often you do not want random numbers in the range produced by the **RND** function. You might want random integers in a certain range or a set of random numbers with a particular set of characteristics. Therefore, you need to know how to generate sets of random numbers with characteristics you can specify.

Let's begin with the characteristics of the **RND** function. **RND** delivers numbers in the range 0 to 1, that is, from 0 to slightly less than 1. If you multiply **RND** by **N**, you multiply the range of the function by **N**. Thus, **N \* RND** will produce random numbers in the range 0 to slightly less than **N**. If desired, you could shift the numbers (keeping the same range) by adding a number. **(N \* RND) + A** would produce random numbers from **A** to slightly less than **(A + N)**. Finally, you could use the **INT** function to take the integer of an expression and in this way generate random integers. The examples below indicate how you might use the **RND** function.

BASIC Expression	Result
<b>(5 * RND) + 10</b>	Random numbers in the range 10 to 14.99999
<b>INT(5 * RND) + 10</b>	Random integers 10, 11, 12, 13, 14
<b>INT(2 * RND) + 1</b>	Random integers 1, 2
<b>100 * RND</b>	Random numbers in the range 0 to 99.99999

You may have encountered the notion of mean and standard deviation (see problem 11 in this Chapter). You can use the **RND** function to generate numbers that appear to be drawn from a collection of numbers having a given mean and standard deviation. The rule for generating these numbers is

$$X = M + S * ((\text{sum of 12 numbers from RND function}) - 6)$$

where **M** and **S** are the desired mean and standard deviation, respectively. As defined above, the values of **X** will appear to belong to a collection of numbers with mean **M** and standard deviation **S**. The values of **X** can be used to simulate a process following a bell curve.

A note on troubleshooting: Recall that in BASICA, there is a way to run a program several times and repeat the sequence of random numbers generated by the **RND** function (see steps 11 and 12 of the Discovery Exercises). It is a good practice to write programs initially so that they will generate the same sequence of random numbers each time you run them. Once you are sure that the program is working correctly, you can modify the program to generate truly random numbers.

## 10—4 WORKING WITH PROGRAM EXAMPLES

### Example 1 - Dice Rolling Simulation

The game of craps is played with 2 dice. You roll the dice and observe the sum. The rules of the game state that if, on the first roll, you roll a 7 or 11, you win; if you roll a 2, 3, or 12, you lose. Otherwise, the sum you rolled is called your point, and you must roll this point before you roll a 7 in order to win. If you roll a 7 before you roll your point, then you lose. The possible outcomes of the first roll are indicated in the following table:

Sum on First Roll	Outcome
7, 11	You win
2, 3, 12	You lose
4, 5, 6, 8, 9, 10	Roll again

## Main Routine

The pencil-and-paper outline of the main routine might look like this:

```

PROGRAM DICE ROLLING SIMULATION
  Seed random number generator
  Roll dice and display sum
  Win, lose, or roll again
END

```

The main routine follows easily from this outline. Here is how it might look:

```

1000 'PROGRAM DICE ROLLING SIMULATION
1010   RANDOMIZE(TIMER)
1020   GOSUB 2000 'Roll dice & display sum
1030   GOSUB 3000 'Win, lose, or roll again
1040   END

```

## Roll Dice Subroutine

You are now in a position to tackle the two subroutines in the main routine. The first subroutine to roll the dice and display the sum requires the use of the **RND** function. You will need to produce random numbers between 1 and 6 for each one of the two dice and then find their sum. An action block that could accomplish this task follows:

```

2000 'SUB ROLL DICE & DISPLAY SUM
2010   LET DIE1 = INT(6 * RND) + 1
2020   LET DIE2 = INT(6 * RND) + 1
2030   LET SUM = DIE1 + DIE2
2040   PRINT "YOUR ROLL IS "; SUM
2050   RETURN

```

## Win-Lose Subroutine

The second subroutine is more complicated. Here is the task it must accomplish and the appropriate structure:

```

SUB WIN, LOSE, OR ROLL AGAIN

  Win, lose, or roll again
  (Case block)

RETURN

```

The subroutine might be written as follows:



```

3000 'SUB WIN, LOSE, OR ROLL AGAIN
3010 'CASE
3020     IF SUM = 7 OR SUM = 11 THEN PRINT "YOU WIN"
3030     IF SUM <= 3 OR SUM = 12 THEN PRINT "YOU LOSE"
3040     IF SUM > 3 AND SUM < 11 AND SUM <> 7
        THEN GOSUB 4000 'Roll dice for point
3050 'END CASE
3060 RETURN

```

The winning or losing on the first roll is dealt with directly in the subroutine. If you do not win or lose on the first roll, then you must continue rolling the dice until you make your point or roll a 7. Introducing a subroutine to take care of this situation allows you to complete the case statement and delay the details of the "roll again" task until later.

### Roll for Point Subroutine

You can now focus on writing the details of rolling the dice until you make your point or not. After a fair amount of thinking, you might decide that a loop block with an enclosed case block is appropriate. A subroutine outline might look like this:

```

SUB ROLL DICE FOR POINT
    Set your point
    Loop to keep rolling
    Roll dice
    Check if point or 7 was rolled
    Display outcome
RETURN

```

You will exit the loop whenever you roll your point or a 7. The subroutine follows:

```

4000 'SUB ROLL DICE FOR POINT
4010     LET YOURPOINT = SUM
4020     PRINT
4030     PRINT "YOUR POINT IS "; YOURPOINT
4040 'LOOP
4050     GOSUB 2000 'Roll dice & display sum
4060     'CASE
4070         IF SUM = YOURPOINT THEN LET A$ = "YOU WIN"
4080         IF SUM = 7 THEN LET A$ = "YOU LOSE"
4090     'END CASE
4100     IF SUM = YOURPOINT OR SUM = 7 THEN 4110
        ELSE 4040
4110 'END LOOP
4120     PRINT A$
4130 RETURN

```

The complete program follows:

```

1000 'PROGRAM DICE SIMULATION
1010   RANDOMIZE(TIMER)
1020   GOSUB 2000 'Roll dice & display sum
1030   GOSUB 3000 'Win, lose, or roll again
1040   END
1990 '
2000 'SUB ROLL DICE & DISPLAY SUM
2010   LET DIE1 = INT(6 * RND) + 1
2020   LET DIE2 = INT(6 * RND) + 1
2030   LET SUM = DIE1 + DIE2
2040   PRINT "YOUR ROLL IS "; SUM
2050   RETURN
2990 '
3000 'SUB WIN, LOSE, OR ROLL AGAIN
3010   'CASE
3020     IF SUM = 7 OR SUM = 11 THEN PRINT "YOU WIN"
3030     IF SUM <= 3 OR SUM = 12 THEN PRINT "YOU LOSE"
3040     IF SUM > 3 AND SUM < 11 AND SUM <> 7 THEN
        GOSUB 4000 'Roll dice for point
3050   'END CASE
3060   RETURN
3990 '
4000 'SUB ROLL DICE FOR POINT
4010   LET YOURPOINT = SUM
4020   PRINT
4030   PRINT "YOUR POINT IS "; YOURPOINT
4040   'LOOP
4050     GOSUB 2000 'Roll dice & display sum
4060   'CASE
4070     IF SUM = YOURPOINT THEN LET A$ = "YOU WIN"
4080     IF SUM = 7 THEN LET A$ = "YOU LOSE"
4090   'END CASE
4100     IF SUM = YOURPOINT OR SUM = 7 THEN 4110
        ELSE 4040
4110   'END LOOP
4120   PRINT A$
4130   RETURN

```

## Example 2 - Phrase Generator

A phrase generator program generates arbitrary random phrases or sentences from lists of words. The following program selects a subject, a verb, and an object from the following simple sentences:

```

The horse walked to the city.
Jennifer enjoyed the party.
The frog jumped in the pond.
The mayor called New York.

```

One-dimensional string arrays with four elements in each will be used to hold the subject list, the verb list, and the object list obtained from these four sentences.

The program outline might look like this:

```

PROGRAM PHRASE GENERATOR
  Set up arrays
  Seed random-number generator
  Load words from sentences
  Select phrases
  Display phrases
END

```

Two subroutines will be needed: one to load the words and one to randomly select and print out ten phrases. The complete program follows:

```

100 'PROGRAM PHRASE GENERATOR
110   DIM SUBJECT$(4), VERB$(4), OBJECT$(4)
120   RANDOMIZE(TIMER)
130   GOSUB 200 'Load words
140   GOSUB 300 'Select phrases
150 END
190 '
200 'SUB LOAD WORDS
210   FOR K = 1 TO 4
220     READ SUBJECT$(K), VERB$(K), OBJECT$(K)
230   NEXT K
240   DATA "THE HORSE ", "WALKED TO ", "THE CITY"
250   DATA "JENNIFER ", "ENJOYED ", "THE PARTY"
260   DATA "THE FROG ", "JUMPED IN ", "THE POND"
270   DATA "THE MAYOR ", "CALLED ", "NEW YORK"
280 RETURN
290 '
300 'SUB SELECT PHRASES
310   FOR K = 1 TO 10
320     LET S = INT(4 * RND) + 1
330     LET V = INT(4 * RND) + 1
340     LET O = INT(4 * RND) + 1
350     PRINT SUBJECT$(S); VERB$(V); OBJECT$(O)
360   NEXT K
370 RETURN

```

You may wish to enter this program and run it to see what phrases are generated.

### Example 3 - Random Walk

This program uses the graphics screen to simulate a random walk. If you walk about in a city aimlessly, or at random, you are taking a random walk. You could take different walks or paths.

The program below simulates a random walk in a city where all the blocks are the same size (10). The infinite loop in the main routine can be terminated by pressing **CtrlBreak**. The selection of one of the four possible directions at each corner is based on the choice of a random number between 1 and 4. The direction of

the walk is determined by the case block in subroutine **SELECT DIRECTION**. The complete program follows:

```

100 'PROGRAM RANDOM WALK
110   RANDOMIZE(TIMER)
120   CLS
130   SCREEN 1
140   'LOOP
150     GOSUB 300 'Select direction
160     DRAW D$
170     GOTO 140
180   'END LOOP
190 END
290 '
300 'SUB SELECT DIRECTION
310   LET R = INT(4 * RND) + 1
320   'CASE
330     IF R = 1 THEN LET D$ = "R10"
340     IF R = 2 THEN LET D$ = "L10"
350     IF R = 3 THEN LET D$ = "U10"
360     IF R = 4 THEN LET D$ = "D10"
370   'END CASE
380 RETURN

```

#### Example 4 - Circles

Suppose you wish to create a work of art made up of circles of random sizes. You would select a random position for the circle on the canvas and then select a random radius. The following program creates such a work of art on the graphics screen:

```

1000 'PROGRAM RANDOM CIRCLES
1010   RANDOMIZE(TIMER)
1020   CLS
1030   SCREEN 1
1040   FOR K = 1 TO 50
1050     GOSUB 2000 'Select center & radius
1060     CIRCLE (X,Y), R
1070   NEXT K
1080 END
1990 '
2000 'SUB SELECT CENTER & RADIUS
2010   LET X = INT(320 * RND)
2020   LET Y = INT(200 * RND)
2030   LET R = INT(30 * RND)
2040 RETURN

```

You may wish to enter and run this program to create your own works of art.

**Example 5 - Birthday Pairs in a Crowd**

Suppose there are 50 people in a room. What is the probability that 2 of the people have the same birthday? (Consider only the day of the year, not the year of birth.) This problem is famous in probability theory and has surprising results. You can attack the problem with the following strategy. By generating random integers over the range 1 to 365, you can simulate a birthday for each of the people. If you use a one-dimensional array for the birthdays as they are generated, it is easy to check for identical birthdays. After the birthdays have all been loaded into the array, begin with the first birthday **B(1)**, program the computer to check if it matches any of the remaining ones. Then have the computer check **B(2)** in the same way, and so on.

The most complex part of the program is the **CHECK FOR MATCH** subroutine. Read it carefully. It should check each value of **B(K)** (starting at **K = 1**) against the rest of the array elements (starting at **K + 1**). If a match is found, then **PAIRS** should be increased by 1. The complete program follows:

```

100 'PROGRAM BIRTHDAY PAIRS
110   DIM B(50)
120   RANDOMIZE(TIMER)
130   GOSUB 200 'Load array
140   GOSUB 300 'Check for match
150   PRINT "NUMBER OF BIRTHDAY PAIRS IS "; PAIRS
160   END
190 '
200 'SUB LOAD ARRAY
210   FOR K = 1 TO 50
220     LET B(K) = INT(365 * RND) + 1
230   NEXT K
240   RETURN
290 '
300 'SUB CHECK FOR MATCH
310   LET PAIRS = 0
320   FOR K = 1 TO 49
330     FOR J = K + 1 TO 50
340       IF B(K) = B(J) THEN 350 ELSE 370
350       'THEN CLAUSE
360         LET PAIRS = PAIRS + 1
370       'END IF
380     NEXT J
390   NEXT K
400   RETURN

```

There is still a bug in this program. What happens if there are three or more people in the room with the same birthday? You might give some thought to how this bug could be fixed.

This is a very interesting program to experiment with. You can modify the number of people in the crowd by making simple changes in the program. You can run the program many times to see how many birthday pairs, on the average, there will be in a crowd of a specified size.



**Example - 6 Shuffle and Deal**

Suppose you wish to simulate a card game. Usually, the deck of cards is shuffled and cards are then dealt from the top of the deck. This example simulates shuffling a deck of cards and then dealing out five cards. The outline of this program might look like the following:

```

PROGRAM SHUFFLE AND DEAL
  Set up the array
  Seed the random-number generator
  Shuffle the deck
  Deal five cards
END

```

The **SHUFFLE DECK** routine uses the following procedure to shuffle the deck. First, each element in an array to hold the deck is set equal to its own subscript. Then, you choose a random number **R** between 1 and 52. Interchange the element at position **R** in the array with the first element of the array. Choose a new random number **R**, but this time between 2 and 52. Interchange the element at position **R** with the second element in the array. Repeat this procedure until 52 cards have been selected.

If the first value of **R** is 27, then the number 27, which is in position 27 in the array **DECK**, will be interchanged with the number 1 in the first position. If the second choice of **R** is also 27 (between 2 and 52), then the number 1 will be interchanged with 2 in the second position. Thus, in this case, the first two cards have numbers 27 and 1. Continuing in this way, you reorder the numbers in the array. The subroutine for shuffling the deck might look like this:

```

300 'SUB SHUFFLE DECK
305   LET J = 1
310   LET N = 52
315   'LOOP
320     LET R = INT(N * RND) + J
325     LET TEMP = DECK(R)
330     LET DECK(R) = DECK(J) 'Switch values
335     LET DECK(J) = TEMP
340     IF J = 52 THEN 360
345     LET J = J + 1
350     LET N = N - 1
355     GOTO 320
360   'END LOOP
365   RETURN

```

To deal the cards, you need to be able to convert the numbers 1 through 52 to a suit and face value. The subroutine beginning at line number 500 does this. The complete program follows:

```

100 'PROGRAM SHUFFLE CARDS
110   DIM DECK(52)
120   RANDOMIZE(TIMER)
130   GOSUB 200 'Initialize deck

```

```

140     GOSUB 300 'Shuffle deck
150     GOSUB 400 'Deal 5 cards
160     END
190 '
200 'SUB INITIALIZE DECK
210     FOR K = 1 TO 52
220         LET DECK(K) = K
230     NEXT K
240     RETURN
290 '
300 'SUB SHUFFLE DECK
305     LET J = 1
310     LET N = 52
315     'LOOP
320         LET R = INT(N * RND) + J
325         LET TEMP = DECK(R)
330         LET DECK(R) = DECK(J) 'Switch values
335         LET DECK(J) = TEMP
340         IF J = 52 THEN 360
345         LET J = J + 1
350         LET N = N - 1
355         GOTO 315
360     'END LOOP
365     RETURN
390 '
400 'SUB DEAL 5 CARDS
410     FOR K = 1 TO 5
420         LET CARD = DECK(K)
430         GOSUB 500 'Suit & face value
440         PRINT TAB(5); FACE$; TAB(10); SUIT$
450     NEXT K
460     RETURN
490 '
500 'SUB SUIT & FACE VALUE
505     'Find suit
510         LET X = INT((CARD - 1) / 13) + 1
515         'CASE
520             IF X = 1 THEN LET SUIT$ = "CLUB"
525             IF X = 2 THEN LET SUIT$ = "DIAMOND"
530             IF X = 3 THEN LET SUIT$ = "HEART"
535             IF X = 4 THEN LET SUIT$ = "SPADE"
540         'END CASE
545     'Convert card to range 1-13
550     'LOOP
555         IF CARD <= 13 THEN 570
560         LET CARD = CARD - 13
565         GOTO 550
570     'END LOOP
575     'Find face
580     'CASE

```

```

585      IF CARD < 11 AND CARD > 1 THEN LET FACE$ =
        STR$(CARD)
590      IF CARD = 1 THEN LET FACE$ = " A"
595      IF CARD = 11 THEN LET FACE$ = " J"
600      IF CARD = 12 THEN LET FACE$ = " Q"
605      IF CARD = 13 THEN LET FACE$ = " K"
610      'END CASE
615      RETURN

```

## 10—5 PROBLEMS

1. Write a program to generate and print 25 random numbers of the form X.Y where X and Y are digits selected randomly from the set 0, 1, 2, 3, ..., 9.
2. Write a program to generate and print 50 integers selected at random from the range 13 to 25.
3. What will be printed if the following program is run?

```

100 'PROGRAM PROBLEM 3
110   FOR N = 1 TO 20
120     PRINT (INT(20 * RND) + 1) / 100
130   NEXT N
140   END

```

4. If you run the following program, what will you see on the screen?

```

100 'PROGRAM PROBLEM 4
110   FOR K = 1 TO 10
120     PRINT INT(100 * RND) / 10
130   NEXT K
140   END

```

5. Write a program that will simulate tossing a coin 10, 50, 500, and 1000 times. In each case, print the total number of heads and tails that occur.
6. Construct a dice-throwing simulation in BASIC. The dice are to be thrown 20 times. For each toss, print the die faces that are uppermost.
7. Write a program to generate and print out the average of 100 random numbers selected from the range 0 to 1.
8. Modify the program in Example 5 and run it as many times as needed to find the size of the crowd such that there is a 50 percent chance that at least two people in the crowd have the same birthday.
9. John and Bill want to meet at the library. Each agrees to arrive at the library sometime between 1:00 and 2:00 pm. They further agree that each will wait 10 minutes after arriving (but not after 2:00 pm). If after 10 minutes the other person has not arrived, the first one to arrive will leave. Write a BASIC program to compute the probability that John and Bill will meet. Do a simulation of the problem using the random-number generator.

10. Suppose a basket contains colored golf balls. There are 10 red, 5 blue, 2 green, and 11 yellow balls. Write a BASIC program to simulate drawing 5 balls at random from the bucket. The balls are drawn in sequence and are not replaced after being drawn.
11. Use the rule given in the discussion section in this chapter to generate and print out 25 numbers selected at random from a bell curve distribution of numbers with mean 10 and standard deviation 2. Round off the numbers to two places past the decimal point.
12. Suppose a soap manufacturer decides to select a five-character brand name. The first, third, and fifth characters are selected at random from the letters BCD-FGHJKLMNPQRSTUVWXYZ. The second and fourth letters are selected at random from the vowels AEIOU. Write a program to generate and print out 100 trial soap names using the rules above.
13. Modify the program from problem 6 to simulate tossing a pair of dice 1000 times. Print out the number of times each of the 11 possible outcomes happened in the simulation.

## 10—6 PRACTICE TEST

1. Write a BASIC program to generate and print out 100 random integers selected from the set 1, 2, 3, and 4.
2. Write a BASIC program to generate and print 100 random numbers over the range 25 to 50.
3. What will be printed if you run the following program?

```

100 'PROGRAM PRACTICE TEST 3
110 FOR K = 1 TO 10
120   LET N = INT(2 * RND) + 1
130   IF N = 1 THEN 140 ELSE 170
140     'THEN CLAUSE
150       PRINT "WHITE"
160       GOTO 190
170     'ELSE CLAUSE
180       PRINT "RED"
190   'END IF
200 NEXT K
210 END

```

---

4. What will be printed if you run the following program?

```

100 'PROGRAM PRACTICE TEST 4
110   FOR J = 1 TO 5
120     PRINT INT(100 * RND) / 100
130   NEXT J
140 END

```

---

# FILES

## 11—1 OBJECTIVES

### Opening and Closing Buffers

Before you can store information in a file, you have to create a holding area—a buffer—in which the information is held before it is transferred. You will learn to open and close these areas.

### Storing Information to a File

You will learn to create files in which to store information in two ways. The **WRITE** statement stores information in a **DATA** statement format. **LSET** stores information in a more structured format.

### Retrieving Information from a File

Once information is stored, you must be able to retrieve it. You will study methods of retrieving information stored on files.

### Modifying Information Stored in Files

Frequently, you need to change the information stored in files. You will learn how to do this.

### Working with Program Examples

You will learn to write programs that use files to manage information. You will see how such files can be used effectively to store and retrieve needed information.

## 11—2 DISCOVERY EXERCISES

Historically, files have been difficult to work with on computers. Fortunately, advances in computer languages have significantly eased the difficulties. You will find that writing programs that use files requires many techniques and concepts learned in previous chapters.



1. Turn on your computer and bring up BASICA.

### Opening and Closing Buffers

2. Suppose you want to write a program that stores information about a gift contributor and the amount given. Start with the following program outline:

```

PROGRAM GIFTLIST WRITE
  Open buffer
  Enter information
  Write to buffer
  Buffer to diskette
  Close buffer
END

```

Notice that a buffer is used. A buffer is a holding area in memory set aside to store information temporarily. Information must be assigned to a variable and then transferred to a buffer before it can be placed in a file on diskette. As indicated in the program outline, a buffer is opened and then closed after information in it is transferred to a file on diskette.

### Storing Information to a File

3. The program below could be written from the program outline. The remarks have been included for clarity. Enter it now.

```

100 'PROGRAM GIFTLIST WRITE
110   OPEN "GIFTLIST" AS #1 'Open buffer
120   INPUT "LAST NAME: "; LASTNAME$ 'Enter information
130   INPUT "AMOUNT: "; AMT
140   WRITE #1, LASTNAME$, AMT 'Write to buffer
150   PUT #1, 2 'Buffer to diskette
160   CLOSE #1 'Close buffer
170   END

```

A record is a structured set of information. A random-access file, the type of diskette file you will be using, is a collection of records. Line 110 creates the data file **GIFTLIST** and assigns to it the #1 holding area or buffer. In line 140, the **WRITE** statement places **LASTNAME\$** and **AMT** in this buffer. The **PUT** statement in line 150 then places the contents of the buffer in the file **GIFTLIST** on the diskette. The **2** in line 150 indicates that the computer will place the information into the second record of the random access file **GIFTLIST**. Line 160 closes buffer #1.

Display the program and check it, especially the semicolons and commas.

4. Run the program and enter your name and an appropriate gift amount. Did the disk light go on?

---

What happened to the information represented by **LASTNAME\$** and **AMT**?

---

5. Save this program as **WRITDISK**. (See Chapter 2 if you have forgotten how to do this.)
6. Enter

### FILES

to obtain a list of files and programs on your diskette. **BASICA** programs are identified by a **.BAS** at the end of the file name. Where is **GIFTLIST** found on the list?

---

### Retrieving Information from a File

7. Clear the memory and enter the following program:

```
100 'PROGRAM GIFTLIST READ
110   OPEN "GIFTLIST" AS #1
120   GET #1, 2
130   INPUT #1, LASTNAME$, AMT
140   CLOSE #1
150   END
```

In line 110, the buffer #1 is assigned to the file **GIFTLIST** created when you ran the **WRITDISK** program. The **GET** statement in line 120 places the contents of the second record of **GIFTLIST** into this buffer. In line 130, the **INPUT** statement assigns the contents of the buffer to the variables listed. Finally, the buffer is closed in line 140. Once the information from a record is assigned to variables, you can use it in the program.

8. Display the program and check it. Run the program. What happened?
- 

9. Add line 132 as follows:

```
132   PRINT LASTNAME$, AMT
```

Run the program. What was displayed this time?

---

10. Save this program as **READDISK**.
11. You should open and close a file buffer each time you use it. To write to a diskette file, use **WRITE** and **PUT** statements. To read from a file, use **GET** and **INPUT** statements.

### Storing More Than One Record

12. Now let's move on to using files with more than one record. Load the program **WRITDISK**. Display the program. Add the the following lines:

```

102   LET K = 1
104   PRINT "ENTER QUIT FOR LASTNAME WHEN DONE."
112   'LOOP
125   IF LASTNAME$ = "QUIT" THEN 156
152   LET K = K + 1
154   GOTO 112
156   'END LOOP

```

13. Change line 150 to read:

```

150   PUT #1, K

```

Enter **RENUM 100** to renumber the program. Display the program once again. Check its accuracy by comparing it with the program below. Note that you will have to indent some lines and that the trailing remarks have been eliminated.

```

100 'PROGRAM GIFTLIST WRITE
110 LET K = 1
120 PRINT "ENTER QUIT FOR LASTNAME WHEN DONE."
130 OPEN "GIFTLIST" AS #1
140 'LOOP
150 INPUT "LAST NAME: "; LASTNAME$
160 IF LASTNAME$ = "QUIT" THEN 220
170 INPUT "AMOUNT: "; AMT
180 WRITE #1, LASTNAME$, AMT
190 PUT #1, K
200 LET K = K + 1
210 GOTO 140
220 'END LOOP
230 CLOSE #1
240 END

```

14. Run the program. At the input prompts, enter the following information:

JONES	100
BROWN	35
CASTENADA	125
SMITH	25
RIZZO	65

What do you enter to end the information entry? (See line 120.)

---

Try it to see if you were correct.

15. Save this new program under the name **WRITDISK**.
16. Now let's look at how the **GIFTLIST** file is formatted. Enter

**SYSTEM**

to return to the PC DOS operating system (A>). Now enter

**TYPE GIFTLIST**

What happened?

---

Note the quotation marks enclosing the strings and the commas between the names and amounts. You may recognize that this file format is the same as the **DATA** statement format.

### Selecting Information from a File

17. Bring up BASICA. Load **READDISK**. Display the program. Add lines 112, 114, 125, 134, 136, and 138 as follows:

```
112    LET K = 1
114    'LOOP
125    IF EOF(1) THEN 138
134    LET K = K + 1
136    GOTO 114
138    'END LOOP
```

Change line 120 as follows:

```
120    GET #1, K
```

18. Enter **RENUM 100** to renumber the program. Display the program and check its accuracy against the program below. Note the additional indentation.

```
100 'PROGRAM GIFTLIST READ
110 OPEN "GIFTLIST" AS #1
120 LET K = 1
130 'LOOP
140 GET #1, K
150 IF EOF(1) THEN 200
160 INPUT #1, LASTNAME$, AMT
170 PRINT LASTNAME$, AMT
180 LET K = K + 1
190 GOTO 130
200 'END LOOP
210 CLOSE #1
220 END
```

The expression **EOF(1)** in line 150 is true when the computer reaches the end of the file **GIFTLIST**. What do you think will happen if you run the program?

---

Run the program and see if you were right. Before going on, save the program as **READDISK**.

19. Now let's extract some information from your file. Display the program. Add the following lines:

```

162      IF AMT < 50 THEN 164 ELSE 175
164      'THEN CLAUSE
175      'END IF

```

Run the program. What names were displayed?

---

20. You can also compile information from the file. Display the program. Add the following lines:

```

125      LET C = 0
166      LET C = C + 1
215      PRINT "THERE ARE "; C; "GIFTS UNDER 50."

```

When the program is run, what do you think will be displayed? (See step 14.)

---

Run the program. Does the number displayed agree with the information you entered?

---

21. Now let's add some things up. Display the program. Add line 125 and change lines 166 and 215 as follows:

```

125      LET SUM = 0
166      LET SUM = SUM + AMT
215      PRINT "TOTAL FOR GIFTS UNDER 50 IS "; SUM

```

Display the program. Run the program. What was the total of the gifts under 50?

---

### Defining Record Structures Precisely

22. Random-access files also allow you to structure records more precisely. Load **WRITDISK**. Display the program. Add lines 135 and 185 and change lines 130 and 180 as follows:



```

130 OPEN "LISTGIFT" AS #1 LEN = 80
135 FIELD #1, 6 AS BLASTNAME$, 12 AS BAMS$
180 LSET BLASTNAME$ = LASTNAME$
185 LSET BAMS$ = STR$(AMT)

```

Make special note of line 130 where **GIFTLIST** has been changed to **LISTGIFT** and **LEN = 80** has been added. In line 135, the **FIELD** statement specifies that each record will have two items — the first of length 6 and the second of length 12. The **B**'s in the names are to indicate that they are buffer locations in a **FIELD** statement rather than the standard memory variables. Buffer location names must be different from variable names used in the program. The **LSET** statements in lines 180 and 185 place information precisely in the buffer — the first 6 places in the buffer will contain **LASTNAME\$** and the next 12 **STR\$(AMT)**. As with string arrays, every item placed in the buffer by an **LSET** statement must be a string.

23. Display the program. You will use the same information for names and amounts that you used in step 14, namely,

JONES	100
BROWN	35
CASTENADA	125
SMITH	25
RIZZO	65

Run the program and enter the information given above at the input prompts.

24. Let's compare the format of **GIFTLIST** with the format of **LISTGIFT**. Return to the PC DOS operating system (see step 16). Now enter

**TYPE GIFTLIST**

Describe the format of the file.

---

25. Now enter

**TYPE LISTGIFT**

How do these formats compare?

---

How many spaces are allotted for the names in the **LISTGIFT** file?

---

26. You will now read out the information in **LISTGIFT**. Bring up **BASICA**. Load **READDISK**. Display the program. Add lines 115 and 165 and change lines 110 and 160 as follows:

```

110 OPEN "LISTGIFT" AS #1 LEN = 80
115 FIELD #1, 6 AS BLASTNAME$, 12 AS BMT$
160 LET LASTNAME$ = BLASTNAME$
165 LET AMT = VAL(BMT$)

```

Notice that the **OPEN** and **FIELD** statements are the same as before. Since **BMT\$** is a string, you convert it to a numeric variable using the **VAL** function. The contents of the buffer locations are assigned to the program variable in lines 160 and 165. Run the program. What happened?

---

How do you explain what happened to the name CASTENADA?

---

27. That ends the Discovery Exercises. Turn off the computer and go on to the next section.

## 11—3 DISCUSSION

### Opening and Closing Buffers

While using a diskette file, you worked with a holding area or buffer. Information must be stored in a buffer before it can be written to a diskette file. **OPEN** and **CLOSE** statements open and close a buffer to a diskette file. For example, the lines

```

200 OPEN "FILEONE" AS #1
300 CLOSE #1

```

open and close buffer #1. Line 200 opens buffer #1 as the holding area for the random access file **FILEONE**. Line 300 closes buffer #1 as a holding area for **FILEONE**, thus releasing it for reassignment as a buffer for another file. **BASICA** automatically establishes three buffers for use with files. You can assign additional buffers when bringing up **BASICA** as described in Chapter 2 of the **BASIC** reference manual.

- Use **OPEN** and **CLOSE** to open and close buffers.

You can also specify the length of each record in a file with the **OPEN** statement. For example,

```

110 OPEN "FILEONE" AS #1 LEN = 50

```

assigns buffer #1 to **FILEONE**. The statement will create the file if it does not already exist. **BASICA** automatically sets aside 128 characters for each record in the file if there is no **LEN** clause in the **OPEN** statement. It is possible to set aside fewer characters for each record. For instance, in line 110 above, **LEN = 50** sets aside 50 characters for each record. Therefore, the information in each record can have no more than 50 characters. It is good programming technique to append the record

length to the name of every random-access file, even if it is the standard length of 128.

### Storing Information to a File

Once a file is opened, you can write to the file or read from it. To write to a file, you use a **WRITE** statement and a **PUT** statement. For example,

```
100 LET AMT = 5280
110 OPEN "FILEONE" AS #2 LEN = 50
120 WRITE #2, AMT
130 PUT #2, 5
```

will assign buffer #2 to **FILEONE** (line 110) and next place the contents of **AMT** in buffer #2 (line 120). Finally, the information in the buffer is placed on diskette in record 5 of **FILEONE** (line 130). If there are several information items, they will be written to the buffer and then to the record in the order that they appear in the **WRITE** statement.

In the following example, a file is opened and some information is written to it:

```
100 'PROGRAM FILE WRITE
110 LET K = 1
120 OPEN "QUIZ1" AS #1 LEN = 50
130 'LOOP
140 INPUT "LASTNAME: "; LASTNAME$
150 IF LASTNAME$ = "QUIT" THEN 210
160 INPUT "LETTER GRADE: "; G$
170 WRITE #1, LASTNAME$, G$
180 PUT #1, K
190 LET K = K + 1
200 GOTO 130
210 'END LOOP
220 CLOSE #1
230 END
```

In line 120 the file **QUIZ1** is opened (created if necessary), and each record is assigned a length of 50 characters. In line 170, **LASTNAME\$** and **G\$** are written to the buffer associated with **QUIZ1** (buffer #1). Line 180 tells the computer to copy these two information items from buffer #1 to record **K** of the file **QUIZ1** on diskette. Line 220 closes buffer #1 and thus releases it from the file **QUIZ1**.

■ Use **WRITE** and **PUT** to write to a file.

### Retrieving Information from a File

To read information from a file use a **GET** statement and an **INPUT** statement, as in the following example:

```

100 'PROGRAM FILE READ
110   LET K = 1
120   OPEN "QUIZ1" AS #1 LEN = 50
130   'LOOP
140     GET #1, K
150   IF EOF(1) THEN 200
160     INPUT #1, LASTNAME$, G$
170     PRINT LASTNAME$, G$
180     LET K = K + 1
190     GOTO 130
200   'END LOOP
210   CLOSE #1
220   END

```

Recall that without the instruction in line 170, the computer would display nothing on the screen. The **INPUT** statement in line 160 only reads information into the variables. What is then done with the variables depends on the desired outcome. Again, **GET** and **INPUT** are a pair of statements that are used together to copy information from a file into program variables.

In line 150, the exit condition of the loop is the expression **EOF(1)** which stands for "end of file number 1." . **EOF(1)** is true if the end of the file **QUIZ1** has been reached. When the loop terminates, the computer goes on to the **CLOSE** statement in line 210.

- Use **GET** and **INPUT** to read from a file.

### Designing Record Structures with Field Widths

You can set up a record structure and precisely define the lengths of each item in the record with the **FIELD** statement.

- Use **FIELD** to design record structures.

For example, the program segment

```

120 OPEN "DATAFILE" AS #3 LEN = 80
130 FIELD #3, 15 AS BNAME$, 7 AS BPHONE$

```

establishes a record format with length 80 and two items of length 15 and 7. You use **LSET** to place information in the buffer from program variables. For example, the program segment

```

190 LSET BNAME$ = NAME$
200 LSET BPHONE$ = PHONE$
210 PUT #3, 5

```

assigns the contents of the variables **NAME\$** and **PHONE\$** to the buffer locations **BNAME\$** and **BPHONE\$**, respectively. The first 15 places in the buffer are allocated



to the characters in **NAME\$**, the next 7, to the characters in **PHONE\$**. If there are more characters in **NAME\$** and **PHONE\$** than are allocated, the characters on the right are lost. Line 210 places the information in the buffer into record 5 of **DATAFILE**.

#### ■ Use LSET and PUT to write to structured record files.

To retrieve information from a file where structured records are used, you use **GET** and **LET**. For example, if the **FIELD** statement in line 130 above is in effect, the program segment

```
230 GET #3, 5
240 LET NAME$ = BNAME$
250 LET PHONE$ = BPHONE$
```

first places the information in record 5 into buffer #3. Lines 240 and 250 assign the information in buffer location **BNAME\$** (the first 15 places of the buffer) and buffer location **BPHONE\$** (the next 7 places) to the program variables **NAME\$** and **PHONE\$**.

#### ■ Use GET and LET to read from structured record files.

The space-saving features of structured record files are covered completely in Appendix B of the BASIC reference manual.

Another kind of file, called a sequential file, is also available in BASICA. You have seen that random-access files allow you to read from and write to any record by indicating the number of the record you want to use. By contrast, to access information in a sequential file, you must read every item in the file until you reach the information you want. Certain tasks are better handled with sequential files than with direct access files. Information about sequential files can also be found in your BASIC reference manual.

## 11—4 WORKING WITH PROGRAM EXAMPLES

You will write a menu-driven program to create and maintain a mailing list for advertising and billing purposes. The menu will list the choices available in the program. Each menu choice will be treated separately. The initial design of your program needs to allow for several choices. You can find a listing of the complete program, with all the menu choices, at the end of this section.

### Example 1 - Mail List Data Entry

A mailing list contains names, addresses, and other information about individuals. Thus, the program might request the following items:



```

FIRST NAME:  (You type in)
LAST NAME:   (You type in)
STREET:      (You type in)
CITY:        (You type in)
STATE:       (You type in)
ZIP CODE:    (You type in)
BALANCE:     (You type in)

```

For every person on the mailing list, there will be a record containing this information in the file. It is useful to keep track of the number of records in the file. An excellent way to accomplish this is to store this number in the first record of the file. The program below will use this technique. An outline of the program follows:

```

PROGRAM MAIL LIST
  Clear screen
  Determine number of records
  Loop
    Present menu
    Check exit condition
    Enter data
  End loop
END

```

The main program to accomplish this might look like the following:

```

1000 'PROGRAM MAIL LIST
1010   CLS
1020   GOSUB 7000 'Determine number of records
1030   'LOOP
1040     GOSUB 2000 'Present options menu
1050     IF ANS = 5 THEN 1100
1060     'CASE
1070       IF ANS = 1 THEN GOSUB 3000 'Enter data
1080     'END CASE
1090     GOTO 1030
1100   'END LOOP
1110   END

```

Notice that a CASE statement is used in line 1060 to allow for additional choices that will be added in the future. The subroutine to handle the menu follows:

```

2000 'SUB PRESENT OPTIONS MENU
2010   PRINT "1.  ENTER DATA"
2020   PRINT "5.  QUIT"
2030   'LOOP
2040     INPUT "ENTER NUMBER OF YOUR CHOICE "; ANS
2050     IF ANS >= 1 AND ANS <= 5 THEN 2080
2060     PRINT "ENTER A NUMBER BETWEEN 1 AND 5"
2070     GOTO 2030
2080   'END LOOP
2090   RETURN

```

Since you are at the beginning of the program design, there are only two choices available — enter data or exit the program. Notice that the loop in lines 2030-2080 warns you if an improper choice has been entered and allows you to choose again.

The following subroutine handles the data entry:

```

3000 'SUB ENTER DATA
3010   LET RECNUM = RECNUM + 1
3020   GOSUB 8000 'Request data
3030   OPEN "MAILLIST" AS #1 LEN = 128
3040   WRITE #1, FIRST$, LAST$, STREET$, CITY$,
        STATES$, ZIP$, BAL
3050   PUT #1, RECNUM
3060   GOSUB 9000 'Update number of records
3070   CLOSE #1
3080   RETURN

```

Notice the two subroutines called. The first ('Request data) obtains the data to be stored. The second ('Update number of records) updates and stores the number of records.

The subroutine to determine the number of records follows:

```

7000 'SUB DETERMINE NUMBER OF RECORDS
7010   OPEN "MAILLIST" AS #1 LEN = 128
7020   IF LOF(1) = 0 THEN 7030 ELSE 7060
7030   'THEN CLAUSE
7040       LET RECNUM = 1
7050       GOTO 7090
7060   'ELSE CLAUSE
7070       GET #1, 1
7080       INPUT #1, RECNUM
7090   'END IF
7100   CLOSE #1
7110   RETURN

```

Notice the interesting features of this subroutine. The length of the file **MAILLIST** (**LOF(1)**) is checked in line 7020. If it is 0, the number of records is set equal to 1. This is the case at the start of a new file. If the length of the file is not 0, then the number of records in the file is known to be stored in the first record of the file. The program then gets that number and makes it available.

The following subroutine does the relatively easy task of requesting the data to be stored on the file **MAILLIST**:

```

8000 'SUB REQUEST DATA
8010   INPUT "FIRST NAME: "; FIRST$
8020   INPUT "LAST NAME: "; LAST$
8030   INPUT "STREET: "; STREET$
8040   INPUT "CITY: "; CITY$
8050   INPUT "STATE: "; STATES$
8060   INPUT "ZIP CODE: "; ZIP$

```

```

8070     INPUT "PAYMENT BALANCE: "; BAL
8080     RETURN

```

Each time a record is written to the file, the program updates the number of records and stores that number in the first record of the file. The following subroutine accomplishes that task:

```

9000 'SUB UPDATE NUMBER OF RECORDS
9010     WRITE #1, RECNUM
9020     PUT #1, 1
9030     RETURN

```

You may wish to enter and run this part of the complete program using data of your own choosing.

### Example 2 - Mailing Label Program

This program will use the **MAILLIST** random-access file to generate mailing labels for envelopes. The labels should have three lines as shown below:

```

GEORGE JONES
1234 DATAFILE DRIVE
SAN JOSE, CA 95128

```

Modifying the program in Example 1 is not too difficult. You need to add lines 1072 and 2012 and subroutines 4000 and 9200 as follows:

```

1072             IF ANS = 2 THEN GOSUB 4000 'Complete labels
2012     PRINT "2.  COMPLETE SET OF LABELS"

```

```

4000 'SUB COMPLETE SET OF LABELS
4010     OPEN "MAILLIST" AS #1 LEN = 128
4020     FOR K = 2 TO RECNUM
4030         GET #1, K
4040         INPUT #1, FIRST$, LAST$, STREET$, CITY$,
            STATE$, ZIP$, BAL
4050         GOSUB 9200 'Print labels
4060     NEXT K
4070     CLOSE #1
4080     RETURN

```

```

9200 'SUB PRINT LABELS
9210     PRINT FIRST$; " "; LAST$
9220     PRINT STREET$
9230     PRINT CITY$; ", "; STATE$; " "; ZIP$
9240     PRINT
9250     PRINT
9260     RETURN

```

Note that line 2012 presents another selection that you can choose, namely, that of obtaining a complete set of labels of the individuals in the file **MAILLIST**. Line 1072 of the **CASE** statement creates a second task. If you choose 2 from the menu, you will obtain a complete set of labels. Subroutine 4000 gets each record from the file and subroutine 9200 prints the requested information. Also, the need for the **CASE** statement is clearer as you add more choices to the options menu. The complete menu-driven program with these subroutines appears at the end of this section.

### Example 3 - Selected Labels Program

Once you have entered records to a file using the Enter Data menu option, you can begin to work with the information in the file. For instance, you can decide under what conditions a record will be selected. In this example, you obtain a set of labels for billing customers by selecting the records for customers whose balances are greater than zero. To do this, modify the previous program by adding lines 1074 and 2014 and by adding subroutine 5000 as follows:

```

1074          IF ANS = 3 THEN GOSUB 5000 'Selected labels
2014          PRINT "3.  SELECTED SET OF LABELS"

5000 'SUB SELECTED SET OF LABELS
5010      OPEN "MAILLIST" AS #1 LEN = 128
5020      FOR K = 2 TO RECNUM
5030          GET #1, K
5040          INPUT #1, FIRST$, LAST$, STREETS$, CITY$,
              STATES$, ZIP$, BAL
5050          IF BAL > 0 THEN 5060 ELSE 5080
5060          'THEN CLAUSE
5070              GOSUB 9200 'Print labels
5080          'END IF
5090      NEXT K
5100      CLOSE #1
5110      RETURN

```

The heart of this subroutine is line 5050, which prints the label only if the person's balance is greater than zero. If not, the next record is obtained, and the process is repeated.

### Example 4 - Modifying the MAILLIST File

Changing a record requires a great deal of additional programming effort. You need to locate the record you wish to change and display the information in that record. Then the program should request changes in items in that record until all are made. These changes should be written to the same record, and a request for another record to be changed should be made. The substantial extension of the previous program (Example 3) requires the addition of lines 1076 and 2016, as well as subroutines 6000, 9400, 9600, and 9800 as shown below:

```

1076          IF ANS = 4 THEN GOSUB 6000 'Modify a record
2016          PRINT "4.  MODIFY A RECORD"

```



```

6000 'SUB MODIFY A RECORD
6010     PRINT "PLEASE ENTER THE LAST NAME OF"
6020     PRINT "THE RECORD YOU WISH TO MODIFY"
6030     INPUT PERSON$
6040     OPEN "MAILLIST" AS #1 LEN = 128
6050     GET #1, 1
6060     INPUT #1, RECNUM
6070     FOR K = 2 TO RECNUM
6080         GET #1, K
6090         INPUT #1, FIRST$, LAST$, STREET$, CITY$,
            STATES$, ZIP$, BAL
6100         GOSUB 9400 'Check match
6110     NEXT K
6120     CLOSE #1
6130     RETURN

9400 'SUB CHECK MATCH
9410     IF PERSON$ = LAST$ THEN 9420 ELSE 9490
9420     'THEN CLAUSE
9430     'LOOP
9440         GOSUB 9600 'Display record
9450         IF N = 8 THEN 9480
9460         GOSUB 9800 'Make change
9470         GOTO 9430
9480     'END LOOP
9490     'END IF
9500     RETURN

9600 'SUB DISPLAY RECORD
9610     PRINT "1.  FIRST NAME: "; FIRST$
9620     PRINT "2.  LAST NAME: "; LAST$
9630     PRINT "3.  STREET: "; STREET$
9640     PRINT "4.  CITY: "; CITY$
9650     PRINT "5.  STATE: "; STATES$
9660     PRINT "6.  ZIP CODE: "; ZIP$
9670     PRINT "7.  PAYMENT BALANCE: "; BAL
9680     PRINT "8.  EXIT"
9690     PRINT "ENTER THE NUMBER OF THE"
9700     PRINT "ITEM YOU WISH TO MODIFY ";
9710     INPUT N
9720     RETURN

9800 'SUB MAKE CHANGE
9810     'CASE
9820         IF N = 1 THEN INPUT "FIRST NAME: "; FIRST$
9830         IF N = 2 THEN INPUT "LAST NAME: "; LAST$
9840         IF N = 3 THEN INPUT "STREET: "; STREET$

```



```

9850      IF N = 4 THEN INPUT "CITY: "; CITY$
9860      IF N = 5 THEN INPUT "STATE: "; STATES$
9870      IF N = 6 THEN INPUT "ZIP CODE: "; ZIP$
9880      IF N = 7 THEN INPUT "PAYMENT BALANCE: "; BAL
9890      'END CASE
9900      CLOSE #1
9910      OPEN "MAILLIST" AS #1
9920      WRITE #1, FIRST$, LAST$, STREET$, CITY$,
        STATES$, ZIP$, BAL
9930      PUT #1, K
9940      RETURN

```

Subroutine 6000 requests the last name of the record you wish to modify. After getting RECNUM from the file, each record from the start, ( $K = 2$ ), is obtained from the file MAILLIST. Subroutine 9400 is called to determine if the last name you input matches the last name in the record currently being considered. If there is no match, the next record is processed. If a match occurs, you are presented with the change menu. When you are finished modifying the record, you choose menu item 8 to see the main menu again. Changes in a record are made via the subroutines starting at lines 9600 and 9800. The subroutine at line 9600 displays the record that matches the last name you input and allows you to choose the item you wish to change (or to exit). If you choose an item to modify, the subroutine beginning at line 9800 allows you to make the change, and the modified record is then rewritten to the file on diskette.

Here is the complete program which handles all the menu items developed in the four examples:

```

1000 'PROGRAM MAIL LIST
1010   CLS
1020   GOSUB 7000 'Determine number of records
1030   'LOOP
1040       GOSUB 2000 'Present options menu
1050   IF ANS = 5 THEN 1100
1060   'CASE
1070       IF ANS = 1 THEN GOSUB 3000 'Enter data
1072       IF ANS = 2 THEN GOSUB 4000 'Complete labels
1074       IF ANS = 3 THEN GOSUB 5000 'Selected labels
1076       IF ANS = 4 THEN GOSUB 6000 'Modify a record
1080   'END CASE
1090       GOTO 1030
1100   'END LOOP
1110   END
1990 '
2000 'SUB PRESENT OPTIONS MENU
2010   PRINT "1.  ENTER DATA"
2012   PRINT "2.  COMPLETE SET OF LABELS"
2014   PRINT "3.  SELECTED SET OF LABELS"
2016   PRINT "4.  MODIFY A RECORD"
2020   PRINT "5.  QUIT"
2030   'LOOP

```

```

2040      INPUT "ENTER NUMBER OF YOUR CHOICE "; ANS
2050      IF ANS >= 1 AND ANS <= 5 THEN 2080
2060      PRINT "ENTER A NUMBER BETWEEN 1 AND 5"
2070      GOTO 2030
2080      'END LOOP
2090      RETURN
2990      '
3000      'SUB ENTER DATA
3010      LET RECNUM = RECNUM + 1
3020      GOSUB 8000 'Request data
3030      OPEN "MAILLIST" AS #1 LEN = 128
3040      WRITE #1, FIRST$, LAST$, STREETS$, CITY$,
        STATES$, ZIP$, BAL
3050      PUT #1, RECNUM
3060      GOSUB 9000 'Update number of records
3070      CLOSE #1
3080      RETURN
3990      '
4000      'SUB COMPLETE SET OF LABELS
4010      OPEN "MAILLIST" AS #1 LEN = 128
4020      FOR K = 2 TO RECNUM
4030          GET #1, K
4040          INPUT #1, FIRST$, LAST$, STREETS$, CITY$,
            STATES$, ZIP$, BAL
4050          GOSUB 9200 'Print labels
4060      NEXT K
4070      CLOSE #1
4080      RETURN
4990      '
5000      'SUB SELECTED SET OF LABELS
5010      OPEN "MAILLIST" AS #1 LEN = 128
5020      FOR K = 2 TO RECNUM
5030          GET #1, K
5040          INPUT #1, FIRST$, LAST$, STREETS$, CITY$,
            STATES$, ZIP$, BAL
5050          IF BAL > 0 THEN 5060 ELSE 5080
5060          'THEN CLAUSE
5070              GOSUB 9200 'Print labels
5080          'END IF
5090      NEXT K
5100      CLOSE #1
5110      RETURN
5990      '
6000      'SUB MODIFY A RECORD
6010      PRINT "PLEASE ENTER THE LAST NAME OF"
6020      PRINT "THE RECORD YOU WISH TO MODIFY"
6030      INPUT PERSON$
6040      OPEN "MAILLIST" AS #1 LEN = 128
6050      GET #1, 1
6060      INPUT #1, RECNUM

```

```

6070     FOR K = 2 TO RECNUM
6080         GET #1, K
6090         INPUT #1, FIRST$, LAST$, STREET$, CITY$,
            STATES$, ZIP$, BAL
6100         GOSUB 9400 'Check match
6110     NEXT K
6120     CLOSE #1
6130     RETURN
6990 '
7000 'SUB DETERMINE NUMBER OF RECORDS
7010     OPEN "MAILLIST" AS #1 LEN = 128
7020     IF LOF(1) = 0 THEN 7030 ELSE 7060
7030         'THEN CLAUSE
7040             LET RECNUM = 1
7050             GOTO 7090
7060         'ELSE CLAUSE
7070             GET #1, 1
7080             INPUT #1, RECNUM
7090         'END IF
7100     CLOSE #1
7110     RETURN
7990 '
8000 'SUB REQUEST DATA
8010     INPUT "FIRST NAME: "; FIRST$
8020     INPUT "LAST NAME: "; LAST$
8030     INPUT "STREET: "; STREET$
8040     INPUT "CITY: "; CITY$
8050     INPUT "STATE: "; STATES$
8060     INPUT "ZIP CODE: "; ZIP$
8070     INPUT "PAYMENT BALANCE: "; BAL
8080     RETURN
8990 '
9000 'SUB UPDATE NUMBER OF RECORDS
9010     WRITE #1, RECNUM
9020     PUT #1, 1
9030     RETURN
9190 '
9200 'SUB PRINT LABELS
9210     PRINT FIRST$; " "; LAST$
9220     PRINT STREET$
9230     PRINT CITY$; ", "; STATES$; " "; ZIP$
9240     PRINT
9250     PRINT
9260     RETURN
9390 '
9400 'SUB CHECK MATCH
9410     IF PERSON$ = LAST$ THEN 9420 ELSE 9490
9420         'THEN CLAUSE
9430         'LOOP
9440             GOSUB 9600 'Display record

```

```

9450         IF N = 8 THEN 9480
9460         GOSUB 9800 'Make change
9470         GOTO 9430
9480         'END LOOP
9490     'END IF
9500     RETURN
9590 '
9600 'SUB DISPLAY RECORD
9610     PRINT "1.  FIRST NAME: "; FIRST$
9620     PRINT "2.  LAST NAME: "; LAST$
9630     PRINT "3.  STREET: "; STREET$
9640     PRINT "4.  CITY: "; CITY$
9650     PRINT "5.  STATE: "; STATE$
9660     PRINT "6.  ZIP CODE: "; ZIP$
9670     PRINT "7.  PAYMENT BALANCE: "; BAL
9680     PRINT "8.  EXIT"
9690     PRINT "ENTER THE NUMBER OF THE"
9700     PRINT "ITEM YOU WISH TO MODIFY ";
9710     INPUT N
9720     RETURN
9790 '
9800 'SUB MAKE CHANGE
9810     'CASE
9820         IF N = 1 THEN INPUT "FIRST NAME: "; FIRST$
9830         IF N = 2 THEN INPUT "LAST NAME: "; LAST$
9840         IF N = 3 THEN INPUT "STREET: "; STREET$
9850         IF N = 4 THEN INPUT "CITY: "; CITY$
9860         IF N = 5 THEN INPUT "STATE: "; STATE$
9870         IF N = 6 THEN INPUT "ZIP CODE: "; ZIP$
9880         IF N = 7 THEN INPUT "PAYMENT BALANCE: "; BAL
9890     'END CASE
9900     CLOSE #1
9910     OPEN "MAILLIST" AS #1
9920     WRITE #1, FIRST$, LAST$, STREET$, CITY$,
        STATE$, ZIP$, BAL
9930     PUT #1, K
9940     RETURN

```

This is the most complex program you have seen so far. However, note that the program evolved in easy steps. At each point, there was no question about what was needed or how to incorporate the changes into the program. This is due to the fact that top-down organization and program structures were used from the beginning. With the techniques you have learned, you can write similar programs by yourself.

## 11—5 PROBLEMS

1. Design an appropriate record structure for a random-access file to index your cassette tape collection. Be sure to indicate the maximum length of each item in the record and the total length in characters of the record. Determine valid BASICA variable names for each item in the record structure.



2. Design an appropriate record structure for a random-access file to keep track of your checkbook entries.
3. Design an appropriate record structure for a random-access file to keep an inventory of your household possessions.
4. Write out an appropriate record structure for a file to keep your mailing list. Remember that birthdays and anniversaries are important to your friends.
5. Write a program that will use a file called **CHARGE** to manage your charge cards. Each record should have the following structure:

Variable	Description	Length
CARD\$	Name of Card	20
STORE\$	Name of Store	30
CHARGEDATE\$	Date of Purchase	8
DES\$	Description of Purchase	50
AMT	Amount of Purchase	8

The program should allow you to total the amount of money you have charged to each card in the entire file. You may use the **FIELD** and **LSET** statements in the program if you wish.

6. Write a program that uses the record structure from problem 4 to manage your personal mailing list. The program should allow you to print labels for your Christmas cards and for messages to your friends at work.
7. Modify Example 4 (MAILLIST Program) to display the total amount owed (sum the payment balance of every record). You can replace the Selected Labels option and its subroutines to accomplish this task.
8. Modify Example 4 (MAILLIST Program) to display the names of all the people living in the 95000 to the 95200 zip code areas. You can replace the Selected Labels option and its subroutines to accomplish this task.

## 11—6 PRACTICE TEST

1. If the following program is run

```

100 'PROGRAM PRACTICE TEST 1
110 OPEN "FILETWO" as #1 LEN = 70
120 WRITE #1, "HELLO", "GEORGE"
130 PUT #1, 5
140 CLOSE #1
150 END

```

- a. What file will be used?

---

- b. How many characters are allowed in each record?

---



c. How many items are placed in the record?

---

d. Which record will be written to?

---

2. Write a program line that opens a random-access file named **INVENTORY** with record length 45.

---

3. Write a program that will read five information items from record 75 in a random-access file named **TEXT1** with record length 50. Be sure to close the file.

---

4. What is wrong with the following program line?

```
200 OPEN "FILEONE" AS 1 LEN 70
```

---

5. What is wrong with the following program lines for reading **FIRST\$** from record 5 of a data file named **FILEONE**?

```
200 OPEN "FILEONE" AS #1 LEN = 100
210 GET 1, 5
220 INPUT #1, FIRST$
```

6. In what positions in record 7 of **LIBRARY** will the following program segment place the information in the program variable **BOOK\$**?

```
300 OPEN "LIBRARY" AS #3 LEN = 200
310 FIELD #3, 14 AS BAUTHOR$, 30 AS BBOOK$,
    20 AS BPUB$
320 LSET BAUTHOR$ = AUTHOR$
330 LSET BBOOK$ = BOOK$
340 LSET BPUB$ = PUBLISHER$
350 PUT #3, 7
```

---

# GRAPHICS REVISITED

## 12—1 OBJECTIVES

### Creating and Using Graphics Regions

You will learn to reserve certain portions of the screen for graphics displays of lines and points. You will learn to designate the maximum and minimum coordinate values that can be used in the graphics region.

### Learning to Draw Arcs

You will learn to use the **CIRCLE** statement to draw arcs or portions of a circle.

### Learning to Use Arrays in Graphics

Shapes are determined by points. You will learn to draw shapes by using points stored as coordinate pairs in arrays.

### Labeling Displays

Labels help people understand graphics displays. You will learn how to label shapes.

### Working with Program Examples

You will learn to organize graphics statements to display and label shapes and charts.

## 12—2 DISCOVERY EXERCISES

1. Turn on the computer and bring up BASICA.

### The Graphics Region

2. Ordinarily, the graphics region is the entire screen. The **VIEW** statement can be used to restrict the portion of the screen where points and lines can be plotted. Enter the following program:

```

100 'PROGRAM GRAPHICS REGION
110     KEY OFF
120     CLS
130     SCREEN 1
140     GOSUB 200 'Enter corners
150     VIEW (A, B) - (C, D), , 1
160     LINE (10, 10) - (20, 20)
170     END
190 '
200 'SUB ENTER CORNERS
210     INPUT "ENTER UPPER LEFT CORNER "; A, B
220     INPUT "ENTER LOWER RIGHT CORNER "; C, D
230     RETURN

```

The **KEY OFF** statement in line 110 keeps the line ordinarily at the bottom of the screen from being displayed when you run the program. Note the two commas near the end of line 150. Run the program. At the input prompt, enter the following pairs of coordinates:

```

20, 20
100, 100

```

What happened?

---

Was the line drawn inside the box or not?

---

The box is the boundary of the graphics region.

3. Run the program again, using the following coordinates:

```

20, 20
150, 70

```

How has the graphics region changed?

---

Where is the line drawn this time?

---

4. Run the program using the following coordinates:

160, 20  
260, 120

Where is the graphics region this time?

---

How long are the sides of the box?

---

The sides are 100 units each, the difference between the coordinates you entered.  
Run the program again using the following coordinates:

100, 100  
150, 150

Where is the graphics region?

---

How large is the graphics region this time?

---

### Redefining Coordinate Values in the Graphics Region

5. Ordinarily, the coordinate values can be between 0 and 319 horizontally and between 0 and 199 vertically. The **WINDOW** statement allows you to change these limits. Add line 145 as follows:

145     **WINDOW** (10, 10) - (20, 20)

Display the program. Run the program entering the following coordinate pairs:

20, 20  
100, 100

Where is the line drawn this time?

---

Display lines 140 through 150 by entering the following command:

**LIST 140-150**

You have entered the coordinates for the upper left corner of the graphics region (**A**, **B**) and the lower right corner (**C**, **D**). The **WINDOW** statement limits the coordinate values that can be plotted in the graphics region. The horizontal coordinates can be between 10 and 20. The same is true for the vertical coordinates. Also, the **WINDOW** statement changes the orientation of the graphics region. The orientation is now the standard one you have seen in newspaper and magazine charts and graphs. Thus, the lower left corner of the graphics region is at (10,10) while the upper right corner is at (20,20). The **LINE** statement draws a line between these two points, the corners of the box. How does this compare to the way the line was drawn before you added the **WINDOW** statement?

---

6. Run the program again, entering the following coordinates:

**20, 20**  
**150, 70**

Where was the line drawn this time?

---

Is the line longer or shorter than before?

---

7. Edit line 145 as follows:

**145 WINDOW (0, 0) - (20, 20)**

Run the program and enter the following coordinates:

**20, 100**  
**150, 150**

Where was the line drawn this time?

---

List lines 140 through 150. Compare the **LINE** statement with the **WINDOW** statement to see why the line is drawn this way. Notice that the center of the graphics region is at (10,10) since the horizontal and vertical coordinates are both between 0 and 20.



8. Now edit line 145 as follows:

```
145    WINDOW (0, 0) - (40, 40)
```

List lines 140 through 150. Where do you think the line will be drawn this time?

---

Where is the center of the graphics region?

---

Run the program with the same data as in step 7 to see if your answers were correct.

### Multiple Graphics Regions

9. Clear the memory and the screen. Enter the following program:

```
100 'PROGRAM 4 VIEWS
110    KEY OFF
120    CLS
130    SCREEN 1
140    FOR K = 0 TO 144 STEP 48
150        VIEW (K + 20, K + 20) - (K + 50, K + 50), , 1
160        WINDOW (0, 0) - (10, 10)
170    NEXT K
180    PSET (5, 5)
190    END
```

Run the program. In which graphics region does PSET (5, 5) place the point?

---

Add line 165 as follows:

```
165    PSET (2, 2)
```

In what graphics region(s) do you think the statement PSET (2, 2) will plot a point?

---

Where in the graphics region(s) will the point be plotted?

---

Run the program to see if you were correct.

**Drawing Arcs**

10. Clear the memory and the screen. Enter the following program:

```

100 'PROGRAM DRAWING ARCS
110   KEY OFF
120   CLS
130   SCREEN 1
140   VIEW (0, 0) - (319, 199)
150   WINDOW (-2, -2) - (2, 2)
160   LET FULLCIRCLE = 2 * 3.14159
170   INPUT "ENTER TWO NUMBERS: "; A, B
180   LET START = A * FULLCIRCLE
190   LET FINISH = B * FULLCIRCLE
200   CIRCLE (0, 0), 1, , -START, -FINISH
210   END

```

Run the program. Enter the following pair of numbers at the input prompt:

0, .25

What happened?

---

Run the program again and enter

0, .5

What portion of a circle was drawn?

---

Run the program again entering

0, 1

What portion of the circle was drawn this time?

---

Run the program and enter

.5, .75

What portion of the circle was drawn this time?

---

How does the position of this arc compare with that of the arc drawn with the numbers 0, .25?

---

11. Edit line 200 by removing the minus signs as follows:

```
200    CIRCLE (0, 0), 1, , START, FINISH
```

Run the program and enter

.5, .75

How does this arc differ from the previous one?

---

Edit line 200 as follows:

```
200    CIRCLE (0, 0), 1, , -START, FINISH
```

What do you think will happen this time when you run the program and enter .5, .75?

---

Run the program and see if you were correct. Display the program. The number `2 * 3.14159` in line 160 is the measure of a full circle angle (360 degrees) in radians. Hence, the decimal numbers you entered are fractions (between -1 and 1) of the circle.

### Storing Shapes in Arrays

12. Clear the memory and the screen. You will now explore saving graphics shapes in arrays. The following program will be used throughout the remainder of the discovery exercises. You should enter it now.

```
1000 'PROGRAM SHAPE ARRAY
1010   GOSUB 2000 'Set screen
1020   GOSUB 3000 'Load shape array
1030   GOSUB 4000 'Set starting position
1040   GOSUB 5000 'Draw shape
1050   END
1990 '
2000 'SUB SET SCREEN
2010   KEY OFF
2020   CLS
2030   SCREEN 1
2040   VIEW (85, 25) - (235, 175), , 1
2050   WINDOW (0, 0) - (200, 200)
2060   RETURN
2990 '
3000 'SUB LOAD SHAPE ARRAY
3010   READ N
3020   DIM SHAPE(100, 2)
3030   FOR K = 1 TO N
```

```

3040      READ SHAPE(K, 1),  SHAPE(K, 2)
3050      NEXT K
3060      DATA 6
3070      DATA 0, 0
3080      DATA 0, 30
3090      DATA 20, 40
3100      DATA 40, 30
3110      DATA 40, 0
3120      DATA 0, 0
3130      RETURN
3990 '
4000 'SUB SET STARTING POSITION
4010      INPUT "ENTER STARTING POINT: "; A, B
4020      PSET (A, B), 0
4030      RETURN
4990 '
5000 'SUB DRAW SHAPE
5010      FOR K = 1 TO N
5020          LINE - (A + SHAPE(K, 1), B + SHAPE(K, 2))
5030      NEXT K
5040      RETURN

```

Check the program by displaying portions of it. Run the program and enter the numbers

0, 0

at the input prompt. What happened?

---

The house is placed inside the graphics region. Describe the location of the graphics region.

---

Is the point (0,0) in the lower left corner or in the upper left corner?

---

13. Run the program again and enter

100, 100

Where is the lower left corner of the house in the graphics region?

---

Run the program and enter

150, 10


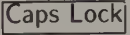

Where is the house placed this time?

14. You placed the house in the graphics region by entering the position of the lower left corner. Now you will place the house in the graphics region by moving the graphics cursor to the desired position. Change the **SET STARTING POSITION** subroutine (4000) and add the **PRESS A KEY** subroutine (4500) as follows:

```

4000 'SUB SET STARTING POSITION
4010   LET A = 100
4020   LET B = 100
4030   PSET (A, B)
4040   'LOOP
4050   LET A$ = "" 'Initialize A$
4060   GOSUB 4500 'Press a key
4070   LET LASTA = A
4080   LET LASTB = B
4090   'CASE
4100   IF A$ = "L" THEN LET A = A - 5
4110   IF A$ = "R" THEN LET A = A + 5
4120   IF A$ = "U" THEN LET B = B + 5
4130   IF A$ = "D" THEN LET B = B - 5
4140   'END CASE
4150   PSET (LASTA, LASTB), 0
4160   PSET (A, B)
4170   IF A$ = "Q" THEN 4190
4180   GOTO 4050
4190   'END LOOP
4200   RETURN
4490 '
4500 'SUB PRESS A KEY
4510   'LOOP
4520   IF A$ <> "" THEN 4550
4530   LET A$ = INKEY$
4540   GOTO 4510
4550   'END LOOP
4560   RETURN

```

In line 4530, the new function **INKEY\$** assigns whatever key is pressed to the string variable **A\$**. When the computer enters this subroutine the first time, the value of **A\$** is the null string. The computer stays in the loop as long as the value remains the null string. As soon as a key is pressed, this condition is no longer true, and the computer leaves the loop. Type **RUN** but don't press  yet. If the letters are not uppercase, press the  key so that the subsequent letters will be capitalized. Now press  to run the program. When a point appears in the



center of the graphics region, press **L** five times and then press **Q** (to quit). What happened?

---

15. Run the program again. If you press **U**, which way will the point move?
- 

Press **U** five times and then press **Q**. Where was the house placed?

---

You may wish to experiment with moving the point around the graphics region before placing the house.

16. Modify the program so you can use the arrow keys to move the point about the graphics region. The only changes you need to make are in lines 4100 through 4130, as follows:

```

4100      IF MID$(A$, 2, 1) = "K" THEN LET A = A - 5
4110      IF MID$(A$, 2, 1) = "M" THEN LET A = A + 5
4120      IF MID$(A$, 2, 1) = "H" THEN LET B = B + 5
4130      IF MID$(A$, 2, 1) = "P" THEN LET B = B - 5

```

When an arrow key is pressed, **INKEY\$** assigns a two-character string to the string variable **A\$**. The first characters are the same for all the arrow keys, while the second characters differ. You can see why the **MID\$** function is used in the **CASE** block to decide which arrow key has been pressed. Run the program and use the arrow keys to move the point about the graphics region before pressing **Q** to place the house. How far does the point at the graphics cursor move each time you press an arrow key?

---

See what happens if you move the starting point for the house near the top or right side of the graphics region.

### Labeling Graphics Regions

17. Add line 1045 to the main routine and the following subroutine:

```

1045      GOSUB 7000 'Label graphics region
6990      '
7000      'SUB LABEL GRAPHICS REGION
7010      LOCATE 24, 16
7020      PRINT "SITE PLAN";
7030      RETURN

```

The **LOCATE** statement in line 7010 places the text cursor in row 24, column 16. In **SCREEN 1**, the text screen has 25 rows and 40 columns. Run the program. Press the

three times and then press . Where does **SITE PLAN** appear in relationship to the graphics region?

---

18. Finally, you will label the house after it is placed in the graphics region. To do this, you need to use the graphics **VIEW** coordinates of the house to position the text cursor. Add line 1047 to the main routine as well as the following subroutine:

```

1047   GOSUB 8000 'Label shape
7990   '
8000   'SUB LABEL SHAPE
8010   LET X = POINT(0)
8020   LET Y = POINT(1)
8030   LET COL = INT((X + 85) / 8) + 1
8040   LET ROW = INT((Y + 25) / 8) + 2
8050   LOCATE ROW, COL
8060   PRINT "HOUSE"
8070   RETURN

```

The upper left corner of the **VIEW** graphics region is at (85,25), and each character is 8 dots wide and 8 dots high. These numbers are used in the functions defined in the subroutine. The **POINT** functions return **VIEW** coordinates of the graphics cursor. The defined functions change these first and second graphics coordinates to column and row text locations. Run the program. Press  four times and then press . What happened?

---

Run the program again, pressing  three times. Then press . How does this placement of the **HOUSE** label compare with the previous one?

---

19. This ends the Discovery Exercises. Turn off the computer and go on to the next section.

## 12—3 DISCUSSION

A point in the graphics region is designated by two numbers called coordinates. The first coordinate determines the horizontal position of the point. The second coordinate determines its vertical position. Some graphics statements require parentheses and some do not. When in doubt, refer to the *BASIC* reference manual.

### Creating and Using Graphics Regions

Recall that on the standard medium-resolution graphics screen (**SCREEN 1**) you can plot only points whose first coordinate is between 0 and 319 and whose second coordinate is between 0 and 199. The point (0, 0) is in the upper left corner. You use the **VIEW** statement to restrict graphics displays to a particular portion of

the screen. Such a restricted portion of the screen is called a graphics region. For example,

```
VIEW (0, 0) - (160, 100)
```

will designate the upper left quarter of the screen as the graphics region. Here, again, the upper left corner of the screen is the point (0, 0). The center of the screen, (160,100), is over 160 and down 100 from this starting point. The center of the graphics region, however, is the point (80, 50). In the **VIEW** statement

```
VIEW (A, B) - (C, D)
```

the point (A, B) is the upper left corner of the reserved graphics region and (C, D) is the lower right corner of the region. The position of these points is determined starting in the upper left corner.

■ **The upper left corner is the starting position for VIEW.**

Only points with first coordinate between 0 and 319 and second coordinate between 0 and 199 can be plotted on the standard medium-resolution graphics screen. The **WINDOW** statement changes the limits for these coordinates. For example,

```
SCREEN 1
VIEW (0, 0) - (160, 100)
WINDOW (0, 0) - (200, 400)
```

makes the graphics region the upper left quarter of the screen. Also, this **WINDOW** statement allows only points with first coordinate between 0 and 200 and second coordinate between 0 and 400 to be plotted on this graphics region. In addition, the **WINDOW** statement reorients the graphics region so that the point (0, 0) is in the lower left corner of the graphics region. The point (100, 50) is located 100 units to the right and 50 units up from the lower left corner of the graphics region.

■ **WINDOW reorients the graphics region so that the lower left is the starting position.**

Let's look at another example. Running the program segment

```
130 SCREEN 1
140 VIEW (160, 100) - (319, 199)
150 WINDOW (-1, -1) - (1, 1)
160 PSET (1, 1)
170 PSET (0, 0)
```

makes the lower right quarter of the screen the graphics region and requires that all points plotted have first and second coordinates between -1 and 1. Also, the **WINDOW** statement reorients the screen. Thus, the points plotted are at the upper right corner and the center of the graphics region.

- Use **VIEW** to define the graphics region.
- Use **WINDOW** to determine the coordinates to be used.

### Learning to Draw Arcs

Use the **CIRCLE** statement to draw an arc of a circle. For example, running the following program segment

```

230  SCREEN 1
240  VIEW (0, 0) - (150, 150)
250  WINDOW (-100, -100) - (100, 100)
260  LET START = .25 * 2 * 3.14159
270  LET FINISH = .5 * 2 * 3.14159
280  CIRCLE (0, 0), 50, , -START, -FINISH

```

will draw a quarter circle in the center of the graphics region. The minus signs in line 280 cause the ends of the arc to be joined to the center of the circle. These line segments will not be drawn if the numbers are positive. The **START** and **FINISH** numbers are the measures of the angles (in radians) that determine the arc to be drawn.

### Learning to Use Arrays in Graphics

The coordinates of the points that form a shape can be stored in an array. This array can then be used to draw the shape. For example, the following program

```

1000 'PROGRAM FLAG ARRAY
1010  GOSUB 2000 'Set screen
1020  GOSUB 3000 'Load array
1030  GOSUB 4000 'Draw flag
1040  END
1990 '
2000 'SUB SET SCREEN
2010  KEY OFF
2020  CLS
2030  SCREEN 1
2040  VIEW (0, 0) - (319, 199)
2050  WINDOW (0, 0) - (100, 100)
2060  RETURN
2990 '
3000 'SUB LOAD ARRAY
3010  READ N
3020  DIM FLAG(N, 2)
3030  FOR K = 1 TO N
3040    READ FLAG(K, 1), FLAG(K, 2)
3050  NEXT K
3060  DATA 5
3070  DATA 0, 0

```



```

3080 DATA 0, 80
3090 DATA 30, 80
3100 DATA 30, 50
3110 DATA 0, 50
3120 RETURN
3990 '
4000 'SUB DRAW FLAG
4010 LET K = 1
4020 PSET (FLAG(K, 1), FLAG(K, 2))
4030 FOR K = 1 TO N
4040 LINE - (FLAG(K, 1), FLAG(K, 2))
4050 NEXT K
4060 RETURN

```

draws a flag in the lower left corner of the graphics region. Notice that the first **DATA** statement for the flag gives the number of points that form the flag. To place the flag at any point in the graphics region, you must modify subroutine **DRAW FLAG**. If you wish to draw the flag beginning at the point (10, 60), the **DRAW FLAG** subroutine should look like this:

```

4000 'SUB DRAW FLAG
4010 LET K = 1
4020 PSET (10 + FLAG(K, 1), 60 + FLAG(K, 2))
4030 FOR K = 1 TO N
4040 LINE - (10 + FLAG(K, 1), 60 + FLAG(K, 2))
4050 NEXT K
4060 RETURN

```

If you make these changes and run the program, you will see that part of the flag is outside the graphics region. The computer clips anything outside the graphics region. Only that information that falls inside the region is plotted.

This subroutine still uses the points that are stored in array **FLAG**, but adds 10 to all the first coordinates and 60 to all the second coordinates. Naturally, the program would be more flexible if the starting point were determined interactively, as in the Discovery Exercises. Because the coordinates of the points that form the flag start at (0, 0), you can scale the flag, as you saw in the Discovery Exercises.

### Labeling Displays

Labels help make graphics displays understandable. You use the **LOCATE** statement to position the text cursor where you wish to place the label. In medium-resolution graphics, there are 24 rows and 40 columns for text. The first row is at the top of the screen and the first column position is at the left of the screen. Thus, the direct mode statements

```
LOCATE 12, 20: PRINT "A"
```

will place the text cursor at the center of the screen and print the letter "A" there.

You need to know where to position the text cursor in order to label graphics regions and displays. Once you have created a graphics region, you place labels outside this region using the **LOCATE** statement.



To label shapes inside the graphics region, you need to know the graphics position of the shape. You then need to relate the row and column of the text cursor to the first and second coordinates of the graphics cursor. To do this, you must take into account the current **VIEW** and **WINDOW** settings. For example, running the program

```

1000 'PROGRAM LABEL A SHAPE
1010   GOSUB 2000 'Set screen
1020   VIEW (0, 0) - (319, 199), , 1
1030   WINDOW (0, 0) - (100, 100)
1040   PSET (50, 50)
1050   LET X = POINT(0)
1060   LET Y = POINT(1)
1070   LET COL = INT(X / 8)
1080   LET ROW = INT(Y / 8)
1090   LOCATE ROW, COL
1100   PRINT "DOT"
1110   END
1990 '
2000 'SUB SET SCREEN
2010   KEY OFF
2020   CLS
2030   SCREEN 1
2040   RETURN

```

will plot a point in the center of the screen and then print the word **DOT** just above it. The **POINT(0)** function returns the current physical first coordinate. Physical coordinate refers to the dots available on the screen. In medium-resolution graphics, there are 319 dots across and 199 dots down. With the **VIEW** and **WINDOW** settings used in lines 1020 and 1030, the **PSET (50, 50)** statement plots a point in the center of the graphics region (in this case, the entire screen). Since the first (**X**) coordinate of the center of the screen is 160 in medium-resolution graphics, the value 160 will be assigned to **X** in line 1050. Each character is 8 dots wide, so the column location of the text position is obtained by dividing the **X** coordinate (160) by 8. The **INT** function in line 1070 is then used to obtain an integer value for the text column. The text row is found in a similar manner using **POINT(1)** to obtain the second coordinate. The **LOCATE ROW, COL** statement in line 1090 places the text cursor at row 12, column 20. The word **DOT** will be printed there.

You need to adjust the text row and column calculations if the **VIEW** setting is not the standard screen. For example, suppose the (standard) **VIEW** setting in the above program were changed to

```

1020   VIEW (170, 90) - (270, 190), , 1

```

If so, you would need to change lines 1070 and 1080 as follows in order to place the label above the plotted point:

```

1070   LET COL = INT((X + 170) / 8)
1080   LET ROW = INT((Y + 90) / 8)

```

Because **POINT(0)** and **POINT(1)** return the first and second coordinates with

respect to the graphics region, an adjustment is required. You can see that the adjustment needed is the addition of 170 to **X** and 90 to **Y** and that these values give the location of the upper left corner of the graphics region defined in the **VIEW** statement.

- **POINT(0) and POINT(1) return physical coordinates of the graphics cursor.**

If you wish to place the word **DOT** below the point, a further adjustment is needed. To move **DOT** below the point, you move the text cursor down two rows. The following modification of line 1080 will do this:

```
1080      LET ROW = INT((Y + 90) / 8) + 2
```

- **LOCATE and VIEW have their starting point in the upper left corner.**

Labels are easier to place outside a graphics region than inside. Well placed labels can greatly increase the attractiveness of your graphics displays.

## 12—4 WORKING WITH PROGRAM EXAMPLES

### Example 1 - Budget Bar Chart

Often, budget items are displayed graphically as a bar chart. Suppose you want to display a bar chart for the household budget given in the following table:

Item	Budgeted Amount
1. Energy	\$1300
2. Auto	3000
3. Food	5000
4. Mortgage	7200
5. Clothing	1500
6. Miscellaneous	1000

The main routine resulting from an outline might look like this:

```
1000 'PROGRAM BUDGET BAR CHART
1010   GOSUB 2000 'Set screen
1020   GOSUB 3000 'Load array
1030   GOSUB 4000 'Compute percentages
1040   GOSUB 5000 'Draw bars
1050   GOSUB 6000 'Label bars
1060   GOSUB 7000 'Display table
1070   LOCATE 1, 1
1080   INPUT "PRESS RETURN TO END."; ANS$
1090   END
```

The subroutine that sets the screen is routine. The graphics region is placed on the left of the screen to leave room for the table at the right of the screen. The subroutine follows:

```

2000 'SUB SET SCREEN
2010     KEY OFF
2020     CLS
2030     SCREEN 1
2040     VIEW (0, 30) - (140, 170)
2050     WINDOW (0, 0) - (120, 100)
2060     RETURN

```

The following subroutine loads the budget data from **DATA** statements into arrays.

```

3000 'SUB LOAD ARRAYS
3010     READ N
3020     DIM BUDGET$(N), BUDGET(N)
3030     FOR K = 1 TO N
3040         READ BUDGET$(K), BUDGET(K)
3050     NEXT K
3060     DATA 6
3070     DATA "ENERGY", 1300
3080     DATA "AUTO", 3000
3090     DATA "FOOD", 5000
3100     DATA "MORTGAGE", 7200
3110     DATA "CLOTHING", 1500
3120     DATA "MISC.", 1000
3130     RETURN

```

You can compute the percentages for each item in your budget by summing the item dollar amounts and then dividing each item dollar amount by this sum. The percentage for each item is then obtained by multiplying these quotients by 100. The subroutine to accomplish this task follows:

```

4000 'SUB COMPUTE PERCENTAGES
4010     DIM PERCENT(N)
4020     LET SUM = 0
4030     FOR K = 1 TO N
4040         LET SUM = SUM + BUDGET(K)
4050     NEXT K
4060     FOR K = 1 TO N
4070         LET PERCENT(K) = (BUDGET(K) / SUM) * 100
4080     NEXT K
4090     RETURN

```

To draw the bars representing the percent of the total that each item represents, you can use the **LINE** statement. The **BARWIDTH** is based on the **WINDOW** statement in the **SET SCREEN** subroutine, where the first coordinates are restricted to numbers between 0 and 120. You can then use the **BARWIDTH** and the item

percents to determine the corners of the boxes drawn by the **LINE** statement. The subroutine follows:

```

5000 'SUB DRAW BARS
5010   LET BARWIDTH = 120 / N
5020   LET K = 1 'First bar
5030   LINE (0, 0) - (BARWIDTH, PERCENT(1)), 1, B
5040   FOR K = 2 TO N 'Other bars
5050     LINE ((K-1) * BARWIDTH, 0) - (K * BARWIDTH,
           PERCENT(K)), 1, B
5060   NEXT K
5070   RETURN

```

The final two subroutines use the **LOCATE** statement to label the bar chart and display the table that associates the labels with the budget items. These subroutines follow:

```

6000 'SUB LABEL BARS
6010   FOR K = 1 TO N
6020     LOCATE 23, K * 3 - 2
6030     PRINT K;
6040   NEXT K
6050   RETURN

7000 'SUB DISPLAY TABLE
7010   FOR K = 1 TO N
7020     LOCATE K + 15, 22
7030     PRINT K; " "; BUDGET$(K)
7040   NEXT K
7050   RETURN

```

The complete program follows:

```

1000 'PROGRAM BUDGET BAR CHART
1010   GOSUB 2000 'Set screen
1020   GOSUB 3000 'Load array
1030   GOSUB 4000 'Compute percentages
1040   GOSUB 5000 'Draw bars
1050   GOSUB 6000 'Label bars
1060   GOSUB 7000 'Display table
1070   LOCATE 1, 1
1080   INPUT "PRESS RETURN TO END."; ANS$
1090   END
1990 '
2000 'SUB SET SCREEN
2010   KEY OFF
2020   CLS
2030   SCREEN 1
2040   VIEW (0, 30) - (140, 170)
2050   WINDOW (0, 0) - (120, 100)
2060   RETURN

```

```

2990 '
3000 'SUB LOAD ARRAYS
3010     READ N
3020     DIM BUDGET$(N), BUDGET(N)
3030     FOR K = 1 TO N
3040         READ BUDGET$(K), BUDGET(K)
3050     NEXT K
3060     DATA 6
3070     DATA "ENERGY", 1300
3080     DATA "AUTO", 3000
3090     DATA "FOOD", 5000
3100     DATA "MORTGAGE", 7200
3110     DATA "CLOTHING", 1500
3120     DATA "MISC.", 1000
3130     RETURN
3990 '
4000 'SUB COMPUTE PERCENTAGES
4010     DIM PERCENT(N)
4020     LET SUM = 0
4030     FOR K = 1 TO N
4040         LET SUM = SUM + BUDGET(K)
4050     NEXT K
4060     FOR K = 1 TO N
4070         LET PERCENT(K) = (BUDGET(K) / SUM) * 100
4080     NEXT K
4090     RETURN
4990 '
5000 'SUB DRAW BARS
5010     LET BARWIDTH = 120 / N
5020     LET K = 1 'First bar
5030     LINE (0, 0) - (BARWIDTH, PERCENT(1)), 1, B
5040     FOR K = 2 TO N 'Other bars
5050         LINE ((K-1) * BARWIDTH, 0) - (K * BARWIDTH,
            PERCENT(K)), 1, B
5060     NEXT K
5070     RETURN
5990 '
6000 'SUB LABEL BARS
6010     FOR K = 1 TO N
6020         LOCATE 23, K * 3 - 2
6030         PRINT K;
6040     NEXT K
6050     RETURN
6990 '
7000 'SUB DISPLAY TABLE
7010     FOR K = 1 TO N
7020         LOCATE K + 15, 22
7030         PRINT K; " "; BUDGET$(K)
7040     NEXT K
7050     RETURN

```



You may wish to enter and run this program.

### Example 2 - Budget Pie Chart

You can modify the program in Example 1 to display the budget information as a pie (circular) chart rather than a bar chart. If you consider the main routine of the BUDGET BAR CHART program below,

```

1000 'PROGRAM BUDGET BAR CHART
1010   GOSUB 2000 'Set screen
1020   GOSUB 3000 'Load array
1030   GOSUB 4000 'Compute percentages
1040   GOSUB 5000 'Draw bars
1050   GOSUB 6000 'Label bars
1060   GOSUB 7000 'Display table
1070   LOCATE 1, 1
1080   INPUT "PRESS RETURN TO END."; ANS$
1090   END

```

you can see that subroutines 5000 and 6000 need to be modified. The main routine for a pie chart program would look like this:

```

1000 'PROGRAM BUDGET PIE CHART
1010   GOSUB 2000 'Set screen
1020   GOSUB 3000 'Load array
1030   GOSUB 4000 'Compute percentages
1040   GOSUB 5000 'Draw pie slices
1050   GOSUB 6000 'Label pie slices
1060   GOSUB 7000 'Display table
1070   LOCATE 1, 1
1080   INPUT "PRESS RETURN TO END."; ANS$
1090   END

```

To simplify the necessary modifications in these subroutines, you could change the WINDOW statement in the SET SCREEN subroutine as follows:

```

2050   WINDOW (-50, -50) - (50, 50)

```

The modified subroutine 5000 could look like this:

```

5000 'SUB DRAW PIE SLICES
5010   GOSUB 5500 'Convert percent to circle numbers
5020   CIRCLE (0, 0), 50, 1, 0, NUMBER(1)
5030   FOR K = 2 TO N
5040     CIRCLE (0, 0), 50, 1, -NUMBER(K-1), -NUMBER(K)
5050   NEXT K
5060   RETURN
5490 '
5500 'SUB CONVERT PERCENT TO CIRCLE NUMBERS
5510   LET NUMBER(1) = (PERCENT(1) / 100) * 2 * 3.14159
5520   FOR K = 2 TO N

```

```

5530      LET NUMBER(K)=(PERCENT(K)/100)*2*3.14159
          +NUMBER(K-1)
5540      NEXT K
5550      RETURN

```

You need to convert the percents to the circle numbers that will be used to draw each pie slice. You do this by converting the percents to fractions which are then multiplied by a full circle angle ( $2 * 3.14159$ ) in radians. Since the pie slices accumulate around the circle, you must build each circle number on the previous one. You do this by adding `NUMBER(K - 1)` in line 5530. These numbers are used to draw the pie slices in line 5040. The first action of the `DRAW PIE SLICES` subroutine is to call the subroutine that does this conversion. The pie slices are then drawn by the `CIRCLE` statements in lines 5020 and 5040.

The labeling subroutine (6000) is modified as follows:

```

6000 'SUB LABEL SLICES
6010   LET K = 1 'First slice
6020   LET ANGLE = NUMBER(1) / 2
6030   GOSUB 6500 'Place label
6040   FOR K = 2 TO N 'Other slices
6050     LET ANGLE = NUMBER(K - 1) + (NUMBER(K) -
        NUMBER(K - 1)) / 2
6060     GOSUB 6500 'Place label
6070   NEXT K
6080   RETURN
6490 '
6500 'SUB PLACE LABEL
6510   PSET (36 * COS(ANGLE), 36 * SIN(ANGLE)), 0
6520   LET X = POINT(0)
6530   LET Y = POINT(1)
6540   LET COL = INT((X + 0) / 8)
6550   LET ROW = INT((Y + 30) / 8) + 1
6560   LOCATE ROW, COL
6570   PRINT K
6580   RETURN

```

You can number the slices by locating the text cursor near the middle of each slice. In lines 6020 and 6050, `ANGLE` is computed to be an angle of radian measure that is half way inside a slice. The `PLACE LABEL` subroutine (6500) is called to print the number in the slice. Line 6510 uses the built-in functions `COS` and `SIN` which return the first and second coordinates of a point on a circle of radius 1. These functions use the argument `ANGLE` to compute the coordinates of the point associated with a pie slice. You then use `PSET` to move the graphics cursor into a desirable position in the slice by multiplying these coordinates by 36.

As you saw in the Discovery Exercises, the remainder of the subroutine locates the text cursor near this position in a slice and prints the number of the slice. Because the text cursor cannot be located precisely, you may need to experiment with the row and column positions to place the labels in the best position.

You may wish to make these modifications and run the revised program.

**Example 3 - Function Plotting**

Analysts and researchers often need to graph data and functions in their work. This program graphs the sine function, a common built-in trigonometric function. You select the range of values for which the function will be graphed and the number of points used. Two arrays, **FX** and **FY** are used – one for the first coordinates and one for the second coordinates of the points to be plotted. The arrays are loaded using a **FOR/NEXT** loop in the **LOAD FUNCTION ARRAYS** subroutine. The complete program follows:

```

1000 'PROGRAM FUNCTION PLOTTING
1010   DEF FN F(X) = SIN(X)
1020   GOSUB 2000 'Set up graphics
1030   GOSUB 3000 'Load function arrays
1040   GOSUB 4000 'Find max function value
1050   WINDOW (A, -M) - (B, M) 'Set coordinate limits
1060   GOSUB 5000 'Draw axes
1070   GOSUB 6000 'Draw graph of function
1080   LOCATE 4, 1
1090   INPUT "PRESS RETURN TO END."; ANS$
1100   END
1990 '
2000 'SUB SET GRAPHICS
2010   KEY OFF
2020   CLS
2030   SCREEN 1
2040   VIEW (90, 40) - (225, 140)
2050   RETURN
2990 '
3000 'SUB LOAD FUNCTION ARRAYS
3010   INPUT "ENTER NUMBER OF POINTS "; N
3020   DIM FX(N), FY(N)
3030   INPUT "ENTER MIN AND MAX X VALUES: "; A, B
3040   FOR K = 1 TO N
3050     LET FX(K) = A + ((B - A) / N) * K
3060     LET FY(K) = FN F(A + ((B - A) / N) * K)
3070   NEXT K
3080   RETURN
3990 '
4000 'SUB FIND MAX FUNCTION VALUE
4010   LET M = FY(1)
4020   FOR K = 1 TO N - 1
4030     IF M < ABS(FY(K)) THEN 4040 ELSE 4060
4040     'THEN CLAUSE
4050     LET M = ABS(FY(K))
4060   'END IF
4070   NEXT K
4080   RETURN
4990 '
5000 'SUB DRAW AXES

```

```

5010    LINE (0, -M) - (0, M)
5020    DRAW "NG5 NF5" 'Arrow head
5030    LINE (A, 0) - (B, 0)
5040    DRAW "NH5 NG5" 'Arrow head
5050    LOCATE 3, 1
5060    PRINT "Y-AXIS: "; -M; " TO "; M
5070    LOCATE 6, 9
5080    PRINT INT(M + .1)
5090    RETURN
5990 '
6000 'SUB DRAW GRAPH OF FUNCTION
6010    PSET (FX(1), FY(1))
6020    FOR K = 2 TO N
6030        LINE -(FX(K), FY(K))
6040    NEXT K
6050    RETURN

```

You can easily change the program to graph any function you wish by editing line 1010.

#### Example 4 - Zooming

The computer can act like a zoom lens on a camera. You create a zooming effect by successively decreasing the **WINDOW** coordinates while drawing a circle of constant radius (50). A short program that shows this effect follows:

```

1000 'PROGRAM ZOOMING
1010    GOSUB 2000 'Set up graphics region
1020    FOR X = 1000 TO 100 STEP -50
1030        WINDOW (-X, -X) - (X, X)
1040        CIRCLE (0, 0), 50
1050        CLS
1060    NEXT X
1070    CIRCLE (0, 0), 50
1080    END
1990 '
2000 'SUB SET UP GRAPHICS REGION
2010    KEY OFF
2020    CLS
2030    SCREEN 1
2040    VIEW (0, 0) - (319, 199)
2050    RETURN

```

The **CIRCLE** statement in line 1070 draws a final circle that is not erased. You may wish to enter and run the program to see how the zooming effect works.

#### Example 5 - House Design

This program draws a house. The house frame and roof are drawn first. You then decide whether to have windows in the roof. You are also given the choice of placing windows in the frame of the house. Finally, you are asked to choose the



position of the door. The **SET GRAPHICS POSITION** subroutine beginning in line 8000 uses the arrow keys to allow you to select the position of the windows and the door. You press **D** to draw the window(s) and door after you select their positions. This subroutine is quite similar to the **SET GRAPHICS POSITION** subroutine in the final version of the **SHAPE ARRAY** program in the Discovery Exercises. The complete program follows.

```

1000 'PROGRAM HOUSE DESIGN
1010   GOSUB 2000 'Set up screen
1020   GOSUB 3000 'Set scale
1030   GOSUB 4000 'Draw frame and roof
1040   GOSUB 5000 'Draw roof windows
1050   GOSUB 6000 'Draw frame windows
1060   GOSUB 7000 'Draw door
1070 END
1990 '
2000 'SUB SET UP SCREEN
2010   KEY OFF
2020   CLS
2030   SCREEN 1
2040   VIEW (85, 25) - (235, 175)
2050   WINDOW (0, 0) - (200, 200)
2060   RETURN
2990 '
3000 'SUB SET SCALE
3010   GOSUB 9000 'Clear & set text position
3020   PRINT "ENTER A DECIMAL NUMBER BETWEEN "
3030   PRINT "1 AND 3 FOR THE SCALING FACTOR: ";
3040   INPUT S
3050   RETURN
3990 '
4000 'SUB DRAW FRAME & ROOF
4010   LINE (0, 0) - (60 * S, 40 * S), , B
4020   PSET (0, 40 * S)
4030   LINE (0, 40 * S) - (30 * S, 60 * S)
4040   LINE (30 * S, 60 * S) - (60 * S, 40 * S)
4050   RETURN
4990 '
5000 'SUB DRAW ROOF WINDOWS
5010   'LOOP
5020   GOSUB 9000 'Clear & set text position
5030   INPUT "DO YOU WANT A ROOF WINDOW?(Y/N) ";ANS$
5040   IF ANS$ = "N" THEN 5100
5050   LET A = 30 * S
5060   LET B = 50 * S
5070   GOSUB 8000 'Set graphics position
5080   LINE (A, B) - (A + 6 * S, B + 10 * S), , B
5090   GOTO 5010
5100   'END LOOP
5110   RETURN

```



```

5990 '
6000 'SUB DRAW FRAME WINDOWS
6010 'LOOP
6020     GOSUB 9000 'Clear & set text position
6030     INPUT "DO YOU WANT A FRAME WINDOW?(Y/N) ";ANS$
6040     IF ANS$ = "N" THEN 6100
6050     LET A = 30 * S
6060     LET B = 20 * S
6070     GOSUB 8000 'Set graphics position
6080     LINE (A, B) - (A + 6 * S, B + 10 * S), , B
6090     GOTO 6010
6100 'END LOOP
6110 RETURN
6990 '
7000 'SUB DRAW DOOR
7010     GOSUB 9000 'Clear & set text position
7020     PRINT "POSITION THE DOOR, PLEASE."
7030     LET A = 30 * S
7040     LET B = 20 * S
7050     GOSUB 8000 'Set graphics position
7060     LINE (A, B) - (A + 6 * S, B + 15 * S), , B
7070 RETURN
7990 '
8000 'SUB SET GRAPHICS POSITION
8010     PSET (A, B)
8020 'LOOP
8030     LET A$ = "" 'Initialize A$
8040     GOSUB 8500 'Press a key
8050     LET LASTA = A
8060     LET LASTB = B
8070 'CASE
8080     IF MID$(A$, 2, 1) = "K" THEN LET A = A - 5
8090     IF MID$(A$, 2, 1) = "M" THEN LET A = A + 5
8100     IF MID$(A$, 2, 1) = "H" THEN LET B = B + 5
8110     IF MID$(A$, 2, 1) = "P" THEN LET B = B - 5
8120 'END CASE
8130     PSET (LASTA, LASTB), 0
8140     PSET (A, B)
8150     IF A$ = "D" THEN 8170
8160     GOTO 8030
8170 'END LOOP
8180 RETURN
8490 '
8500 'SUB PRESS A KEY
8510 'LOOP
8520     IF A$ <> "" THEN 8550
8530     LET A$ = INKEY$
8540     GOTO 8510
8550 'END LOOP
8560 RETURN

```

```

8990 '
9000 'SUB CLEAR & SET TEXT POSITION
9010     LOCATE 1, 1
9020     FOR K = 1 TO 120 'Clear 3 lines of 40 spaces
9030         PRINT " ";
9040     NEXT K
9050     LOCATE 1, 1
9060     RETURN

```

When you enter and run this program, be sure the Caps Lock key is activated so that all the letters you type are capitalized.

## 12—5 PROBLEMS

1. Look at Example 2, "Budget Pie Chart." Modify the program so that the second sector is drawn with a radius twice as large as the radius of the circle. You will need to change the radius in the two **CIRCLE** statements in the **DRAW PIE SLICES** subroutine from 50 to 25 so that the large slice will be within the graphics region.
2. Modify the "Budget Pie Chart" program so that any sector can be enlarged, as in problem 1.
3. Look at Example 3, "Function Plotting." Modify the program to graph the function  $2\sin(x)$ .
4. Modify the "Function Plotting" program to graph the two functions  $\sin(x)$  and  $\cos(x)$  in two adjacent viewports.
5. Modify the "Function Plotting" program to graph the two function  $\sin(x)$  and  $\cos(x)$  together.
6. Change Example 4, the "Zooming" program. Modify the program so that the circle starts out large and becomes smaller and smaller.
7. Look at Example 5, the "House Design" program. Modify the program to draw the windows wider than they are long.
8. Modify the "House Design" program to position the door in the same place every time the program is run.
9. Write a program to draw a picture with 50 randomly positioned squares with sides of random integer lengths between 5 and 20. Use a 100-by-100 window.
10. Write a program with 50 randomly positioned circles of random radii between 4 and 10. Use a graphics regions in the lower right quarter of the screen and a 100x100 window.

## 12—6 PRACTICE TEST

1. Where will the following **SCREEN** and **VIEW** statements position the graphics region?

```

SCREEN 1
VIEW (0, 100) - (100, 199)

```

---

2. Write a **WINDOW** statement that will allow points with first coordinates between -10 and 10 and second coordinates between 5 and 50 to be plotted in the graphics region.
- 

3. If the **VIEW** statement is omitted from a program, what graphics region is available?
- 

4. What will be drawn when the following program segment is run?

```
120    SCREEN 1
130    WINDOW (0, 0) - (20, 20)
140    LET FACTOR = -.25 * 2 * 3.14159
150    CIRCLE (10,10), 5, 1, FACTOR, 2 * FACTOR
```

---

5. Where will the words "I CAN PROGRAM NOW!" be placed when the following program segment is run?

```
240    SCREEN 1
250    LOCATE 17, 20
260    PRINT "I CAN PROGRAM NOW!"
```

---



# COMPATIBLE COMPUTERS




## **Compaq**

The Compaq computer with its MS DOS System Diskette and the Compaq version of BASICA can be used with this book. You need to note that the cursor movement keys, at the right of the keyboard, have different numbers on two of them compared to the IBM Personal Computer.

## **AT&T Personal Computer**

The AT&T Personal Computer with its MS DOS System Diskette can be used with this book. The markings on the cursor movement keys, the shift keys and the return key are slightly different from those on the IBM Personal Computer. To bring up BASICA, you must type GWBASIC rather than BASICA at the A> prompt. In addition, the lower drive is drive A and the upper drive is drive B.

## **IBM PCjr**






The IBM PCjr can be used with this book. You must have the Cartridge BASIC installed in the left slot below the disk drive. The cursor movement keys are in a slightly different location and have different markings than the IBM Personal Computer. The , , and  keys also have Backspace, Enter, and Shift, respectively, on them.




















# PRACTICE TEST SOLUTIONS

## Chapter 1

1. Press the  key.
2. Type `PRINT AUTHOR$` and press .
3. \*
4. Either enter `CLS` or hold down the  key and then press .
5. Division
6. `2` will be displayed on the screen.
7. `25 / 5 + 2` will be displayed.
8. Press the key with the left arrow and 4 on it until the cursor is on the G. Then type T and press .
9. `10` will be displayed on the screen.

## Chapter 2

1. Press .
2. The statement `PRINT C` has no line number.
3. The number `2` would be displayed on the screen.
4. Type the line using a line number not already in the program and press .
5. Retype the line including the line number and press .
6. Type the line number and press .
7. Enter `CLS` or press .

8.
  - a. Type **LOAD** " (name of program)" and press .
  - b. Type **SAVE** " (name of program)" and press .
  - c. Type **KILL** " (name of program).BAS" and press .
  - d. Type **NEW** and press .
  - e. Type **LIST** and press .
  - f. Type **RUN** and press .
  - g. Type **FILES** and press .
  
9.
  - a. Type **EDIT 120** and press .
  - b. Type **AUTO 100** and press .
  - c. Type **RENUM 200, ,30** and press .
  
10. Two
  
11. One more space needs to be inserted before **PRINT** in line 110.
  
12. The apostrophe (') before **PROGRAM** is missing.

### Chapter 3

1. A square with its center at the center of the screen.
2. A house
3. **PSET (160, 100)**  
**DRAW "R80 U40 L40 D40"**
4.
  - a. **PSET (160, 100)**
  - b. **PSET (0, 0)**
  - c. **PSET (160, 0)**
  - d. **PSET (240, 150)**.
5. Radius is 30, center is (140, 90).
6. **200 CIRCLE (80, 50), 40.**
7. **SCREEN 1.**
8. The letter E.

### Chapter 4

1. The **END** statement.
2. A remark statement that names the program.

3. The **RETURN** statement.
4. A remark statement naming the subroutine.
5. To separate the subroutines and improve readability.
6. Write an English-language outline of the program and create a program skeleton from the outline.
7. To show they are part of the main routine or subroutine.
8. The following will be displayed on the screen:

```
TWO
ONE
TWO
```

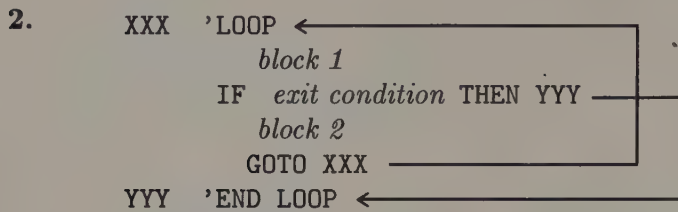
## Chapter 5

1. By using the statements **LET**, **INPUT**, and **READ/DATA**.
2. **DATA**.
3. Two.
4. As many as needed.
5. To obtain variable spacing in the output.
6. Exponentiation.
7. 0.00123.
8. 1.2345E+11.
9. Semicolons cause output to be spaced more closely than commas do.
10.
 

```
100 'PROGRAM MILES TO KILOMETERS
110   INPUT "ENTER NO. OF MILES "; M
120   LET K = 1.609 * M
130   PRINT M; " MILES EQUALS "
140   PRINT K; " KILOMETERS"
150   END
```

## Chapter 6

1. Press **Ctrl|Break**.



3. **XXX FOR A = B TO C STEP D**. The **STEP D** is needed only if the increment is not 1.

4. 1.

5. Often the final value of the loop variable is not known when the loop is entered. In cases like this, the standard form of the loop works but the **FOR/NEXT** abbreviation will not.

```

6. 100 'PROGRAM PROB. 6, PRACTICE TEST
    110 PRINT "MILES", "KILOMETERS"
    120 PRINT "-----", "-----"
    130 FOR M = 10 TO 100 STEP 5
    140 LET K = 1.609 * M
    150 PRINT M, K
    160 NEXT M
    170 END
  
```

7. A **GOTO 120** statement is missing after line 150.

8. 1  
2  
3  
2  
4  
6  
3  
6  
9

## Chapter 7

1. GOOD  
BETTER  
BEST



## 2. 100 'PROGRAM PROB 2, PRACTICE TEST

```

110 'LOOP
120     INPUT "HOW MANY WIDGETS? "; N
130     GOSUB 200 'Determine unit price
140     PRINT "PRICE PER WIDGET IS "; P
150     PRINT "TOTAL COST OF ORDER IS "; N * P
160     GOTO 110
170 'END LOOP
180 END
190 '
200 'SUB DETERMINE UNIT PRICE
210 'CASE
220     IF N <= 20 THEN LET P = 2
230     IF (N > 20) AND (N <= 50) THEN LET P = 1.8
240     IF N > 50 THEN LET P = 1.5
250 'END CASE
260 RETURN

```

## 3. The key to this solution is that the two smaller numbers will have the smaller average.

```

100 'PROGRAM PROB 3, PRACTICE TEST
110     INPUT "ENTER 3 NUMBERS "; A, B, C
120     GOSUB 200 'Find average
130     GOSUB 300 'Display average
140 END
190 '
200 'SUB FIND AVERAGE
210     LET A1 = (A + B) / 2
220     LET A2 = (A + C) / 2
230     LET A3 = (B + C) / 2
240 RETURN
290 '
300 'SUB DISPLAY AVERAGE
310 'CASE
320     IF (A1 <= A2) AND (A1 <= A3) THEN PRINT A1
330     IF (A2 <= A1) AND (A2 <= A3) THEN PRINT A2
340     IF (A3 <= A1) AND (A3 <= A2) THEN PRINT A3
350 'END CASE
360 RETURN

```

## 4. 100 'PROGRAM PROB 4, PRACTICE TEST

```

110     LET SUM = 0
120     LET N = 1
130     PRINT "NUMBER", "SUM"
140 'LOOP
150     LET SUM = SUM + N
160     IF SUM >= 100 THEN 200
170     PRINT N, SUM
180     LET N = N + 1
190     GOTO 140

```

```

200 'END LOOP
210 END

```

```

5. 100 'PROGRAM PROB 5, PRACTICE TEST
    110 INPUT "ENTER A POSITIVE WHOLE NUMBER "; N
    120 LET PRODUCT = 1
    130 LET NUMBER = 1
    140 'LOOP
    150 IF PRODUCT > 10000 OR NUMBER = N THEN 190
    160 LET NUMBER = NUMBER + 1
    170 LET PRODUCT = PRODUCT * NUMBER
    180 GOTO 140
    190 'END LOOP
    200 'CASE
    210 IF PRODUCT <= 10000 THEN PRINT PRODUCT
    220 IF PRODUCT > 10000 THEN PRINT "PROBLEM ABANDONED"
    230 'END CASE
    240 END

```

## Chapter 8

1. a. 6   b. 7   c. 22.8   d. -1
2. By appending \$ to a variable name.
3. False
4. MID\$(A\$,5,9)
5.

```

100 'PROGRAM PROB 4, PRACTICE TEST
110 INPUT "ENTER A STRING "; A$
120 FOR X = LEN(A$) TO 1 STEP -1
130 PRINT MID$(A$,1,X)
140 NEXT X
150 END

```

## Chapter 9

1. To specify how large an array is and to save memory space for the array.
2. X(3,4).
3.

```

100 'PROGRAM PROB 3, PRACTICE TEST
110 DIM A(50)
120 INPUT "HOW MANY NUMBERS "; N
130 GOSUB 200 'Load array
140 GOSUB 300 'Sum positive numbers
150 PRINT "SUM OF POSITIVE NUMBERS IS "; S

```

```

160 END
190 '
200 'SUB LOAD ARRAY
210 PRINT "WHAT ARE THE NUMBERS"
220 FOR K = 1 TO N
230 PRINT K,
240 INPUT A(K)
250 NEXT K
260 RETURN
290 '
300 'SUB SUM POSITIVE NUMBERS
310 LET S = 0
320 FOR K = 1 TO N
330 IF A(K) > 0 THEN 340 ELSE 360
340 'THEN CLAUSE
350 LET S = S + A(K)
360 'END IF
370 NEXT K
380 RETURN

```

4. 100 'PROGRAM PROB 4, PRACTICE TEST  
 110 DIM X(4,6)  
 120 FOR R = 1 TO 4  
 130 FOR C = 1 TO 6  
 140 LET X(R,C) = 4  
 150 NEXT C  
 160 NEXT R  
 170 END

5. 2 0 0 0 0  
 0 2 0 0 0  
 0 0 2 0 0  
 0 0 0 2 0  
 0 0 0 0 2

6. a. DIM A(2,3), b.  $A(2,3) = 4$ , c.  $A(X,Y) = A(1,2) = 3$ ,  
 d.  $A(A(1,1), A(2,2)) = A(1,2) = 3$ .

## Chapter 10

1. 100 'PROGRAM PROB 1, PRACTICE TEST  
 110 RANDOMIZE(TIMER)  
 120 FOR K = 1 TO 100  
 130 LET X = INT(4 \* RND) + 1  
 140 PRINT X;  
 150 NEXT K  
 160 END

```

2. 100 'PROGRAM PROB 2, PRACTICE TEST
    110 RANDOMIZE(TIMER)
    120 FOR K = 1 TO 100
    130 LET X = 25 + 25 * RND
    140 PRINT X,
    150 NEXT K
    160 END

```

3. The output will be randomly selected from **WHITE** and **RED**. Three program outputs are shown to indicate the random nature of the process.

(1)	(2)	(3)
RED	WHITE	WHITE
RED	WHITE	RED
WHITE	RED	WHITE
WHITE	WHITE	WHITE
RED	WHITE	WHITE
RED	RED	WHITE
RED	RED	RED
WHITE	RED	WHITE
RED	WHITE	WHITE
RED	WHITE	RED

4. Five random numbers of the form **X.XX** over the range **0.00** to **9.99**. Three program outputs are shown below to illustrate the random nature of the process:

(1)	(2)	(3)
0.51	6.69	1.15
9.34	4.04	8.87
9.08	9.06	9.26
9.26	6.71	2.59
5.98	8.15	3.05

## Chapter 11

- a. FILETWO, b. 70, c. 2, d. 5.
- 130 OPEN "INVENTORY" AS #1 LEN = 45
- 100 'PROGRAM PROB 3, PRACTICE TEST
 

```

110 OPEN "TEST1" AS #1 LEN = 50
120 GET #1, 75
130 INPUT A$, B$, C$, D$, E$
140 CLOSE #1
150 END

```

4. The correct program line is:  
**200 OPEN "FILEONE" AS #1 LEN = 70**
5. Line 210 should be:  
**210 GET #1, 5**
6. In positions 15 through 44. In addition, if the string assigned to **BOOK\$** is longer than 30 characters, only the first 30 characters will be used.

## Chapter 12

1. The lower left quarter of the screen.
2. **150 WINDOW (-10, 5) - (10, 50)**
3. The entire screen.
4. A quarter of a circle whose center is at the center of the graphics region. The orientation of the circle is up and to the left of the center.
5. Near the bottom right of the screen. Actually, the I will be located at the 17th row and 20th column of the text screen.





# SOLUTIONS TO ODD-NUMBERED PROBLEMS

## Chapter 4

1. 

```
100 'PROGRAM CHAP 4, PROB 1
110   GOSUB 200 'Draw border
120   GOSUB 300 'Draw cross
130 END
190 '
200 'SUB DRAW BORDER
210   CLS
220   SCREEN 1
230   LINE (0, 0) - (319, 199), 1, B
240 RETURN
290 '
300 'SUB DRAW CROSS
310   PSET (160, 100)
320   DRAW "NU80 ND80 NL80 NR80"
330 RETURN
```
3. 

```
100 'PROGRAM CHAP 4, PROB 3
110   GOSUB 200 'Draw car body
120   GOSUB 300 'Draw wheels
130 END
190 '
200 'SUB DRAW CAR BODY
210   CLS
220   SCREEN 1
230   PSET (40, 20)
240   DRAW "ND110 R120 F40 R80 D70"
250   DRAW "L20 BL50 L100 BL50 L20"
260 RETURN
290 '
300 'SUB DRAW WHEELS
310   CIRCLE (85, 120), 25
320   CIRCLE (235, 120), 25
330 RETURN
```

## Chapter 5

5. 100 'PROGRAM CHAP 5, PROB 1  
 110 READ A, B, C, D  
 120 DATA 10, 9, 1, 2  
 130 LET S = A + B  
 140 LET P = A \* B \* C \* D  
 150 PRINT S, P  
 160 END
  
7. 100 'PROGRAM CHAP 5, PROB 3  
 110 READ A, B, C, D  
 120 DATA 21, 18, 6, 3  
 130 PRINT A  
 140 PRINT B  
 150 PRINT C  
 160 PRINT D  
 170 END
  
9. 100 'PROGRAM CHAP 5, PROB 5  
 110 INPUT "LENGTH (CM) "; L  
 120 INPUT "WIDTH (CM) "; W  
 130 INPUT "HEIGHT (CM) "; H  
 140 PRINT "VOLUME IS "; L \* W \* H; "CUBIC CM"  
 150 END
  
11. 100 'PROGRAM CHAP 5, PROB 7  
 110 READ A, B, C, D, E, F, G, H, I, J  
 120 DATA 92, 63, 75, 82, 72, 53, 100, 89, 70, 81  
 130 LET GRADE = (A + B + C + D + E + F + G + H + I + J)/10  
 140 PRINT GRADE  
 150 END
  
13. 100 'PROGRAM CHAP 5, PROB 9  
 110 INPUT "QUOTED INTEREST RATE (%)" ; R  
 120 INPUT "NO. OF TIMES COMPOUNDED PER YEAR" ; M  
 130 LET T = ((1 + R / (100 \* M)) ^ M - 1) \* 100  
 140 PRINT "TRUE ANNUAL INTEREST RATE IS" ; T  
 150 END
  
15. 100 'PROGRAM CHAP 5, PROB 11  
 110 INPUT "INITIAL INVESTMENT" ; P  
 120 INPUT "ANNUAL INTEREST RATE (%)" ; I  
 130 INPUT "YEARS LEFT TO ACCRUE INTEREST" ; N  
 140 LET T = P \* (1 + I / 100) ^ N  
 150 PRINT "TOTAL VALUE IS" ; T  
 160 END

## Chapter 6

1. 

```

100 'PROGRAM CHAP 6, PROB 1
110   PRINT "FAHRENHEIT", "CELCIUS"
120   FOR F = 0 TO 100 STEP 5
130     LET C = (5 / 9) * (F - 32)
140     PRINT F, C
150   NEXT F
160 END
```
  
3. A vertical column of numbers beginning at 50, ending at 5, and spaced 5 apart.
  
5. 

```

100 'PROGRAM CHAP 6, PROB 5
110   INPUT "ENTER A WHOLE NO. > 1 "; N
120   GOSUB 200 'Compute factorial
130   PRINT N; " FACTORIAL IS "; F
140 END
190 '
200 'SUB COMPUTE FACTORIAL
210   LET F = 1
220   FOR K = 2 TO N
230     LET F = F * K
240   NEXT K
250 RETURN
```
  
7. 

```

100 'PROGRAM CHAP 6, PROB 7
110   PRINT "INCHES", "CENTIMETERS"
120   GOSUB 200 'Compute & print centimeters
130 END
190 '
200 'SUB COMPUTE & PRINT CENTIMETERS
210   FOR INCH = 0 TO 10 STEP 0.5
220     LET CM = 2.54 * INCH
230     PRINT INCH, CM
240   NEXT INCH
250 RETURN
```

## Chapter 7

1. 

```

100 'PROGRAM CHAP 7 PROB 1
110   INPUT "ENTER TWO NUMBERS "; X, Y
120   'CASE
130   IF X >= Y THEN PRINT X
140   IF Y > X THEN PRINT Y
150   'END CASE
160 END
```

3. 100 'PROGRAM CHAP 7, PROB 3  
 110 LET SUM = 0  
 120 'LOOP  
 130 READ X  
 140 IF X = 999 THEN 170  
 150 GOSUB 300 'Sum desired numbers  
 160 GOTO 120  
 170 'END LOOP  
 180 DATA -1, 22, 17, -6, 4, 7, 999  
 190 PRINT SUM  
 200 END  
 290 '  
 300 'SUB SUM DESIRED NUMBERS  
 310 IF X >= -10 AND X <= 10 THEN 320 ELSE 340  
 320 'THEN CLAUSE  
 330 LET SUM = SUM + X  
 340 'END IF  
 350 RETURN
5. 100 'PROGRAM CHAP 7, PROB 5  
 110 FOR K = 1 TO 4  
 120 LET OLD = 0  
 130 LET SUM = 0  
 140 READ EPSILON  
 150 DATA .001, .0001, .00001, .000001  
 160 GOSUB 300 'Sum the series  
 170 PRINT SUM  
 180 NEXT K  
 190 END  
 290 '  
 300 'SUB SUM THE SERIES  
 310 LET N = 1  
 320 'LOOP  
 330 LET SUM = SUM + (1 / N) ^ 2  
 340 IF (SUM - OLD) < EPSILON THEN 380  
 350 LET OLD = SUM  
 360 LET N = N + 1  
 370 GOTO 320  
 380 'END LOOP  
 390 RETURN
7. 100 'PROGRAM CHAP 7, PROB 7  
 110 INPUT "ENTER TWO NUMBERS "; X, Y  
 120 'CASE  
 130 IF (X >= 10) AND (Y >= 10) THEN PRINT X + Y  
 140 IF (X < 10) AND (Y < 10) THEN PRINT X \* Y  
 150 IF (X >= 10) AND (Y < 10) THEN PRINT X - Y  
 160 IF (X < 10) AND (Y >= 10) THEN PRINT Y - X  
 170 'END CASE  
 180 END



```

9. 100 'PROGRAM CHAP 7, PROB 9
    110   GOSUB 200 'Initialize counters
    120   GOSUB 400 'Count the integers
    130   GOSUB 600 'Print the results
    140   END
    190 '
    200 'SUB INITIALIZE COUNTERS
    210   LET ONE = 0
    220   LET TWO = 0
    230   LET THREE = 0
    240   LET FOUR = 0
    250   RETURN
    290 '
    400 'SUB COUNT THE INTEGERS
    410   'LOOP
    420     READ X
    430     IF X = 999 THEN 510
    440     'CASE
    450       IF X = 1 THEN LET ONE = ONE + 1
    460       IF X = 2 THEN LET TWO = TWO + 1
    470       IF X = 3 THEN LET THREE = THREE + 1
    480       IF X = 4 THEN LET FOUR = FOUR + 1
    490     'END CASE
    500     GOTO 410
    510   'END LOOP
    520   DATA 3, 1, 2, 1, 4, 4, 1, 2, 2, 2, 3, 999
    530   RETURN
    590 '
    600 'SUB PRINT THE RESULTS
    610   PRINT "THERE ARE "; ONE; " ONES"
    620   PRINT "THERE ARE "; TWO; " TWOS"
    630   PRINT "THERE ARE "; THREE; " THREES"
    640   PRINT "THERE ARE "; FOUR; " FOURS"
    650   RETURN

```

## Chapter 8

```

1. 25      5      20
    109

```

```

3. 100 'PROGRAM CHAP 8, PROB 3
    110   DEF FN A(R) = 3.14159 * R ^ 2
    120   DEF FN B(R) = 4 * 3.14159 * R ^ 3 / 3
    130   GOSUB 200 'Print headings
    140   GOSUB 300 'Compute & display results
    150   END
    190 '

```

```

200 'SUB PRINT HEADINGS
210   PRINT
220   PRINT "R", "AREA OF", "VOLUME OF"
230   PRINT " ", "CIRCLE", "SPHERE"
240   PRINT
250   RETURN
290 '
300 'SUB COMPUTE & DISPLAY RESULTS
310   FOR R = 1 TO 10 STEP 0.5
320     PRINT R, FN A(R), FN B(R)
330   NEXT R
340   RETURN

```

5. The program reads each number in the **DATA** statements, rounds off each number to the nearest hundredth, and prints the results.

```

7. 100 'PROGRAM CHAP 8, PROB 7
    110   INPUT "ENTER A STRING "; A$
    120   FOR K = 1 TO LEN(A$)
    130     PRINT MID$(A$, K, 1)
    140   NEXT K
    150   END

9. 100 'PROGRAM CHAP 8, PROB 9
    110   INPUT "ENTER A STRING "; A$
    120   GOSUB 200 'Initialize counters
    130   GOSUB 300 'Count vowels
    140   GOSUB 500 'Display number of each vowel
    150   END
    190 '
    200 'SUB INITIALIZE COUNTERS
    210   LET A = 0
    220   LET E = 0
    230   LET I = 0
    240   LET O = 0
    250   LET U = 0
    260   RETURN
    290 '
    300 'SUB COUNT VOWELS
    310   FOR K = 1 TO LEN(A$)
    320     'CASE
    330       IF MID$(A$,K,1) = "A" THEN LET A = A + 1
    340       IF MID$(A$,K,1) = "E" THEN LET E = E + 1
    350       IF MID$(A$,K,1) = "I" THEN LET I = I + 1
    360       IF MID$(A$,K,1) = "O" THEN LET O = O + 1
    370       IF MID$(A$,K,1) = "U" THEN LET U = U + 1
    380     'END CASE
    390   NEXT K
    400   RETURN
    490 '

```

```

500 'SUB NUMBER OF EACH VOWEL
510 PRINT "A = "; A
520 PRINT "E = "; E
530 PRINT "I = "; I
540 PRINT "O = "; O
550 PRINT "U = "; U
560 RETURN

```

```

11. 100 'PROGRAM CHAP 8, PROB 11
110 INPUT "ENTER A STRING "; A$
120 FOR K = 1 TO LEN(A$)
130 IF MID$(A$,K,1) <> " " THEN 140 ELSE 160
140 'THEN CLAUSE
150 PRINT MID$(A$,K,1);
160 'END IF
170 NEXT K
180 END

```

```

13. 100 'PROGRAM CHAP 8, PROB 13
110 LET C = 0
120 GOSUB 200 'Count how many
130 PRINT "NUMBER OF THEs = "; C
140 END
190 '
200 'SUB COUNT HOW MANY
210 FOR K = 1 TO 5
220 INPUT "ENTER A SENTENCE "; A$
230 FOR L = 1 TO LEN(A$) - 3
240 IF MID$(A$,L,4) = "THE " THEN 250 ELSE 270
250 'THEN CLAUSE
260 LET C = C + 1
270 'END IF
280 NEXT L
290 NEXT K
300 RETURN

```

```

15. 100 'PROGRAM CHAP 8, PROB 15
110 INPUT "ENTER A STRING "; A$
120 GOSUB 200 'Count number of times
130 PRINT "NUMBER TIMES IN OCCURS IS "; C
140 END
190 '

```

```

200 'SUB COUNT NUMBER OF TIMES
210   LET C = 0
220   FOR K = 1 TO LEN(A$) - 1
230     IF MID$(A$,K,2) = "IN" THEN 240 ELSE 260
240     'THEN CLAUSE
250     LET C = C + 1
260   'END IF
270   NEXT K
280   RETURN

```

## Chapter 9

1. 

```

100 'PROGRAM CHAP 9, PROB 1
110   DIM X(20)
120   GOSUB 200 'Load array
130   GOSUB 300 'Print array
140   END
190 '
200 'SUB LOAD ARRAY
210   READ N
220   FOR K = 1 TO N
230     READ X(K)
240   NEXT K
250   DATA 12
260   DATA 2, 1, 4, 3, 2, 4, 5, 6, 3, 5, 4, 1
270   RETURN
290 '
300 'SUB PRINT ARRAY
310   FOR K = 1 TO N
320     PRINT X(K)
330   NEXT K
340   RETURN

```
  
3. 

```

100 'PROGRAM CHAP 9, PROB 3
110   DIM A(10,10)
120   GOSUB 200 'Input array
130   GOSUB 300 'Sum the main diagonal
140   PRINT "SUM OF THE MAIN DIAGONAL IS "; SUM
150   END
190 '
200 'SUB INPUT ARRAY
210   INPUT "ENTER SIZE OF ARRAY "; N
220   FOR R = 1 TO N
230     FOR C = 1 TO N
240       INPUT A(R,C)
250     NEXT C
260   NEXT R
270   RETURN
290 '

```

```

300 'SUB SUM THE MAIN DIAGONAL
310   LET SUM = 0
320   FOR K = 1 TO N
330     LET SUM = SUM + A(K,K)
340   NEXT K
350   RETURN

```

```

5. 100 'PROGRAM CHAP 9, PROB 5
110   DIM A(15,15)
120   GOSUB 200 'Input array
130   GOSUB 300 'Sum the array
140   PRINT "SUM OF ENTRIES IS "; SUM
150   END
190 '
200 'SUB INPUT ARRAY
210   INPUT "ENTER ARRAY DIMENSIONS "; M, N
220   FOR R = 1 TO M
230     FOR C = 1 TO N
240       INPUT A(R,C)
250     NEXT C
260   NEXT R
270   RETURN
290 '
300 'SUB SUM THE ARRAY
310   LET SUM = 0
320   FOR R = 1 TO M
330     FOR C = 1 TO N
340       LET SUM = SUM + A(R,C)
350     NEXT C
360   NEXT R
370   RETURN

```

7. -10

9. 16

```

11. 100 'PROGRAM CHAP 9, PROB 11
110   DIM X(100)
120   INPUT "ENTER ARRAY SIZE "; N
130   GOSUB 200 'Input array
140   GOSUB 400 'Sort array
150   GOSUB 600 'Print sorted array
160   END
190 '
200 'SUB INPUT ARRAY
210   FOR K = 1 TO N
220     INPUT X(K)
230   NEXT K
240   RETURN
390 '

```



```

400 'SUB SORT ARRAY
410 'LOOP
420     LET C = 0
430     FOR K = 1 TO N - 1
440         IF X(K) < X(K+1) THEN 450 ELSE 500
450         'THEN CLAUSE
460             LET TEMP = X(K+1)
470             LET X(K+1) = X(K)
480             LET X(K) = TEMP
490             LET C = C + 1
500         'END IF
510     NEXT K
520     IF C = 0 THEN 540
530     GOTO 410
540 'END LOOP
550 RETURN
590 '
600 'SUB PRINT SORTED ARRAY
610     FOR K = 1 TO N
620         PRINT X(K); " ";
630     NEXT K
640 RETURN

```

```

13.  1      1      1      1      1      1
      0      0      0      0      0      0
      0      0      1      1      1      1
      0      0      0      0      0      0
      0      0      0      0      1      1
      0      0      0      0      0      0

```

```

15.  100 'PROGRAM CHAP 9, PROB 15
      110 DIM X(2,5)
      120 GOSUB 200 'Load array
      130 GOSUB 300 'Print array
      140 END
      190 '
      200 'SUB LOAD ARRAY
      210     FOR R = 1 TO 2
      220         FOR C = 1 TO 5
      230             READ X(R,C)
      240         NEXT C
      250     NEXT R
      260     DATA 2, 1, 0, 5, 1
      270     DATA 3, 2, 1, 3, 1
      280 RETURN
      290 '

```

```

300 'SUB PRINT ARRAY
310   FOR R = 1 TO 2
320     FOR C = 1 TO 5
330       PRINT X(R,C); " ";
340     NEXT C
350   PRINT
360   PRINT
370 NEXT R
380 RETURN

17. 100 'PROGRAM CHAP 9, PROB 17
110   DIM X(20,20)
120   INPUT "ENTER ARRAY DIMENSIONS - ROW, COL "; M, N
130   GOSUB 200 'Input array
140   GOSUB 300 'Compute & print row sums
150   GOSUB 400 'Compute & print column products
160 END
190 '
200 'SUB INPUT ARRAY
210   FOR R = 1 TO M
220     FOR C = 1 TO N
230       INPUT X(R,C)
240     NEXT C
250   NEXT R
260 RETURN
290 '
300 'SUB COMPUTE & PRINT ROW SUMS
310   FOR R = 1 TO M
320     LET RSUM = 0
330     FOR C = 1 TO N
340       LET RSUM = RSUM + X(R,C)
350     NEXT C
360     PRINT "SUM OF ROW "; R; " IS "; RSUM
370   NEXT R
380 RETURN
390 '
400 'SUB COMPUTE & PRINT COLUMN PRODUCTS
410   PRINT
420   FOR C = 1 TO N
430     LET PCOL = 1
440     FOR R = 1 TO M
450       LET PCOL = PCOL * X(R,C)
460     NEXT R
470     PRINT "PRODUCT OF COLUMN "; C; " IS "; PCOL
480   NEXT C
490 RETURN

```

```

19. 100 'PROGRAM CHAP 9, PROB 19
    110   DIM X(4,6)
    120   GOSUB 200 'Load array
    130   GOSUB 400 'Compute & print daily sales
    140   GOSUB 600 'Compute & print weekly salesperson sales
    150   GOSUB 800 'Compute & print total weekly sales
    160   END
    190 '
    200 'SUB LOAD ARRAY
    210   FOR R = 1 TO 4
    220     FOR C = 1 TO 6
    230       READ X(R,C)
    240     NEXT C
    250   NEXT R
    260   DATA 48, 40, 73, 120, 100, 90
    270   DATA 75, 130, 90, 140, 110, 85
    280   DATA 50, 72, 140, 125, 106, 92
    290   DATA 108, 75, 92, 152, 91, 87
    300   RETURN
    390 '
    400 'SUB COMPUTE & PRINT DAILY SALES
    410   FOR C = 1 TO 6
    420     LET DSUM = 0
    430     FOR R = 1 TO 4
    440       LET DSUM = DSUM + X(R,C)
    450     NEXT R
    460     PRINT "TOTAL DAY "; C; " IS "; DSUM
    470   NEXT C
    480   PRINT
    490   RETURN
    590 '
    600 'SUB COMPUTE & PRINT WEEKLY SALESPERSON SALES
    610   FOR R = 1 TO 4
    620     LET WSUM = 0
    630     FOR C = 1 TO 6
    640       LET WSUM = WSUM + X(R,C)
    650     NEXT C
    660     PRINT "TOTAL-SALESPERSON "; R; " IS "; WSUM
    670   NEXT R
    680   PRINT
    690   RETURN
    790 '
    800 'SUB COMPUTE & PRINT TOTAL WEEKLY SALES
    810   LET TSUM = 0
    820   FOR R = 1 TO 4
    830     FOR C = 1 TO 6
    840       LET TSUM = TSUM + X(R,C)
    850     NEXT C
    860   NEXT R
    870   PRINT "TOTAL SALES FOR THE WEEK IS "; TSUM
    880   RETURN

```

```

21. 100 'PROGRAM CHAP 9, PROB 21
110   DIM P(20), X(20)
120   INPUT "HOW LONG ARE THE ARRAYS "; N
130   GOSUB 200 'Load pointer array
140   GOSUB 300 'Load number array
150   GOSUB 400 'Set pointers to numbers
160   GOSUB 600 'Print descending numbers
170   END
190 '
200 'SUB LOAD POINTER ARRAY
210   FOR K = 1 TO N
220     LET P(K) = K
230   NEXT K
240   RETURN
290 '
300 'SUB LOAD NUMBER ARRAY
310   FOR R = 1 TO N
320     INPUT X(R)
330   NEXT R
340   RETURN
390 '
400 'SUB SET POINTERS TO NUMBERS
410   'LOOP
420     LET C = 0
430     FOR K = 1 TO N - 1
440       IF X(P(K)) < X(P(K + 1)) THEN 450 ELSE 500
450       'THEN CLAUSE
460         LET TEMP = P(K)
470         LET P(K) = P(K + 1)
480         LET P(K + 1) = TEMP
490         LET C = C + 1
500       'END IF
510     NEXT K
520     IF C = 0 THEN 540
530     GOTO 410
540   'END LOOP
550   RETURN
590 '
600 'SUB PRINT DESCENDING NUMBERS
610   PRINT "P", "X", "SORTED X"
620   PRINT
630   FOR K = 1 TO N
640     PRINT P(K), X(K), X(P(K))
650   NEXT K
660   RETURN

```

## Chapter 10

1. 

```

100 'PROGRAM CHAP 10, PROB 1
110   RANDOMIZE(TIMER)
120   FOR K = 1 TO 25
130     LET DIGIT1$ = STR$(INT(10 * RND))
140     LET DIGIT2$ = STR$(INT(10 * RND))
150     PRINT DIGIT1$ + "." + MID$(DIGIT2$, 2, 1)
160   NEXT K
170 END
```
  
3. Twenty random numbers in the range 0.00 to 0.20.
  
5. 

```

100 'PROGRAM CHAP 10, PROB 5
110   DIM N(5)
120   GOSUB 200 'Load number of tosses
130   GOSUB 300 'Compute number heads & tails
140 END
190 '
200 'SUB LOAD NUMBER OF TOSSES
210   FOR K = 1 TO 5
220     READ N(K)
230   NEXT K
240   DATA 10, 50, 100, 500, 1000
250 RETURN
290 '
300 'SUB COMPUTE NUMBER HEADS & TAILS
310   RANDOMIZE(TIMER)
320   FOR K = 1 TO 5
330     LET H = 0
340     LET T = 0
350     FOR L = 1 TO N(K)
360       LET X = INT(2 * RND) + 1
370       'CASE
380         IF X = 1 THEN LET H = H + 1
390         IF X = 2 THEN LET T = T + 1
400       'END CASE
410     NEXT L
420     PRINT
430     PRINT "FOR "; N(K); " TOSSES THERE WERE"
440     PRINT H; " HEADS AND "; T; " TAILS"
450   NEXT K
460 RETURN
```



## 7. 100 'PROGRAM CHAP 10, PROB 7

```

110  RANDOMIZE(TIMER)
120  LET SUM = 0
130  FOR K = 1 TO 100
140    LET SUM = SUM + RND
150  NEXT K
160  PRINT "AVERAGE IS "; SUM / 100
170  END

```

## 9. 100 'PROGRAM CHAP 10, PROB 9

```

110  RANDOMIZE(TIMER)
120  LET M = 0
130  FOR K = 1 TO 1000
140    LET A = 60 * RND
150    LET B = 60 * RND
160    IF ABS(A - B) <= 10 THEN 170 ELSE 190
170    'THEN CLAUSE
180      LET M = M + 1
190    'END IF
200  NEXT K
210  PRINT "PROBABILITY OF A MEET IS "; M / 1000
220  END

```

## 11. 100 'PROGRAM CHAP 10, PROB 11

```

110  RANDOMIZE(TIMER)
120  FOR K = 1 TO 25
130    LET S = 0
140    FOR L = 1 TO 12
150      LET S = S + RND
160    NEXT L
170    LET R = 10 + 2 * (S - 6)
180    PRINT INT(100 * R + .5) / 100,
190  NEXT K
200  END

```

## 13. 100 'PROGRAM CHAP 10, PROB 13

```

110  DIM A(12)
120  GOSUB 200 'Zero array
130  GOSUB 300 'Throw dice
140  GOSUB 400 'Print results
150  END
190  '
200  'SUB ZERO ARRAY
210  FOR K = 2 TO 12
220    LET A(K) = 0
230  NEXT K
240  RETURN
290  '

```

```

300 'SUB THROW DICE
310     RANDOMIZE(TIMER)
320     FOR K = 1 TO 1000
330         LET SUM = INT(6 * RND) + INT(6 * RND) + 2
340         LET A(SUM) = A(SUM) + 1
350     NEXT K
360 RETURN
390 '
400 'SUB PRINT RESULTS
410     PRINT
420     PRINT "SUM", "FREQ"
430     FOR K = 2 TO 12
440         PRINT K, A(K)
450     NEXT K
460 RETURN

```

## Chapter 11

1. A possible record structure follows:

<i>Variable</i>	<i>Description</i>	<i>Approx. Length</i>
TITLE\$	<i>Title of cassette</i>	30
LABEL\$	<i>Record company</i>	20
MUSICIANS\$	<i>Name of musicians</i>	50
KIND\$	<i>Category of music</i>	15
PRICE	<i>Price of cassette</i>	5

The record length could be set at 120 or left out since the default record length is 128. You could also set each of the item lengths using the **FIELD** statement.

3. A possible record structure follows:

<i>Variable</i>	<i>Description</i>	<i>Approx. Length</i>
ITEM\$	<i>Name of item</i>	50
ROOM\$	<i>Location of item</i>	15
AMT	<i>Value of item</i>	10

Here it would be worth setting the record length at 75 to conserve space in the file. You could also set each item length using the **FIELD** statement.

5. 

```

1000 'PROGRAM CHAP 11, PROB 5
1010     CLS
1020     'LOOP
1030     INPUT "ENTER CHARGES? (Y/N)"; ANS$
1040     IF ANS$ = "N" OR ANS$ = "n" THEN 1070
1050     GOSUB 2000 'Enter data
1060     GOTO 1020
1070     'END LOOP

```

```

1080     GOSUB 3000 'Total charges of a card
1090     END
1990 '
2000 'SUB ENTER DATA
2010     GOSUB 4000 'Determine number of records
2020     LET RECNUM = RECNUM + 1
2030     GOSUB 5000 'Request data
2040     OPEN "CHARGE" AS #1
2050     WRITE #1, CARDS$, STORE$, CHARGEDATES$, DESS$, AMT
2060     PUT #1, RECNUM
2070     GOSUB 6000 'Update number of records
2080     CLOSE #1
2090     RETURN
2990 '
3000 'SUB TOTAL CHARGES OF A CARD
3010     PRINT
3020     PRINT "ENTER DONE WHEN CHARGE CARD TOTALS COMPLETE."
3030     PRINT
3040     'LOOP
3050         INPUT "ENTER NAME OF CHARGE CARD TO TOTAL: ";
            WHICHCARDS$
3060     IF WHICHCARDS$ = "DONE" THEN 3090
3070         GOSUB 7000 'Total the charges
3080         GOTO 3040
3090     'END LOOP
3100     RETURN
3990 '
4000 'SUB DETERMINE NUMBER OF RECORDS
4010     OPEN "CHARGE" AS #1
4020     IF LOF(1) = 0 THEN 4030 ELSE 4060
4030     'THEN CLAUSE
4040         LET RECNUM = 1
4050         GOTO 4090
4060     'ELSE CLAUSE
4070         GET #1, 1
4080         INPUT #1, RECNUM
4090     'END IF
4100     CLOSE #1
4110     RETURN
4990 '
5000 'SUB REQUEST DATA
5010     INPUT "NAME OF CARD: "; CARDS$
5020     INPUT "NAME OF STORE: "; STORE$
5030     INPUT "DATE OF PURCHASE: "; CHARGEDATES$
5040     INPUT "DESCRIPTION OF PURCHASE: "; DESS$
5050     INPUT "COST OF PURCHASE: "; AMT
5060     RETURN
5990 '

```

```

6000 'SUB UPDATE NUMBER OF RECORDS
6010   WRITE #1, RECNUM
6020   PUT #1, 1
6030   RETURN
6990 '
7000 'SUB SUM THE CHARGES
7010   GOSUB 4000 'Determine number of records
7020   LET SUM = 0
7030   OPEN "CHARGE" AS #1
7040   FOR K = 2 TO RECNUM
7050     GET #1, K
7060     INPUT #1, CARD$, STORE$, CHARGEDATES$, DES$, AMT
7070     IF CARD$ = WHICHCARD$ THEN 7080 ELSE 7100
7080     'THEN CLAUSE
7090     LET SUM = SUM + AMT
7100   'END IF
7110   NEXT K
7120   CLOSE #1
7130   PRINT "THE TOTAL OF "; WHICHCARD$; " IS "; SUM
7140   PRINT
7150   RETURN

```

7. You can find the total amount owed by changing lines 1074, 2014, and lines 5000 through 5100 of Example 4 (MAILLIST Program) as follows:

```

1074           IF ANS = 3 THEN GOSUB 5000 'Total the amount owed

2014   PRINT "3.  TOTAL THE AMOUNT OWED

5000 'SUB TOTAL THE AMOUNT OWED
5010   OPEN "MAILLIST" AS #1
5020   LET TOTAL = 0
5030   FOR K = 2 TO RECNUM
5040     GET #1, K
5050     INPUT #1, FIRST$, LAST$, STREETS$, CITY$, ZIP$, BAL
5060     LET TOTAL = TOTAL + BAL
5070   NEXT K
5080   CLOSE #1
5090   PRINT "TOTAL AMOUNT OWED IS "; TOTAL
5100   RETURN

```

In addition, you should delete line 5110 and lines 9200 through 9390.

## Chapter 12

1. You can solve this problem by changing lines 5020 through 5040 and adding line 5025 in the **DRAW PIE SLICES** subroutine of Example 2 as follows:

```

5020    CIRCLE (0, 0), 25, 1, 0, NUMBER(1)
5025    CIRCLE (0, 0), 50, 1, -NUMBER(1), -NUMBER(2)
5030    FOR K = 3 TO N
5040    CIRCLE (0, 0), 25, 1, -NUMBER(K-1), -NUMBER(K)

```

3. Change line 1010 as follows:

```

1010    DEF FN F(X) = 2 * SIN(X)

```

5. Modify the program by adding lines 1015, 3065, 6042, 6044, 6046, 6048, and changing line 3020 as follows:

```

1015    DEF FN G(X) = COS(X)

3020    DIM FX(N), FY(N), GY(N)

3065    LET GY(K) = FN G(A + ((B - A) / N) * K)

6042    PSET (FX(1), GY(1))
6044    FOR K = 2 TO N
6046    LINE - (FX(K), GY(K))
6048    NEXT K

```

7. The only modifications needed occur in lines 5080 and 6080 as follows

```

5080    LINE (A,B) - (A + 12 * S, B + 6 * S), , B

6080    LINE (A,B) - (A + 12 * S, B + 6 * S), , B

```

9. 100 'PROGRAM CHAP 12, PROB 9

```

110    GOSUB 200 'Set screen
120    RANDOMIZE(TIMER)
130    GOSUB 300 'Draw random squares
140    END
190 '
200 'SUB SET SCREEN
210    CLS
220    SCREEN 1
230    KEY OFF
240    VIEW (0, 0) - (319, 199)
250    WINDOW (0, 0) - (100, 100)
260    RETURN
290 '

```



```
300 'SUB DRAW RANDOM SQUARES
310   FOR K = 1 TO 50
320     LET X = INT(100 * RND)
330     LET Y = INT(100 * RND)
340     PSET (X,Y)
350     LET SIDE = INT(16 * RND) + 5
360     LINE (X, Y) - (X + SIDE, Y + SIDE), 1, B
370   NEXT K
380 RETURN
```

## GLOSSARY

- ABS(X)** A BASIC function that takes the absolute value of X. Positive values of X remain positive. Negative values of X become positive.
- Action Block** A set of BASIC statements that the computer executes in line-number order.
- Arithmetic Operators** Addition +, subtraction −, multiplication \*, division /, and exponentiation ^.
- ASC(A\$)** A BASIC function that converts the first character in A\$ to its equivalent position number in the ASCII character set.
- AUTO** A command for automatically numbering lines of a BASIC program.
- BASIC** An acronym for “Beginners All-Purpose Instruction Code”. More people know how to program computers in BASIC than any other language.
- Branch Block** A set of BASIC statements that allows the program to choose between two specified actions.
- Case Block** A set of BASIC statements that allows the program to choose from among many different specified actions.
- CHR\$(N)** A BASIC function that returns the Nth character from the ASCII character set.
- CIRCLE** A graphics mode statement that draws a circle with a specified center and radius.
- CLOSE** A statement that closes a file buffer.
- CLS** A statement that clears the text screen.
- Control Characters** These are characters typed on the keyboard while holding down the Ctrl key. They are used to send special signals to the computer.
- DATA** A statement used to hold information within a program. This information is called for with the READ statement.
- DEF** A statement used to define functions which are lengthy and will be used often in a program.
- DEL** A command that deletes a series of program lines. DEL 20-50 deletes all the program lines numbered 20 to 50.
- Deleting BASIC Statements** Type the line number of the statement to be deleted and then press ↵.
- DIM** A statement used to specify the size and reserve space for arrays.
- DIR** A command that displays the names of the files on diskette.
- Double Subscripts** Indicated within parentheses following a variable name, and separated by a comma. Used to specify a row and column number in an array.

A(3,5), for example, means the element in the two dimensional array A at row 3 and column 5.

**DRAW** A graphics mode statement that draws line segments using the graphics sublanguage commands. It allows you to draw pictures.

**E Notation** A notation used in BASIC to express either very large or very small numbers.

**EDIT** A command that displays the line specified which can then be modified.

**END** Marks the end of a BASIC program or the end of the main routine.

**FIELD** A statement that allows you to precisely define records.

**File** A collection of information.

**File Buffer** A holding area in memory used in reading to and writing from files.

**FILES** A command that lists the files on a diskette.

**FOR NEXT** Statements used in BASIC to set up loops.

**GET** A statement that reads a record from a diskette file to a file buffer. An INPUT statement can then be used to assign the contents of the file buffer to program variables.

**GOSUB** A statement used to transfer program control to a subroutine.

**GOTO** A statement that sends the computer to a specified line in the program.

**IF THEN** A conditional branch statement.

**IF THEN ELSE** A conditional branch statement that chooses between two actions.

**Infinite Loop** A set of BASIC statements that the computer executes over and over until the program is timed-out or is interrupted by the user.

**INKEY\$** A BASIC function that allows the program to read a character from the keyboard.

**INPUT** A statement that calls for input of information from the keyboard or a file buffer. May be used with the GET statment to copy information from a diskette file to program variables.

**Inserting BASIC Statements** Type in the statement using a line number not already in use.

**INT(X)** A BASIC function that takes the integer part of X. The integer part of X is defined as the first integer less than or equal to X.

**KEY** A statement that allows you to store a string in one of the keys **F1** through **F10**. Thus KEY 7, "RENUM" would store the word RENUM in key **F7** and would display RENUM on the screen when **F7** is pressed.

**KILL** A command that erases a file on a diskette. For example, KILL "AVERAGE.BAS" will erase the file called AVERAGE.BAS from the diskette in the disk drive.

**LEN(A\$)** A BASIC function used to determine the length of a string of characters. For example, if A\$ = "HOUSE" then LEN(A\$) is 5.

**LET** Identifies an assignment statement. It is always followed by a variable name, an equal sign, and a BASIC expression. The LET in the assignment statement is optional.

**LINE** A statement that draws a line between any two points on the screen. It can also draw boxes (filled in if desired).

- LIST** A command used to tell the computer to print out the program in memory.
- LOAD** A command that loads a file from a diskette to the computer memory.
- LOCATE** A statement that places the text cursor at a position you wish to display text.
- LOF** A function that returns the number of the last record in a direct access file.
- Loop Block** A set of BASIC statements that are executed over and over until an exit condition is satisfied.
- LSET** A statement that places information in the buffer from program variables. It is used when working with files.
- MID\$** A statement that returns the requested part of a given string. For example, if A\$ = "TOGETHER", then MID\$(A\$,5,3) will return the string "THE".
- NEW** A command that erases the current program in memory.
- Numeric Variable Names** BASICA on the IBM PC allows variable names up to 40 characters long. The first character must be a letter.
- OPEN** A statement that assigns a file buffer to a specified file.
- POINT** A function that returns the physical coordinates of the graphics cursor.
- PRESET** Draws a point in the background color. It can be used to erase points.
- PRINT** A statement that sends information from the computer to the screen.
- PRINT USING** A statement that prints numbers and strings in a format specified by a string containing decimals points, # signs and \ signs.
- PSET** A graphics mode statement that sets a point at a specified position on the screen.
- PUT** A statement that reads a record from a file buffer to a diskette file. Used with the WRITE statement to copy information from program variables to a diskette file.
- Random Numbers** A sequence of numbers generated by the RND function. They appear to have no pattern or relationship to one another.
- RANDOMIZE** A statement used to reseed the random number generator.
- READ** A statement that calls for input of information stored in DATA statements within the program.
- Record** A structured set of information.
- Remark Statement** Statements that are useful in naming programs and describing the actions performed by the program.
- RENUM** A command that rennumbers the program in memory.
- Replacing BASIC Statements** Retype the statement to be replaced including the line number.
- RETURN** A statement used to transfer program control back from a subroutine to the main routine.
- RND** A BASIC function used to generate random numbers.
- RUN** A command used to tell the computer to begin execution of the program in memory.
- SAVE** A command that saves a file from memory to diskette. For example, SAVE "AVERAGE" would save the program currently in memory to the diskette in drive A under the name AVERAGE.
- SCREEN** A statement that sets one of three screen modes for the IBM PC.



**SGN(X)** A BASIC function that determines the sign of X. SGN(X) is +1, 0, -1 as X is positive, zero, or negative respectively.

**Skeleton Program** A set of statements that name a program and describe the structure of the program using remark statements.

**Single Subscripts** Indicated within parentheses following a variable name. Used to specify a particular element in an array. A(6), for example, means the sixth element of the one dimensional array A.

**SQR(X)** A BASIC function that takes the square root of X. X cannot be negative.

**STR\$** A string function that returns the string representation of the numeric parameter.

**String Variable Names** BASIC string variable names are allowed to be up to 40 characters long. They must start with a letter and end with a \$ sign.

**SYSTEM** A command that ends the BASIC mode, closes all files, and returns to DOS.

**TAB** A BASIC function that will tab to a specific position when in a PRINT statement.

**Top-Down Structure** A structure that allows you to break-up a problem into manageable segments. Along with its indentation style, it makes programs easy to write, easy to read, and easy to change.

**VAL** A numeric function that converts a string representation of a number to its numeric value.

**VIEW** A statement that restricts graphics displays to a specified portion of the screen.

**WINDOW** A statement that allows you to define the range of numbers that are to be used for the horizontal and vertical coordinates of the graphics region.

**WRITE** A statement that sends information to a file buffer. A PUT statement can then be used to send the information from the file buffer to the diskette file.



# INDEX

- A Sales Chart as a String Array program, 192
- A>, 4, 7, 10
- Abbreviated branch block, 132, 136
- ABS, 156, 163
- Action block, 111
- Addition, 11, 16, 82
- An Exploration program, 73
- Apostrophe, 27, 33, 42
- Arithmetic in BASIC, 82
- Arithmetic Operators
  - order, 89-91
  - priority, 89-91
  - symbols, 16
- Arithmetic priority, 89-91
- Array Operations program, 191
- Arrays, 175, 176, 183
- ASC, 158, 163
- ASCII, 157
- ASCII character set, 158, 163
- AUTO, 24, 29
- Automobile License Fees program, 137
- Averaging the Positive and Negative Numbers in a List program, 141
- BASIC arithmetic, 89
- BASIC commands
  - AUTO, 24, 29, 78
  - EDIT, 22, 29, 31
  - FILES, 6, 7, 26, 32
  - KILL, 26, 32
  - LIST, 22, 29, 30
  - LOAD, 26, 32
  - NEW, 26, 30
  - RENUM, 25, 27, 30
  - RUN, 22, 30
  - SAVE, 26, 32
- BASIC functions
  - ABS, 156, 163
  - ASC, 158, 163
  - CHR\$, 157, 163
  - INKEY\$, 251
  - INT, 155, 163
  - LEN, 154, 163
  - MID\$, 156, 163
  - POINT, 253, 257, 258
  - RND, 159, 163, 204
  - SGN, 156, 163
  - SQR, 155, 162, 163
  - STR\$, 159, 163
  - TAB, 87, 93
  - VAL, 158, 163
- BASIC mode, 3
- BASIC parentheses, 91
- BASIC programs
  - entering and controlling, 22
  - execution, 22
  - removal, 26
  - retrieval, 26
  - storage, 26
- BASIC program requirements
  - line numbers, 22, 28
  - order, 22
  - spacing, 29
- BASIC statements
  - CIRCLE, 47, 52
  - CLOSE, 222, 228
  - CLS, 11, 16, 22
  - DATA, 80, 81, 89
  - DEF, 160, 164
  - DIM, 180, 184
  - DRAW, 40, 51, 52, 58
  - END, 22, 28, 65
  - EOF(1), 225, 230
  - FIELD, 227, 230
  - FOR NEXT, 109-111, 115, 185
  - GET, 223, 230
  - GOSUB, 59, 60, 64, 65
  - GOTO, 106, 111
  - IF THEN, 108, 109
  - IF THEN ELSE, 130, 131, 135
  - INPUT, 25, 28, 78, 88, 89, 230
  - LET, 13, 23, 28, 89
  - LINE, 48, 52
  - LOCATE, 252, 256, 258
  - LSET, 227, 231
  - OPEN, 222, 228
  - PRESET, 47, 52
  - PRINT, 10, 13, 15, 16
  - PRINT USING, 87, 94
  - PSET, 46, 152
  - PUT, 222, 229
  - READ, 80, 81, 88, 89
  - RETURN, 59, 64, 65
  - VIEW, 244, 248, 253, 254
  - WINDOW, 245, 248, 254
  - WRITE, 222, 229
  - ' 27, 33, 42
- BASIC variables, names, 13, 17, 18
- BASICA, 6, 7
- Birthday Pairs in a Crowd program, 216
- Branch block, 132, 135
- Bringing up BASICA, 9, 10
- Budget Bar Chart program, 258
- Budget Pie Chart program, 262

- Case block, 134, 136
- Catenation, 12, 25
- Central processing unit, 3
- CHR\$, 157, 163
- CIRCLE, 47, 52
- Circle program, 215
- Clearing the screen, 11, 16, 23
- CLS, 11, 16, 22
- CLOSE, 222, 228
- Commands, See BASIC commands
- Comparison operator, 112, 113
- Converting Temperature program, 96
- Correcting mistakes, 11, 18, 19, 31, 53
- Counting loop, 107, 112
- Course Grades program, 188
- Creating a working diskette, 5, 6
- Cursor, 6, 23
  
- DATA, 80, 81, 89
- DEF, 160, 164
- Defining function keys, 40, 49, 50
- Deleting characters, 18, 19, 23
- Deleting Lines, 23, 29
- Depreciation Schedule program, 121
- Dice Rolling Simulation program, 210
- DIM, 180, 184
- DIR, 5, 6, 7
- Direct mode, 15
- DISKCOPY, 5, 6
- Disk Drive, 3, 4
- Division, 12, 16, 82
- DOS System diskette, 4, 5, 6, 21
- DRAW, 40, 51, 52, 58
- Drive A, 4
- Drive B, 5
  
- EDIT, 22, 29, 31
- Editing lines, 12, 18, 19, 22, 23, 27, 29, 31, 53
- END, 22, 28, 65
- E Notation, 85, 92
- EOF(1), 225, 230
- Enter, 5, 22
- Error correction, 11, 18, 22, 31
- Exact Division program, 165
- Examination Grades program, 185
- Exiting a mode, 6, 24
- Exponentiation, 82, 83, 90
  
- FIELD, 227, 230
- File, definition, 222
- FILES, 6, 7, 26, 32
- Finding the Average of a Group of Numbers program, 116
- FOR NEXT, 109-111, 115, 185
- Function keys, 23, 24, 26, 40, 50
- Function Plotting program, 264
  
- GET, 227, 230
- GOSUB, 59, 60, 64, 65
- GOTO, 106, 111
- Graph Box program, 169
- Graphics mode, 40, 44, 50
- Graphics region, 243, 247, 253
- Graphics screens, 40, 43, 50, 51, 253
- GWBASIC, 6
  
- House Design program, 265
  
- IF THEN, 108, 109
- IF THEN ELSE, 130, 131, 135
- Infinite loop, 105, 111
- INKEY\$, 251
- INPUT, 25, 28, 78, 88, 89, 230
- Inserting characters, 12, 18, 19, 23
- Inserting lines, 29
- Insert mode, 18
- INT, 155, 163
- Interrupt, 24, 79, 88, 107
- Interrupting program executions, 107, 108
  
- Keyboard, 3
- Keys
  - Alt**, 10, 11, 16, 24, 31, 78
  - Caps Lock**, 5, 10, 22
  - Ctrl**, 11
  - CtrlBreak**, 79
  - CtrlHome**, 11, 16, 23, 31
  - Del**, 18, 19, 23, 53
  - Ins**, 12, 18, 19, 23, 53
  - Num-Lock**, 11
  - ←**, 4-6, 10, 15
  - ↑**, 42, 82
  - F1**, 23, 24, 31
  - F2**, 23, 24, 31
  - F3**, 26, 31
  - F4**, 26, 31
- Arrow keys, See end of Index
- KILL, 26, 32
  
- LEN, 154, 163
- LET, 13, 23, 28, 89
- LINE, 48, 52
- Line deletion, 29
- Line numbers, 22, 28
- LIST, 22, 29, 30
- LOAD, 26, 32
- LOCATE, 252, 256, 258
- Loop block, 111
- LSET, 227, 231

- Mail List Data Entry program, 231
- Mailing Label program, 234
- Main routine, 60, 65
- Matrix, 175, 176
- MID\$, 156, 163
- Modifying the **MailList** File program, 235
- Monitor, 3, 4, 10
- Multiplication, 12, 16, 82
  
- NEW, 26, 30
- Numeric variable definition, 17
  
- On-Off switch, 4
- OPEN, 222, 228
- Operating System, 3, 4, 7
- Operations
  - Arithmetic, 11, 12, 16
  - Relational, 112, 113
  - Order, 89-91
  - Order (of arithmetic operations), 84
  
- Parentheses, 84, 91
- PC DOS, 3, 4, 7, 9
- Phrase Generator program, 213
- Plotting Big Letters program, 69
- POINT, 253, 257, 258
- PRESET, 47, 52
- PRINT, 10, 13, 15, 16
- PRINT USING, 87, 94
- Priority of operations, 89-91
- Program mode, 28
- Program transportability, 4
- Programs in book
  - A Sales Chart as a String Array, 192
  - An Exploration, 73
  - Array Operations, 191
  - Automobile License Fees, 137
  - Averaging the Positive and Negative Numbers in a List, 141
  - Birthday Pairs in a Crowd, 216
  - Budget Bar Chart, 258
  - Budget Pie Chart, 262
  - Circles, 215
  - Converting Temperature, 96
  - Course Grades, 188
  - Depreciation Schedule, 121
  - Dice Rolling Simulation, 210
  - Exact Division, 165
  - Examination Grades, 185
  - Finding the Average of a Group of Numbers, 116
  - Function Plotting, 264
  - Graph Box, 169
  - House Design, 265
  - Mail List Data Entry, 231
  - Mailing Label, 234
  - Modifying the **MailList** File, 235
  - Phrase Generator, 213
  - Plotting Big Letters, 69
  - Random Walk, 214
  - Selected Labels program, 235
  - Shuffle and Deal, 217
  - String Reversal, 167
  - Sum and Product of Numbers, 98
  - Table of Numbers and Their Cubes, 125
  - Unit Prices, 94
  - Word Count, 168
  - Zooming, 265
- PSET, 46, 52
- PUT, 222, 229
  
- Random numbers, 203, 205, 206, 209, 210
- Random Walk program, 214
- RANDOMIZE, 159, 204, 208, 209
- READ, 80, 81, 88, 89
- Record, 222
- Relational Operators, 112, 113
- Remark statement, See apostrophe
- Removing a program in memory, 26
- Removing a program on diskette, 26
- RENUM, 25, 27, 30
- RETURN, 59, 64, 65
- Retrieving BASIC programs, 26
- RND, 159, 163, 204
- RUN, 22, 30
  
- SAVE, 26, 32
- Saving a program on diskette, 26
- Scaling figures, 45
- Screen display, 3
- Selected Labels program, 235
- SGN, 156, 163
- Shortcuts, 31
- Shuffle and Deal program, 217
- Skeleton program, 33, 42
- Spacing of printout, 86, 87, 93
- SQR, 155, 162, 163
- Standard branch block, 132, 135
- Standard loop, 108, 113
- Start-up procedure, 9, 15
- STR\$, 159, 163
- String output, 10
- String Reversal program, 167
- String variable definition, 17, 18
- Subroutine bugs, 63, 66-68
- Subroutines, 58, 60, 64, 65
- Subscripts, 176, 183
- Substrings, 156, 163
- Subtraction, 12, 16, 82
- Sum and Product of Numbers program, 98
- Syntax error, 31
- System, 6








TAB, 87, 93  
 Table of Numbers and Their Cubes  
   program, 125  
 Top-down organization, 27, 32, 34, 42, 53, 64, 68  
 Trailing remarks, 42  
 Turning on/off your computer, 4

Unit Prices program, 94  
 User-defined functions, 160, 164

VAL, 158, 163  
 Variable names, 13, 17, 18  
 Variable, numeric, 14, 18  
 Variable, string, 13, 17  
 Variables, subscripted, 176, 183  
 VIEW, 244, 248, 253, 254

WINDOW, 245, 248, 254  
 Word Count program, 168  
 Working diskette, 5, 6, 9, 21  
 Wrapping program lines, 123, 145  
 WRITE, 222, 229

Zooming program, 265

+, 11, 16, 82  
 -, 12, 16, 82  
 /, 12, 16, 82  
 \*, 12, 16, 82  
 =, 109, 112, 113  
 >, 108, 112, 113  
 <, 112, 113  
 <>, 112, 113  
 <=, 112, 113  
 >=, 112, 113  
, 4-6, 10, 15  
, 42, 82  
, 11, 18, 19, 53  
, 11, 19, 53  
, 11, 12, 19, 53  
, 12, 19, 53  
, 19, 53  
 ' 27, 33, 42  
 \, 87, 94  
 ■, 6, 23

130-0  
 82







**LIBRARY  
OF  
BIRMINGHAM-SOUTHERN  
COLLEGE**

\*P1-AEZ-393\*