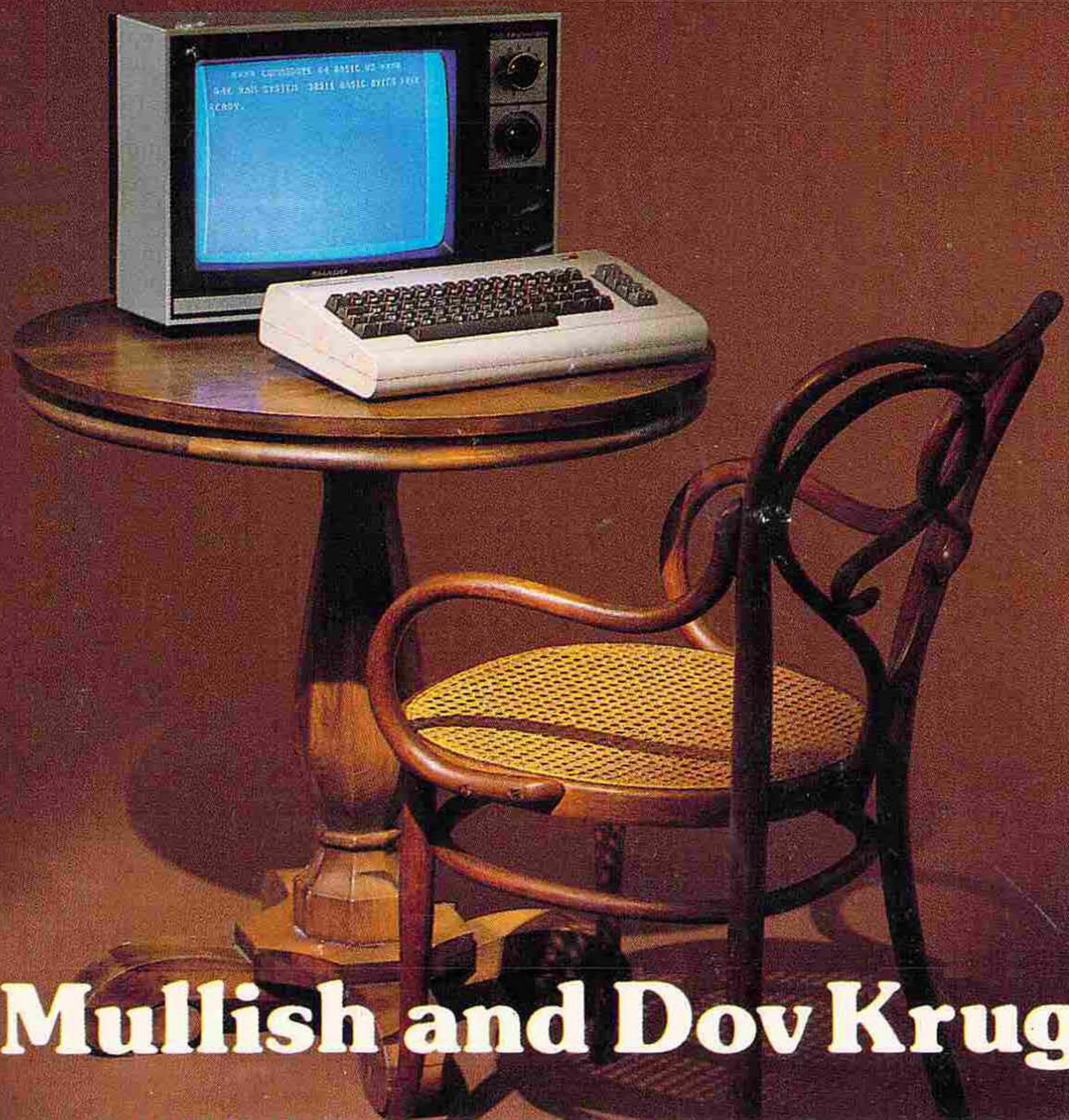


AT HOME WITH BASIC

The Simon & Schuster Guide
to Programming the
COMMODORE 64



Henry Mullish and Dov Kruger





**Other Simon & Schuster books
by Henry Mullish and Dov Kruger:**

*ZAPPERS: Having Fun Playing and Programming
23 Games for the TI-99/4A*





AT HOME WITH BASIC

**The Simon & Schuster
Guide to Programming the
COMMODORE 64**

Henry Mullish and Dov Kruger



**Computer Book Division
SIMON & SCHUSTER NEW YORK**

Copyright © 1984 by Henry Mullish and Dov Kruger

All rights reserved

including the right of reproduction

in whole or in part in any form

Published by the Computer Book Division/Simon & Schuster, Inc.

Simon & Schuster Building

Rockefeller Center

1230 Avenue of the Americas

New York, New York 10020

SIMON AND SCHUSTER and colophon are registered trademarks of

Simon & Schuster, Inc.

Designed by Irving Perkins

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging in Publication Data

Mullish, Henry.

At home with BASIC.

1. Commodore 64 (Computer)—Programming. 2. Basic
(Computer program language) I. Kruger, Dov. II. Title. III. Title:

At home with B.A.S.I.C.

QA76.8.C64M85 1984 001.64'2 84-10542

ISBN 0-671-49861-4

Contents

Introduction	7
CHAPTER 1 Immediate or Direct Mode	13
CHAPTER 2 Programming	27
CHAPTER 3 The Power of BASIC	45
CHAPTER 4 Structured Programming	75
CHAPTER 5 Numeric Functions and Logical Operators	103
CHAPTER 6 Introduction to Character String Manipulation	131
CHAPTER 7 Arrays	163
CHAPTER 8 Advanced String Manipulation	184
CHAPTER 9 Nesting Loops	202
CHAPTER 10 Audio-Visual Program Enhancement	218
CHAPTER 11 Debugging	234
Glossary	247
Appendix A	261
Appendix B	263
Appendix C	267



Introduction

In the past few years, prices of computers have dropped sharply, while at the same time their quality and power have increased. Most of them were selling for well above \$1,000. It seemed to be a general rule that any computer below this price level was classified (and was regarded by many) as a toy. It was not until the beginning of 1983 that Commodore Business Machines (CBM) released its bombshell—the Commodore 64, which sold for the unprecedented low price of \$595. Within six months, the price of the Commodore 64 was further reduced to under \$200, making it by far the most powerful computer available in its price range. This development was made possible by improved mass production techniques. It has become the one true “people’s computer,” providing the novice and experienced programmer alike with a powerful, versatile, and expandable machine.

The Commodore 64 has, as its name implies, 64,000 (64K) “bytes” (characters) of programmable memory and comes equipped with a long list of standard features and available options. These include advanced graphics capability; 16 display colors, a screen that displays 25 rows of 40 columns each; animation with sprites (programmable graphics characters); a three-voice music synthesizer and a “white noise” generator; a 6510 chip (an improved central processor similar to that used in the Apple, Atari, and other leading computers); a built-in serial interface to allow for communication with various external devices, such as printers; a port to allow plug-in cartridges to expand further its capabilities; and finally, two ports to allow input from joysticks, paddles, or a light pen.

However, these are not its only desirable features. There are many other options available, including cartridges to support CP/M (a popular operating system), LOGO (the increasingly widespread educational language), a cassette tape recorder, a disk drive, a printer,

8 ■ Introduction

and a modem (for communication through the telephone with other computers).

The keyboard on the Commodore 64 has 66 full-stroke keys, including four programmable function keys that may be defined within a program for special uses. By taking advantage of its keyboard buffer, it is possible to type up to ten characters ahead while the machine is performing other tasks. Special graphics symbols may be produced directly from the keyboard, or they may be defined by the programmer should their creation become necessary. The full, standard character set that is built into the machine includes both upper and lower case although the BASIC programming language recognizes only upper case.

Getting Started

Once you have the computer connected, it would be a good idea to become familiar with the keyboard since this is the primary method by which you will communicate with the computer. The keyboard is quite similar to that of a standard typewriter—with some important exceptions. The RETURN key for example, which is found on electric typewriters, plays a special role. When it is pressed, the line typed in is sent to the heart of the computer for processing. This is important to remember because if it is not pressed after a command has been issued, the computer simply waits indefinitely, doing absolutely nothing, until it is pressed. After a little practice, remembering to press the RETURN key becomes an automatic reflex.

The SHIFT key works in a manner similar to that on a standard typewriter. However, in addition to enabling you to type in either upper or lower case or to access the rightmost of the two graphics symbols on the front side of many of the keys, it allows other keys to perform a variety of operations. One of these is the CLR/HOME key, which positions the “cursor” at the “home” position of the screen—the cursor being the flashing square on the screen that tells you where the next character will be displayed. The “home” position is the top lefthand corner of the screen. Holding down the

SHIFT key while pressing CLR/HOME not only places the cursor in the home position but also clears the screen.

The CTRL (control) key allows you to perform some specialized operations, one of which is setting the color of the screen. This is accomplished by holding down CTRL and pressing one of the keys labeled "1" through "8". When both keys are released, anything you type appears on the screen in the specified color. For example, if CTRL-2 (the way to indicate pressing both the control key and the 2 key) is pressed, the flashing cursor turns white and all subsequently typed characters appear in white. The CTRL key is always used in conjunction with other keys, never by itself; it is effectively another type of SHIFT, allowing other keys to perform a variety of useful functions.

The Commodore key (the key bearing the famous Commodore logo located at the bottom left of the keyboard) is yet another type of shift key. Among its uses are setting the color in a manner identical to that for the CTRL key. However, whereas using the CTRL key permits access to the first eight colors, the Commodore key allows for a second set of eight colors, as shown in the accompanying table.

Keys	Color Obtained	Keys	Color Obtained
CTRL-1	black	COM-1	orange
CTRL-2	white	COM-2	brown
CTRL-3	red	COM-3	light red
CTRL-4	cyan	COM-4	gray 1
CTRL-5	purple	COM-5	gray 2
CTRL-6	green	COM-6	light green
CTRL-7	blue	COM-7	light blue
CTRL-8	yellow	COM-8	gray 3

You do not have to memorize these color codes since this table may always be referred to when the need arises. There are several other keys that have not yet been mentioned—such as the RUN/STOP and INST/DEL keys—but these will be covered later in the book at the appropriate time.

We assume that you have access to a Commodore 64 computer

with a television (black and white or color) and either a datasette (the Commodore cassette recorder) or a disk drive. The purpose of both the datasette and the disk is to save permanently programs and data which you may type into the computer. This is a very important asset because the computer irretrievably loses the complete contents of its memory every time it is switched off. (Imagine if you forgot everything you know each time you went to sleep and had to relearn everything in the morning.) Once a program has been saved onto either cassette or disk, it may be read into the computer as often as is necessary without any retyping whatever. We will describe later the commands needed to accomplish this task.

As you work your way through this book you will come to realize that at your fingertips you have access to a computer far more powerful than that had by almost any university or corporation a mere generation ago. Today, with the aid of the English-like language called BASIC, you will learn how to control the computer so that it can perform many useful and interesting chores, whether they be of business, education, or home economics.

The BASIC language was developed in the mid-1960s at Dartmouth College, New Hampshire, under the direction of Professors John Kemeny and Thomas Kurtz. It was intended to be simple to learn and use and also inexpensive to implement. In these respects it has achieved its aims magnificently. It was designed to be an interactive language—providing the programmer with an easy means of feeding information into the machine while it is in the process of solving a problem. Thus the user is able to obtain immediate responses to what is typed in at the keyboard.

BASIC was developed for the beginner who perhaps would be expected to convert to other, more specialized languages as his or her skills developed. Since its inception, however, BASIC has undergone considerable evolution, to the extent that it has become the standard language for microcomputers such as the Commodore 64. In its extended form (which is the dialect available on the Commodore 64) it has turned out to be of importance not only for hobby and educational purposes but also for business and many industrial applications.

In the following chapters, all the commands found in Commodore

64 BASIC are discussed in detail and are explained with the aid of a large variety of programs designed to teach you as effectively as possible to write error-free, well-designed programs. In today's world, handling information has become one of the most critical of skills. Perhaps it is true to say that the power of a modern nation can be measured better in terms of the sum total of its computer expertise than by the size of its armies. Whether you are a professional, a factory worker, or a student, you will learn not only the complete repertoire of the BASIC language but also, by example and description, how to program this and any other machine in clear, organized BASIC, written in a consistent, structured style. If this is your first exposure to computers we take this opportunity to welcome you to the wonderful world of programming.

You are about to set out on a journey that may prove to change some aspect of your life. The road ahead is both interesting and challenging and, you are cautioned, may even prove to be addictive. But what a delightful and acceptable addiction it is. In the words of the sixth-century B.C. Chinese philosopher Lao-Tzu, the reputed founder of Taoism, "The journey of a thousand miles begins with the first step."

You have just made that first step. Good luck.



Immediate or Direct Mode

In this chapter you will be introduced to some fundamental concepts of computer programming. Among them are

- direct versus indirect mode
- the meaning of a “literal”
- the use of the double quotation symbol
- the PRINT instruction and its abbreviated form
- the arithmetic operators
- the result of dividing by zero
- the SQR function
- the print zones
- the effect of the semicolon in a PRINT statement
- the Commodore 64 as a calculator
- what is meant by “scientific notation”
- the so-called hierarchy of the mathematical operators
- the use of parentheses

Although the Commodore 64 is a computer, it can be instructed to perform arithmetic operations as if it were a calculator. It can print messages or evaluate mathematical expressions with uncanny speed and accuracy. Calculations may be performed in what is called *immediate* or *direct mode*, as opposed to storing the instructions for later use, a mode often called *deferred* or *indirect* or sometimes *pro-*

gram mode. For now, you will explore some options in direct mode and thereby become acquainted with some useful features that you will be using later on when you learn the fundamentals of programming.

Defining and Printing a Literal

A *literal* is a sequence of characters enclosed within double quotation marks. However, for a literal to be meaningful to the computer, it must be associated with a command. The most common type of command is the PRINT instruction, by which messages may be displayed on the screen. Here are some typical examples of PRINT statements incorporating literals:

```
PRINT "THE COMMODORE 64 IS A PERSONAL  
COMPUTER"  
PRINT "WITH MORE BYTES FOR THE BUCK."  
PRINT "WE SHALL SHORTLY LEARN HOW TO  
PROGRAM IT."
```

Since anything at all may be included in a literal, it does not follow that every statement displayed on the screen is necessarily true. For example,

```
PRINT "2 + 2 = 94.234"
```

cheerfully prints the message that 2 and 2 is equal to 94.234.

When program instructions are performed by the computer (executed) the characters between the quotation marks are printed verbatim. When a PRINT statement is issued, the message is immediately displayed on the screen. Should the message be longer than the width of the screen, it is continued to the beginning of the next line without causing any errors. In this way, PRINT statements up to 80 characters (two lines) long may be used.

Since the double quotation sign is the symbol used to “delimit” a literal—that is, it defines the literal’s beginning and end—the sign

itself cannot be used as one of the characters within the literal. For the time being, the apostrophe may be used as a substitute symbol for the double quotation mark. Later, you will learn other methods of circumventing this restriction.

The computer is extremely literal in its interpretation of commands. Should, by mistake, the instruction

```
PRINT "HELLO TO ALL YOU COMMODORE 64
PROGRAMMERS"
```

be typed in, the computer will respond with ?SYNTAX ERROR because it does not recognize the command PRINT.

Since the PRINT instruction is used so much in BASIC, the Commodore 64 has a special feature to expedite its use. A question mark (the single character) may be substituted for the entire word PRINT. That is, the instruction

```
? "WHAT A WEIRD COMMAND THIS APPEARS TO
BE!"
```

prints the literal enclosed in the quotation marks, just as if the question mark were replaced by the command PRINT.

The Arithmetic Operators

In addition to printing out messages, the Commodore 64 has the capability to act as a supercalculator. For example, the instruction

```
PRINT 2 + 3
```

followed by RETURN (as usual) prints the result of 5, in much the same way as would a calculator. By the same token, the instruction

```
PRINT 4 - 5
```

where the minus sign is the symbol for subtraction, produces the value -1. When used in this context it is called a *binary* operator

16 ■ AT HOME WITH BASIC

because it operates on two values. The minus sign is also used for what is known as *negation*. Any number preceded by a minus sign is a negative number. If the number is already negative, it becomes positive. When the negative sign is used in this fashion, it is called a *unary* operator because it operates on one value. An example of the minus sign in its unary form is

```
PRINT -5
```

Similarly, the statement

```
PRINT --5
```

displays the negative of negative 5 (which is 5).

Multiplication uses the asterisk symbol

```
? 6 * 7
```

This instruction prints 42 immediately after the RETURN key is pressed. In the same way division is effected by using the slash symbol, /, as shown in the next example:

```
? 15 / 5
```

This yields the result of 3. There is a special case of division that is illegal—division by zero. Any attempt to do so generates the message

```
?DIVISION BY ZERO ERROR
```

and execution of the program is terminated. The error message is printed because, according to the rules of mathematics, no number can be divided by zero. Try this yourself. In fact, try it twice—first with the question mark and then by typing PRINT.

The 64 also has the capability to handle fractional values (often called real numbers). For example,

```
PRINT 10.4 / 2
```

produces the result of 5.2.

There is one more arithmetic operator. It is the exponentiation symbol, which raises a number to a power. This function is represented on the Commodore by the symbol \uparrow . An example of its use is

```
PRINT 2 ↑ 4
```

which displays the number 16 (or $2 \times 2 \times 2 \times 2$).

The Square Root Function

Although not, strictly speaking, a true operator, the square root function is so frequently used in conjunction with the mathematical operators that it must be mentioned with them. In BASIC, the square root of a number is usually found by typing SQR followed by the number, which must be enclosed within parentheses. For example, in order to display the square root of 5, the following command is used:

```
PRINT SQR(5)
```

Those who are mathematically inclined may know that the square root is the inverse of the square of a number. In other words, the statements

```
PRINT SQR(4 ↑ 2)
PRINT SQR(4) ↑ 2
```

both return 4 because the two operations, \uparrow and SQR, cancel each other out, in much the same way as in the statement

```
PRINT 4 / 2 * 2
```


18 ■ AT HOME WITH BASIC

Here the division by 2 is canceled by the subsequent multiplication by 2, demonstrating that multiplication and division are inverse operations.

Print Zones

The instruction

```
PRINT 5,-17
```

where a comma separates the two constants, 5 and -17 , prints the constants in a special way. The screen is automatically divided into four zones, each of which occupies ten spaces. The number 5 is printed in the first of these zones, and the number -17 is printed in the second. Both appear at the left of the zones as shown (the 5 is printed in the second column because the computer reserves the first column for a possible negative sign):

		5																		
column numbers	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Using the Semicolon

Whenever a positive number is printed, it is always preceded by a blank space. Thus the constant 5 is printed in column 2. Negative numbers, on the other hand, are preceded by a minus sign—instead of a space. All numbers when printed, whether positive or negative, are followed by a space. In the previous example, the space falls within the zones of ten columns each. In the following example, the semicolon is used instead of the comma. Since the zones are ignored when the semicolon is used, the format of printed numbers becomes visible. Therefore, the statement

```
PRINT 4;15;-22
```

appears as

	4	1	5	-	2	2				
column numbers	1	2	3	4	5	6	7	8	9	10

where the number 4 is printed in position 2 (being preceded by a space because it is positive) and followed by a space (as are all numbers). The next number, 15, is also preceded and followed by a space for the same reasons. The last number on the line, -22, is not preceded by a space because it has a minus sign. It is also followed by a space although this is not visible because it is the last number of the list to be printed.

On the other hand, a comma between literals in a PRINT statement forces the printed items into their corresponding zones with no added spaces, and a semicolon joins them, as illustrated in the following examples: *

```
PRINT "ABRA", "CADABRA"
ABRA          CADABRA
```

```
READY.
```

```
PRINT "ABRA"; "CADABRA"
ABRACADABRA
```

```
READY.
```

The word "READY." is automatically displayed by the system at the end of the output (usually after one blank line) to inform you, the programmer, that the computer has completed its task and is ready to accept further instructions.

In program mode, as you will soon learn, one sometimes places a

* Throughout this book, OUTPUT (characters generated by the computer) appears in dot-matrix type. INPUT (characters that the reader must type in) appears in normal, fully formed type.

semicolon or comma at the end of a PRINT statement in order to force the next PRINT instruction to display its output adjacent to the previous printout. The semicolon simply “splices” the two fields, placing them close together, whereas a trailing comma has the effect of forcing subsequent output to appear in the next zone over.

Scientific Notation

On occasion it is necessary to work with numbers so small or so large that the Commodore 64 cannot represent them in the standard decimal notation—even within its maximum capacity, which is nine digits. In such cases, the computer resorts to so-called “scientific notation.” On the Commodore 64, scientific notation uses the letter E (for exponent) in the following manner. The number (written in scientific notation)

$$2.98 \times 10^{11}$$

appears on the screen as 2.98E+11, where the letter E stands for “times 10 to the power.” This simply means that the number is such that the decimal point is located 11 places to the right of its shown position. In the case of a negative value following the E, such as 5.7216E-10, the decimal place is actually located 10 places to the left of the position displayed. The number 5.7216E-10 would therefore be equivalent to .00000000057216, which is too long a number to be displayed by the Commodore.

The Hierarchy of the Mathematical Operators

Examine the following instruction and try to guess what answer the computer would yield:

```
PRINT 2 + 3 * 4
```

Perhaps you would like to confirm your answer with a pocket calculator first. It might be of interest to you to know that it is possible for two working calculators (depending on the particular type used) to give totally different answers to this question. In Commodore 64 BASIC (and indeed, any other dialect of the language) the answer given will always be 14 because it evaluates arithmetic expressions according to the algebraic rules of precedence. This means that in any given arithmetic expression, exponentiation takes place first, followed by negation (the unary minus), followed by multiplication and division (in left-to-right order), and finally addition and subtraction (again in left-to-right order). This may be summarized in the following way:

Precedence level (highest to lowest)	
↑	exponentiation
-	negation (unary minus)
* and /	multiplication and division (same level)
+ and -	addition and subtraction (same level)

The following expression and subsequent stages of solution, illustrate the order of operations the Commodore obeys:

$$9 + 2 \uparrow 3 / 2 * 4 - 1$$

First, the exponentiation takes place. The term $2 \uparrow 3$ is reduced to the number 8. The expression is now treated as shown:

$$9 + 8 / 2 * 4 - 1$$

Next, all multiplications and divisions are executed. Since both are on the same "level" in the hierarchy, they are executed in a left-to-right order. That is to say, 8 is first divided by 2, giving 4, and then multiplied by 4, giving 16. The resulting expression then becomes the following:

$$9 + 16 - 1$$

22 ■ AT HOME WITH BASIC

At this stage, addition and subtraction, also being on the same level as each other, are executed in a left-to-right sequence. The expression finally becomes the single value 24.

Sometimes, it is desirable to group together terms of an expression. This may be done by placing parentheses around the desired terms. Whatever is enclosed within parentheses is acted on first, however, in the same order of precedence just discussed. So that in the example

$$(4 + 12) / 2$$

the addition is performed first since it is enclosed within parentheses. Therefore, the answer is $16 / 2$, which is 8, rather than $4 + 6$, which would yield the false result of 10.

If an expression contains parentheses within parentheses, the terms in the innermost parentheses are evaluated first. This is illustrated in the next example:

$$((3 + 7) * 2) - 5$$

in which the innermost parentheses containing $3 + 7$ is evaluated first, yielding 10. This is then multiplied by 2, giving 20, and finally 5 is subtracted, giving the result of 15. As long as the parentheses are balanced, no harm is done by enclosing the whole expression in yet another set of parentheses. The expression

$$4 + 5$$

and

$$(((4 + 5)))$$

are equivalent to all intents and purposes. The redundant parentheses in the second version never create errors even though they are more time-consuming to type. However, should the inclusion of redundant parentheses enhance the clarity of an expression, you should feel free to use them.

Review Questions

- 1 ■ What is a literal?

A literal is a group of characters enclosed by double quotation marks. The quotation signs are not part of the literal; they merely indicate where the literal begins and ends (delimiters). A literal is often included in a PRINT statement. When the literal is printed the quotation signs do not appear, only whatever is enclosed between the double quotation signs.

- 2 ■ What role does the PRINT instruction play?

It causes the computer to display the specified information on the screen.

- 3 ■ What symbol is used for

- (a) addition?
- (b) subtraction?
- (c) division?
- (d) multiplication?
- (e) exponentiation?

(a) + (b) - (c) / (d) * (e) ↑

- 4 ■ What is special about division by zero?

Division by zero is illegal and causes the message

?DIVISION BY ZERO ERROR

to be displayed.

- 5 ■ How would you represent the number 42400000 in scientific and exponential (E) notation?

Scientific notation: 4.24×10^7

Exponential form: 4.24E+7

24 ■ AT HOME WITH BASIC

- 6 ■ Which of the five arithmetic operators has the highest precedence?

Exponentiation (represented by the symbol \uparrow).

- 7 ■ Name two operations which have the same precedence level?

Addition and subtraction; multiplication and division.

- 8 ■ Which has the higher precedence—addition or subtraction?

Neither. Since they are on the same level, whichever comes first in a left-to-right scan of the expression is performed first. The same rule applies to multiplication and division.

- 9 ■ Of what use are parentheses?

Parentheses are used to group together terms of an expression that are to be evaluated first. For example, to find the average of 5 and 6 in one instruction, parentheses would have to be used as shown:

```
PRINT ( 5 + 6 ) / 2
```

yielding the answer 5.5. If the parentheses were omitted, the answer would be $5 + 3$, which is 8 and is not the average.

- 10 ■ Is it legal, that is, permissible, to enclose an expression with redundant parentheses?

Yes. This is particularly desirable if the added parentheses add clarity to the expression.

- 11 ■ What is the effect of separating items of a PRINT statement by a comma?

The items are printed in separate zones.

- 12 ■ What is the effect of separating the items of a PRINT statement with a semicolon?

The items are printed close together, the zones being totally ignored.

- 13 ■ What is the effect of a semicolon at the end of a PRINT statement?

A semicolon inhibits the automatic advance of the line feed, thereby causing the items in a subsequent PRINT list to be printed adjacent to the current one. This effect is possible, however, only in program mode.

HANDS-ON PRACTICE

1. Print out your full name in immediate mode.
2. Print the name of your favorite television personality.
3. Have the computer calculate and display the result of multiplying 17 by 34.
4. Find out exactly what happens if the word "PRINT" is misspelled.
5. Instead of typing the letters of the word "PRINT", use its alternate form, the question mark, to print out the current day of the week. Use this method from now on to save time.
6. Print out your age divided by zero. Observe the message and the number that is generated. If you are lucky, this is the last time you will see it.
7. Using a PRINT statement, display the numbers 1, 2, 3, and 4 in separate zones of the screen.
8. Use semicolons instead of the commas in question 7 and compare the spacing.
9. Calculate the square root of 123 by using the square-root function and by raising the number to the power $\frac{1}{2}$. Do both answers agree?

TRY YOUR HAND AT THESE

What is displayed by the following statements:

1. PRINT
2. PRINT 5 + 4 * 3

26 ■ AT HOME WITH BASIC

3. PRINT 3 + (2 * 3)
4. PRINT 2 ↑ (1 + 4 / 2)
5. PRINT 4 ↑ 1 / 2
6. PRINT (((5 * (4 * (3 + 7) / 2) - 3)))
7. ?3 + 4 * 5
8. ?(5 + 2) / 0
9. ?0 / (2 + 10)

Programming

What is Programming?

In this chapter you will learn the fundamentals of programming in BASIC, the primary focus of this book. Among the many topics to be covered are

- the definition of a program
- line numbers and their significance
- direct and indirect modes
- executing programs by using the RUN command
- the role played by the END statement
- sequential processing of the program instructions
- examining programs by using the LIST command
- inserting statements into a program
- editing and modifying a program
- cursor control keys
- INST/DEL keys
- CLR/HOME key
- erasing the program

Now that you have had some experience in using the computer in immediate mode, the time has come to learn how to exploit it in indirect, or deferred, mode. This is by far the most usual mode in which the computer is used. The advantages of program mode are

considerable. You can construct a long list of instructions which may be deferred until some later time before they are acted on, thereby enabling you to construct a very complex sequence of instructions which can be run as often as you like without having to type them in again. In immediate mode they would have to be retyped each time they are required.

A program is nothing more than a specific set of computer instructions (such as those we have already seen) designed to solve a particular problem. The programmer assumes the responsibility of specifying each of the program instructions. The instructions are stored in memory until a command is issued to execute them. It is for this reason that a program is executed in what is known as *deferred* or *indirect* mode. To execute a program the RUN command is used. It executes whatever program resides in memory at the time. Before typing in a program, it is recommended that you type the command NEW, which clears the memory in preparation for a new program.

In deferred mode, each program instruction must be preceded by a line number, which may be any whole number from zero to 63,999. (Incidentally, commas are never included in a number when it is part of a computer program; the previous number would be written as 63999, without the comma.) The presence of a line number is what differentiates an indirect command from a direct one. As seen in the last chapter, as soon as a command without a line number is typed (followed by RETURN) it is executed and the results are displayed on the screen. The instruction is not stored anywhere and therefore cannot be recalled later except by retyping it. This is what is known as a direct command. After the direct command

```
PRINT 5 + 2
```

is executed, the result of 7 is displayed on the screen, followed by the reassuring "READY." to advise the user that the computer is ready to accept further instructions. When the same command is given in indirect form,

```
10 PRINT 5 + 2
```

the computer does not display anything; it merely stores the instruction and waits for further commands. If the programmer were to type in a second indirect instruction,

```
20 PRINT "THE COMMODORE IS STORING THIS  
LINE"
```

it too is tacitly accepted and stored into memory. In order to execute these two commands, the RUN command is issued. This command does not take a line number, as the programmer wishes to execute the program, not add another line to it. As soon as the RETURN key is pressed, the computer responds by executing all the commands placed in program memory—in order of line number.

In some versions of BASIC found on other computers it is necessary for the last line of every BASIC program to be an END statement. Although this is not required on the Commodore, it can, nevertheless, be included without any detriment to the program. If it is included, it does not have to be the last instruction in the program, as is the case in most other versions of BASIC. In fact, several separate END statements may appear in a single Commodore 64 program. Whenever the END statement is executed, the program is terminated immediately.

Line numbers may be entered in any order at all; the computer will automatically sort the individual instructions into ascending order of line number, so that when the program is run, each instruction is executed in the order of ascending line number. Each line is executed completely before the next one is even looked at—like reading a book; as soon as one line is completely read, the reader automatically goes to the next line, and so on.

To verify that this operation does indeed occur, you may type in another line with a line number lower than either of the other two:

```
5 PRINT "THIS IS NOW THE FIRST LINE"
```

When the amended program is again executed (by means of the RUN command) the output appears as

30 ■ AT HOME WITH BASIC

```
THIS IS NOW THE FIRST LINE
7
THE COMMODORE IS STORING THIS LINE

READY.
```

showing clearly that the three-lined program was executed in order of ascending line number. This feature is particularly useful when you want to modify programs, either because they contain errors or because you want to improve them.

To view the complete program you use the LIST command. Once again, this command does not take a line number, for you wish to view the program, not place the command within the program. The LIST command may be obtained by typing either the word "LIST" (followed by the RETURN key) or its permissible abbreviation, which is the letter "L" followed by a shifted "I" (and then RETURN). Whichever method is used, the following listing should now appear on the screen:

```
LIST

5 PRINT "THIS IS NOW THE FIRST LINE"
10 PRINT 5 + 2
20 PRINT "THE COMMODORE IS STORING THIS
    LINE"

READY.
```

This program prints each line of output on consecutive lines. If you wanted to separate them with blank lines, "null" PRINT statements may be included. For example, you can type

```
7 PRINT
15 PRINT
```

which are automatically inserted in their correct positions in the program and have the effect of printing blank lines between each of the three printed lines.

When executing a program, it is not always desirable to start from the beginning. Perhaps the programmer is interested in checking

out just a small section of code, which may reside somewhere in the middle of the program. The command

```
RUN 1570
```

executes the program beginning with line 1570 and extending to the end of the program. Similarly, not all the program need be listed. A single line of a program may be listed by typing the line number after the command LIST. So that

```
LIST 2340
```

lists line 2340 only. (Notice the difference between the action of RUN and LIST when followed by a single line number.) In order to list a program from a given line to the end of the program, the line number in question and a dash must follow the LIST command, as shown in the following example, where lines 190 to the end of the program are listed:

```
LIST 190 -
```

To list the segment of code between two given lines, the instruction LIST is followed by the first line number, a dash, and the second line number. In other words, the instruction

```
LIST 492 - 875
```

lists the program from lines 492 to 875 inclusive. The screen can contain a maximum of 25 lines, each consisting of 40 columns. If a large program is to be listed, the screen is not able to contain it all at one time. When the program is listed, it will fill the screen and then the top line will continually disappear from the top of the screen to make room for each new line at the bottom. This process is known as *scrolling*. The difficulty is that the computer scrolls so fast that people have trouble keeping up with it. For the viewer to examine each line of the program, a special "slow-motion" feature is included.

When anything is being printed on the screen, simply hold down the CTRL key. As long as it is held down, output to the screen is

slowed down. If you need to take a longer look at a section of a program than the slow-motion feature permits, you have a choice of options. First, you can list the specific lines that you are interested in. They will then appear stationary on the screen. The second alternative is to list the program until you reach the part you wish to examine and then press the RUN/STOP key quickly—before the screen scrolls too far. Unfortunately, this command halts the listing and it is impossible to resume displaying the steps of the program without issuing another LIST command.

Numbering the Lines of a Program

One of the glaring realities of computer programming is that programs are invariably changed after they are written, for one of many different reasons. An error might be found in the program or the demands made of the program might have altered or, in the case of an income-tax program, for example, the government may have issued new guidelines which make obsolete the old version of the program. Whatever the reason, it is strongly suggested that assigned line numbers *not* be in consecutive order. In other words, it is not a good idea to assign the line numbers

```
100      101      102      103
```

to four successive lines of a program because it is impossible to insert lines between these numbers should the occasion demand. It is generally advisable to begin at line 100 and skip line numbers in steps of ten. The following program illustrates the typical manner in which line numbers should be selected:

PROGRAM 2-1

```
100 PRINT "      THE COMMODORE"
110 PRINT "HAS A MEMORY CAPACITY OF"
120 PRINT "      64K"
```

RUN

THE COMMODORE
HAS A MEMORY CAPACITY OF
64K

READY.

Editing and Modifying a Program

You will find that when you write computer programs, they will probably not work the first time—perhaps the first few times. Experience has shown that very few programs work at the first try the way they were designed to, no matter how skillful or intelligent the programmer may be. The computer is totally objective when executing programs. It does not care who you are, what your position is, what rank you hold, or what your salary bracket is. A programming error is still a programming error—even if it is only a matter of a missing comma. Indeed, you may rest assured that few of the programs illustrated in this book worked the first time.

In light of this fact, it is essential for any computer to allow for fast, simple modification of programs, since changes will always have to be made. Commodore BASIC provides a built-in, advanced screen editor which makes this task rather easy.

Replacing a Line

In order to correct a line, you may simply retype it with the same line number. When the RETURN key is pressed at the end of the line, the new line replaces the previous version. This operation may be seen in the following illustration:

LIST

100 PRINT "THERE IS A MISTEAK IN THIS LINE"
110 PRINT "BUT THIS LINE IS FINE"

READY.

The following line is now typed:

```
100 PRINT "NOT ANYMORE THERE ISN'T"
```

yielding when relisted

```
LIST
100 PRINT "NOT ANYMORE THERE ISN'T"
110 PRINT "BUT THIS LINE IS FINE"
```

READY.

To erase a line from a program completely, simply type in the line number and press the RETURN key. This move replaces the old line with a new, blank line. For example,

```
LIST
100 PRINT "THIS IS A LINE TO BE DELETED"
```

READY.

The following two lines are typed:

```
100
LIST
```

READY.

Using the Cursor Control Keys

The most frequent error when typing a program into the computer is mistyping a character. If it were done on a regular typewriter, you could either start again on a fresh page or use white-out. However, neither step is necessary on the Commodore. Assuming that the mistake was noticed immediately after it was typed, a simple touch of the INST/DEL (insert/delete) key will delete it. The character immediately to the left of the flashing cursor then disappears, and you can type the correct character in its place.

Often, however, the mistake is not found until you have typed further in the line, as in the following example:

```
170 PRINT 5 + 2 * 3_
```

Deleting each preceding character until the cursor is in the position

```
170 PRI_
```

is both unnecessary and wastes time because correctly type characters will be unnecessarily erased. The best way to make the change is to use the cursor control keys. On the bottom right of the main keyboard, there are two special cursor control keys that, when used in conjunction with the SHIFT key, enable you to move the cursor left, right, up, or down. These cursor control keys do not destroy characters over which the cursor passes. All you need do is move the cursor left twelve spaces until it is in the position shown:

```
170 PRINT_ 5 + 2 * 3
```

At this point, merely type in the correct character (N). The line then appears as

```
170 PRINT_ 5 + 2 * 3
```

If at this point the line is finished, simply press RETURN. If not, move the cursor to the end of the line by hitting the right cursor control key (→) eleven times. From this point on, the line may be typed to its completion and the RETURN key pressed.

There is a common situation in which the cursor keys do not work the way you might expect. If you are in the middle of typing a literal—that is to say, the opening quote sign has been typed—the cursor control keys produce unexpected graphics symbols, rather than simply moving the cursor as they ordinarily do. The reason for this is that once the opening quotation mark has been typed, the computer is in “quote” mode. We take advantage of this mode in Chapter 10, where we discuss the embedding of special characters within a literal. For the present, we recommend that if a character within or before a literal is to be corrected, complete the literal with

the closing quotation mark (which negates quote mode). All corrections can then be made in the normal way.

Suppose now that you have just typed a line and pressed the RETURN key.

```
130 PRINT "THE COMMODORE 64 IS THE
      GREATEST"
```

The cursor is now located at the next line, but the letters "CO" of the name "COMMODORE" have been transposed. In order to correct the error, use the upward cursor control key to place the cursor on the appropriate line. Once there, the procedure to use is identical to that described above (since the closing quotation mark has been typed, quote mode will not be entered). After the corrections have been made, be sure to press the RETURN key again, or the computer will not register the change.

Using the INST/DEL Key to Insert a Character

Suppose you mistyped the word "COMMITTEE" in a literal,

```
100 PRINT "THE COMMITTEE MEMBERS WERE
      ELECTED" _
```

and the cursor is located at the end of the line prior to the line being entered. What you wish to do is to insert another "T" in the misspelled word. Rather than moving the cursor to the location of the error and retyping the whole line from that point on, you should use the INST (insert) function (obtained by holding down SHIFT and pressing the INST/DEL key simultaneously), which makes it possible to insert a character in the required position. The first step, of course, is to move the cursor to the position where the letter must be inserted—that is, to the immediate right of the letter "I."

```
100 PRINT "THE COMMITTEE MEMBERS WERE
      ELECTED"
```


At this point, press the INST/DEL key while holding down SHIFT, thereby generating a space to the immediate right of the cursor and pushing the rest of the line of text one space to the right. If more than one character must be inserted, the process may be repeated as often as is necessary by simply holding down the keys; they are repeated automatically. The desired character (or characters) may now be inserted, and the line entered by pressing RETURN.

In the same way information can be inserted in the middle of a line, so characters may be deleted, by using the INST/DEL key. As the following example shows, the INST/DEL key erases the character immediately to the left of the cursor.

```
460 PRINT "WATCH THIS_ CAREFULLY!"
```

If this operation is performed in the middle of a line, the following occurs:

```
460 PRINT "WATCH THIS CAREFULLY!"
```

where not only has the character to the left of the cursor been deleted but the line has “closed up” to account for it. In other words, the INST/DEL key deletes not only the character but the room that was provided for it as well.

The CLR/HOME Key

Wherever the cursor might be on the screen, hitting the HOME key has the effect of relocating it to its “home” position, which is the top lefthand corner of the screen. This is useful when there is a need to edit a line located near the top of the screen, since it brings the cursor to the vicinity of the line instantly. Holding down SHIFT at the same time as pressing the HOME key has the additional effect of clearing the screen. This operation is useful when the display is cluttered, making it difficult to concentrate on the line at hand. Clearing the screen before editing a line is always a good idea (the

program still resides in memory and is not affected). All you need do then is to list the program in question. In this way, the line to be modified appears alone on the screen, making the job of correcting it that much easier.

Erasing the Program in Memory

When you are finished with an old program and wish to write another, you should first clear program memory. If it is not done, the statements from the new program will be merged into the old one because the computer has no way of knowing that you want the two programs kept separate. For example, if the two lines

```
10 PRINT "THIS IS THE"
20 PRINT "FIRST PROGRAM"
```

are in memory, and two lines from a new program are typed in,

```
10 PRINT "THIS IS"
15 PRINT "THE SECOND PROGRAM"
```

the lines are merely added to the already existing program, resulting in the following amended single program:

```
10 PRINT "THIS IS"
15 PRINT "THE SECOND PROGRAM"
20 PRINT "FIRST PROGRAM"
```

This consists of bits and pieces of both programs, effectively nonsense. The way to avoid this problem is to erase the first program entirely by means of the NEW command. The proper sequence of commands is, therefore,

```
LIST

10 PRINT "THIS IS THE"
20 PRINT "FIRST PROGRAM"
READY.
```

NEW

LIST (This step is not necessary. It is only used to show that there is no program in memory. The second program can now be entered.)

READY.

```
10 PRINT "THIS IS"
15 PRINT "THE SECOND PROGRAM"
LIST
```

```
10 PRINT "THIS IS"
15 PRINT "THE SECOND PROGRAM"
READY.
```

Review Questions

- 1 ■ What is a program?

A program is a specific set of computer instructions designed to solve a particular problem.

- 2 ■ Is the line number 123.45 valid in Commodore BASIC?

No. Only integers (whole numbers) between 0 and 63999 are permitted as line numbers.

- 3 ■ What differentiates a direct instruction from an indirect one?

An indirect instruction has a line number whereas a direct instruction does not.

- 4 ■ What is the function of the RETURN key?

It transmits the line just typed into the computer.

- 5 ■ What command causes execution of a program to commence?

The RUN command.

40 ■ AT HOME WITH BASIC

- 6 ■ What is the effect of running the following program?

```
100 PRINT "3 + 4 * 5 = 2"  
110 PRINT 3 + 4 * 5  
120 PRINT 100 - 25 / 5  
130 PRINT 25 - 100 / 5
```

```
3 + 4 * 5 = 2  
23  
95  
5  
READY,
```

- 7 ■ Does the last statement have to be the END statement in Commodore BASIC?

No, but the END statement may be used anywhere, if desired.

- 8 ■ Must program lines be entered in ascending numerical order?

No. The computer automatically inserts each new line at its correct position (according to line number) each time one is entered.

- 9 ■ What is the meaning of the instruction

```
RUN 2000
```

It has the effect of executing the program in memory, beginning from line 2000.

- 10 ■ What is the effect of the statement

```
LIST 2000
```

The instruction at line 2000 only is displayed (listed).

- 11 ■ What is the meaning of the instruction

LIST 400 -

All lines beginning with 400 and extending to the end of the program are listed.

- 12 ■ How may lines 500 to 620 of a program be listed?

By typing

LIST 500 - 620

- 13 ■ How many lines of text may be displayed on the screen?

Up to 25.

- 14 ■ How many columns may be displayed on the screen?

Up to 40.

- 15 ■ What is meant by scrolling?

The top line of the screen is replaced by the line immediately below it, which in turn is replaced by the line below it. The contents of the screen thus shift upward to make room for a new line at the bottom.

- 16 ■ How can a scrolling screen be slowed down?

By pressing the CTRL key.

- 17 ■ How is it possible to resume scrolling a screen at normal speed once it has been slowed down?

By releasing the CTRL key.

- 18 ■ Why isn't it a good idea to give consecutive line numbers to a segment of code?

Because it prevents the possibility of inserting lines later on should the need arise.

42 ■ **AT HOME WITH BASIC**

19 ■ Which of the following commands is invalid?

- a. 100 PRINT
- b. 110 Print 'HI'
- c. 120 PRINT 'WE GOT YER NUMBA'
- d. 130 pRiNt
- e. 140 PRint

- a. Valid.
- b. Invalid (lower case not permitted in a BASIC instruction).
- c. Valid.
- d. Invalid (same as above).
- e. Invalid (same as above).

20 ■ What is the effect of pressing the CLR/HOME key?

It moves the cursor to the “home” position at the top left of the screen.

21 ■ What is the effect of pressing CLR/HOME while holding down SHIFT?

The screen is cleared and the cursor moved to the “home” position.

22 ■ What role is played by the INST function?

It allows for insertion of a character (or characters) in a BASIC line.

23 ■ What is the effect of the DEL function?

The character at the cursor location is deleted and all characters to the right of the cursor move left to “close up” the line.

24 ■ How are the functions INST and DEL obtained?

DEL is obtained by simply pressing the INST/DEL key. INST is obtained by pressing the same key while holding down SHIFT.

- 25 ■ If a program contains a line whose number is 125, and the programmer types the number 125 followed by RETURN, what happens?

Line 125 is erased. All other lines remain intact.

- 26 ■ If a program contains a line 400, and another line is subsequently given the number 400, how does this affect the program?

The first line 400 is erased and is replaced by the second one.

- 27 ■ What is the effect of the NEW command?

The NEW command clears any program in memory to allow for a new program to be typed.

HANDS-ON PRACTICE

1. What difference is there in the output produced by the following two programs? Type them in and see for yourself.

VERSION 1

```
100 PRINT "HOW ARE YOU?"
105 PRINT "FINE THANK YOU"
110 END
```

VERSION 2

```
105 PRINT "FINE THANK YOU"
100 PRINT "HOW ARE YOU?"
```

2. LIST version 2, using two methods.
3. Type in the following line and do not hit RETURN:

```
10 PRINT "THIS IS A BERRY PORLY SPELT
LITTERAL"
```


44 ■ AT HOME WITH BASIC

Modify the line so that the very poorly spelled literal is correctly typed.

4. Having pressed RETURN in question 3, assume that line 10 must now be erased. Type in the number 10 and hit the RETURN key. Now list the program. What has happened?

TRY YOUR HAND AT THESE

1. Write a program to print your name, address, and social security number.
2. Write a program to display the following output:

```
FORTRAN IS SIMPLE ;  
COBOL IS FUN ,  
PASCAL IS USEFUL  
BUT BASIC'S THE ONE ,
```

CHAPTER

3

The Power of BASIC

In this chapter you will get down to the fundamentals of computer programming. Now that you have learned some of the commands available in BASIC you will incorporate them into viable computer programs that can perform useful functions. In particular, you will become familiar with the following:

- the concept of the loop
- the GOTO instruction
- scrolling information across the screen
- the CTRL slow-scroll feature
- the infinite loop
- halting a program (RUN/STOP)
- semicolons within PRINT statements
- defining variables
- the assignment statement
- rules for constructing variable names
- common pitfalls in constructing variable names
- counting
- initializing variables
- an introduction to decision making in Commodore BASIC (IF . . . THEN)
- the relational operators
- interacting with the computer by using the INPUT statement
- the READ, DATA, and RESTORE statements
- the trailer technique
- storing programs on cassette and disk

The Concept of the Loop

Consider, for the moment, the following program, which consists of two lines, each of which prints a literal.

PROGRAM 3-1

```
100 PRINT "*****"
110 PRINT "=====
```

RUN

```
*****
=====
```

READY.

This program, as you may well have guessed, prints one line of asterisks followed by another line of equal signs. If, for some reason, you now wanted to amend the program so that it would fill the screen with alternating lines of asterisks and equal signs, you could merely add additional lines identical to lines 100 and 110. They might be numbered 120, 130, 140, 150, and so on. However, this program would be very inefficient and particularly tedious to type. Since any new lines will only be repetitions of the first two, BASIC provides several ways to repeat the two desired instructions as many times as necessary. The simplest of these is the GOTO instruction, which unconditionally “goes to” the line number specified in the GOTO statement. In Program 3-2, the GOTO statement tells the computer to go to the line specified. The effect is that the instructions are repeated until the GOTO is executed again, endlessly. In other words, a “loop” is created. The loop is the single most vital concept that separates the computer from other machines for the simple reason that the computer can be instructed to repeat boring tasks endlessly, without human intervention. Without this looping capability it would hardly be worthwhile resorting to a computer for many of the problems for which it is used. Program 3-2, which follows, is an example of one containing an endless or infinite loop.

PROGRAM 3-2

```

100 PRINT "*****"
110 PRINT "====="
120 GOTO 100

```

RUN

```

*****
=====
*****
=====

```

[and so on]

In a short while the screen is filled, but the program keeps printing the two lines of output without end. As a result, the top line keeps moving off the top of the screen, and every line moves up, to be filled immediately by the succeeding line. This occurs so fast that the text seems to remain stationary, although you may notice that it does seem to blink.

By now, you may be wondering how to stop the demon program that you have set in motion! Have no fear, there is always a way to stop a program. At the very worst, you can always pull the plug. There is, however, a more elegant way to accomplish the same purpose and retain the contents of memory intact as well. In order to terminate a program that is in an infinite loop, you may press RUN/STOP. This is sometimes referred to as “breaking” the program. Although it, too, is not the most elegant way to terminate a program, it usually does the trick. If the RUN/STOP key doesn’t work, try holding it down and pressing RESTORE (located above the RETURN key). This move has the additional benefit of resetting the screen color to normal and clearing the screen at the same time.

Now study the following two-line program. You will notice that line 100 is the familiar-looking PRINT statement containing a literal—but this time the literal is terminated by a semicolon. In BASIC a trailing semicolon at the end of a PRINT statement suppresses the normal carriage return, which ordinarily causes the subsequent PRINT statement to start at the beginning of a new line. As a result, the next line of text is displayed immediately to the right of the

literal contained by the PRINT statement with the semicolon. When Program 3-3 is run, the whole screen is filled with the literal, which is repeated end-to-end.

PROGRAM 3-3, VERSION 1

```
100 PRINT "MAY THE FORCE BE WITH YOU";
110 GOTO 100
```

RUN

```
MAY THE FORCE BE WITH YOU MAY THE FORCE B
E WITH YOU MAY THE FORCE BE WITH YOU MAY T
HE
```

[and so on]

If you want to improve the appearance of the printout by separating the displayed phrase by a blank, you must insert a space at the end of the literal but before the closing quote. In the next version of the program, a blank space has been included after the word "YOU" in line 100:

PROGRAM 3-3, VERSION 2

```
100 PRINT "MAY THE FORCE BE WITH YOU ";
110 GOTO 100
```

RUN

```
MAY THE FORCE BE WITH YOU MAY THE FORCE
BE WITH YOU MAY THE FORCE BE WITH YOU MA
Y THE
```

[and so on]

Variables and Assignment

The programs illustrated so far have not done anything very exciting. In fact, all they have done is to print out some literals. It is time now to learn how to instruct the machine to compute, which is, after all, the major purpose of a computer.

First, consider a program that computes numerical results. Suppose you are interested in finding the average of 5 and 6. One way to approach this problem is to print the result directly, as follows:

```
PRINT ( 5 + 6 ) / 2
```

An alternate method is to assign the constants 5 and 6 to specific locations in the computer's memory and then apply the required operations to the contents of those memory locations. This concept is used in algebra, where terms like x and y are used to denote unknown quantities, which in BASIC are known as *variables*. These are in contrast to *constants* (such as 5, 3.14159, 2.71828, and "BINGO"), whose values always remain the same. The program that follows assigns the value 5 to the variable X and 6 to the variable Y. Once these values have been assigned, the program averages the two values, producing the desired result.

In BASIC, assigning values is accomplished by the assignment, or LET, statement. In most versions of BASIC available today (and certainly including Commodore BASIC), the word "LET" may always be omitted without any loss of meaning; nor does its omission generate an error of any kind. In the next program two of the assignment statements have the word "LET", whereas the third does not.

PROGRAM 3-4, VERSION 1

```
100 LET X = 5
110 LET Y = 6
120 Z = ( X + Y ) / 2
130 PRINT Z
```

```
RUN
5.5
```

```
READY.
```

It is important to note that even if line 130 were omitted, the value of Z would still be calculated—even though it would not be apparent to the programmer. The only way the computed value of

Z can be displayed is to explicitly instruct the computer to print it by using the PRINT instruction. If the value of Z is not going to be referred to again subsequently, it is possible to bypass the assignment in line 120 and evaluate the result within the PRINT instruction itself, as is shown in the next version of the program.

PROGRAM 3-4, VERSION 2

```
100 LET X = 5
110 LET Y = 6
120 PRINT (X + Y) / 2
```

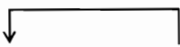
```
RUN
  5.5
```

READY.

Lines 100 and 110 are examples of assignment statements, by which values are stored in variables. It is probably true to say that the assignment statement is the most commonly used statement not just in BASIC but also in most other programming languages. By means of these assignment statements a complex series of computations can be designed, making the program extremely versatile. The program could then be used to solve a host of problems, ranging from a spreadsheet program to an exciting game. In fact, assignment statements are the building blocks from which sophisticated programs are made.

The Structure of an Assignment Statement

The general format of the assignment statement is



line-number LET *variable-name* = *expression*

The value of the expression on the right of the equals sign is evaluated. The expression on the right of the equals sign in an

assignment statement may consist of a single constant or a series of arithmetic operations. Some examples are

```

52
5 + 6
A + B - C + 2 / 3
((2 + 3) ↑ 2)
A * (B + C / 90)
SQR (B ↑ 2 - 4 * A * C)

```

After the expression is evaluated and reduced to a single number it is then stored in the computer's memory, in the location specified by the variable on the left of the equals sign. Thus, although the assignment statement looks like a static instruction, it is in fact dynamic.

As we have said, the line number in a program must always be an integer between zero and 63,999. Since the word "LET" is optional, but when used involves a certain amount of additional work on the part of the programmer, we shall not include it in future programs. However, if you think that its inclusion adds greater clarity to your programs, you should feel free to use it as often as you like.

Variable names must conform to certain rules in order to be acceptable to Commodore BASIC. The first requirement is that they always start with a letter of the alphabet. After the first character, any letter or digit may follow, in any combination, up to a maximum length, which is limited only by the maximum size permitted for program instructions. (On the Commodore 64, it is two lines on the screen, or 80 characters.) Although a variable name may be composed of many characters, only the first two are recognized by Commodore BASIC. That is, the variables RICHARD and RICK are treated as one by the computer since they both start with the letters "RI".

Naming variables is an important element of good programming technique. Of course, most programmers select short names because long names are not only too time-consuming to type but also take up valuable room in memory. There is a certain benefit to be derived from long variable names though, because they help in the docu-

mentation of a program (making it understandable to another programmer). The following are typical valid variable names:

```
DUMMY X SUM1984 SALES QP984Z
```

Here are some examples of some invalid variable names, together with the reasons why they are unacceptable to the computer:

Invalid Variable Name	Reason for Rejection
1984CBM	Name must begin with an alphabetic.
A?B.C	No special characters are permitted.

A variable is also invalid if it is a BASIC command. Therefore, the assignment statements

```
425 LET RUN = 12
430 PRINT = 1
```

are invalid since both the words “RUN” and “PRINT” are keywords. Similarly, any variable name that contains a keyword, for example, “REPRINTED”, is also illegal and generates an error message. A complete list of Commodore BASIC keywords may be found in Appendix A.

In an assignment statement, the variable name must always be followed by an equals sign. It bears repeating that the assignment statement instructs the computer to evaluate the expression that appears to the right of the equals sign, reduces it to a single number (if it is not already a single number), and stores this single value into the computer’s memory in the location specified by the variable name to the left of the equals sign.

Counting

In the next program there are two assignment statements. The first of them sets the value of COUNT to zero. The second adds 1

to the current value of COUNT each time the loop is executed. Within the loop the current value of COUNT is printed out. This program is intended to illustrate an important concept in BASIC programming—that of counting. It is useful when a process must be repeated a given number of times. Study Program 3-5 for a few moments and try to understand what it accomplishes. Pay special attention to line 110, where the variable COUNT appears on both sides of the equals sign.

PROGRAM 3-5

```

100 COUNT=0
110 COUNT=COUNT+1
120 PRINT COUNT
130 GOTO 110

```

RUN

```

1
2
3
4
5
6
7

```

[and so on]

The variable name selected in this program to store the value of the count is appropriately called “COUNT”. It is set to zero in line 100 because in the next line (line 110) 1 is added to it. It is clear that in order for 1 to be added to any variable, it must have a value. The process of setting the starting or initial value for a variable is called *initializing*. Actually, on the Commodore 64 there is no need to initialize variables to zero because they are automatically set to zero when the RUN command is executed. Nevertheless, experience will show that it is a good programming habit to initialize all variables, and we enthusiastically encourage it. Indeed, we shall take this advice ourselves in all the programs illustrated in this book.

The statement in line 110 of Program 3-5, on close scrutiny, looks somewhat ridiculous, particularly when looked at from an algebraic point of view:

```
110 COUNT = COUNT + 1
```

After all, how can COUNT ever be equal to COUNT + 1? The answer is that the statement is an assignment rather than an algebraic equation. As already emphasized, the expression on the right of the equals sign is first evaluated and the result stored in the variable specified on the left of the equals sign. Once COUNT is initialized to zero, 1 is then added to its value, and the result, 1, is stored in the variable on the left. In this case the variable name on the left is also COUNT. What this means then is that the value of COUNT on the right represents its “old” value, whereas that on the left reflects the “current” or updated value. Therefore, it can be said that statement 110 simply means this: add 1 to the current value of COUNT. Whenever a variable is upgraded in this way it must always appear on both sides of the equals sign.

The IF . . . THEN Statement

The previous counting program is, of course, an infinite loop that prints out the numbers 1, 2, 3, 4 . . . ad infinitum. As you will remember, in order to terminate an infinite loop you have to press the RUN/STOP key. But suppose you want the program to terminate automatically as soon as the number 5 has been printed. To accomplish this, the computer’s decision-making capabilities have to be exploited. The BASIC command that enables decisions to be made is called the IF statement, which comes in various formats:

line number IF condition is true THEN GOTO line number

which may be shortened to

line number IF condition is true THEN line number

or may be expressed in an alternate form as

line number IF condition is true GOTO line number

For example, the statement

```
100 IF A = B THEN GOTO 500
```

tests the current values of A and B. If they are equal, control is sent directly to line 500. If they are not equal, the THEN clause is completely ignored and, instead, control drops (falls through) to the next statement. As was just shown, the following three segments of code have the same effect:

```
100 IF A = B THEN 500
100 IF A = B GOTO 500
100 IF A = B THEN GOTO 500
```

Relational Operators

As used in the previous IF statements, the equals sign behaves as a relational operator, comparing the two values A and B, rather than setting the variables equal to each other. There are, of course, other ways to test the relationship between two values. In BASIC they are

Symbol	Meaning
=	equal to
<> or ><	not equal to
>	greater than
=> or >=	greater than or equal to
<	less than
=< or <=	less than or equal to

Here now is the program that prints out the first five numbers, one number per line:

PROGRAM 3-6

```
100 COUNT = 0
110 COUNT = COUNT + 1
```

56 ■ AT HOME WITH BASIC

```
120 PRINT COUNT  
130 IF COUNT < 5 THEN 110
```

```
RUN
```

```
1  
2  
3  
4  
5
```

```
READY.
```

In line 130, a test is made to determine whether the current value of COUNT is less than 5. If it is (meaning that five lines of output have not yet been printed), control is sent to line 110, where the value of COUNT is incremented by 1 and the current value of COUNT is printed out. This process is repeated by the loop. The moment the value of COUNT becomes equal to 5, the test “is COUNT < 5” fails and control drops to the next line. Since there is no next line, BASIC automatically terminates execution of the program.

It is worth pondering for a moment what the effect would be if, in line 130, control were sent to the first statement of the program, line 100, where COUNT is initialized to zero. If this were done, the current value of COUNT would be reinitialized to zero every time around the loop. For this reason, the value of COUNT would always be less than 5. It is therefore an example of an *infinite loop*, so called because it would continue forever.

The INPUT Statement

One of the great advantages of microcomputers, such as the Commodore 64, is that they can be used interactively. That is, during actual execution of the program, information can be fed to the computer. You might want to run a program several times, each time having it work with different data, without modifying the program.

You might want intermediate results printed out to help you trace the course of the program.

You have already seen how values can be assigned to variables by means of the assignment statement. Another method is the INPUT statement, which promotes interactive communication with the computer during the execution of the program. For example, the instruction

```
100 INPUT X
```

prints out a question mark (called the *prompt character* in this context) during execution of the program and waits for the user to type in a value for the variable X. Once this has been done and the RETURN key has been pressed, execution of the program resumes, with X having assumed the value typed in by the user. Until the RETURN key is pressed, no further action is taken by the computer. It will simply wait there doing nothing—forever, if necessary. It is therefore considered a wise practice to precede every INPUT statement with a PRINT statement advising the user (even if that user is you) what kind of information to type in when the prompt appears. For example, if a program computes the area of a rectangle, given its length and width, the PRINT statement requesting this information should contain instructions such as those in lines 100 and 110 in the following program.

PROGRAM 3-7

```
100 PRINT "ENTER LENGTH OF RECTANGLE: ";
110 INPUT L
120 PRINT "ENTER WIDTH OF RECTANGLE: ";
130 INPUT W
140 AREA = L * W
150 PRINT
160 PRINT "THE AREA OF THE RECTANGLE IS:";AREA
```

RUN

```
ENTER LENGTH OF RECTANGLE: ?10
ENTER WIDTH OF RECTANGLE: ?3
```



```
THE AREA OF THE RECTANGLE IS: 30
```

```
READY.
```

We emphasize that it is considered a sound practice to precede all INPUT statements with clear instructions to the user; the author of the program may not be (and often is not) the only person to use the program. Thus familiarity with the program can never be assumed by its author. Even the original authors tend to forget details of their own programs after a while. Moreover, in more complicated programs, several INPUT statements might be necessary, each of which may require completely different information. The careful selection of suitable PRINT statements can make the difference between a successfully run program—and a useless one.

The combination of INPUT and PRINT is so common that Commodore BASIC provides a convenient way to merge the two statements into one. For example, the two lines 100 and 110 in Program 3-7 may be combined into the single statement

```
100 INPUT "ENTER LENGTH OF RECTANGLE: ";L
```

The limitation is that the literal may not be longer than 38 characters. When this instruction is encountered during execution of the program, the literal is printed first and then immediately the prompt character (the question mark), inviting the user to type in the required information—followed, of course, by the use of the RETURN key. No computation is performed; the computer waits indefinitely until the RETURN key is pressed.

You are not confined to a single variable with an INPUT statement. In fact, you may have as many as can fit in a program statement. The only restriction is that each variable name must be separated from the next by a comma. Of course, there must be at least one variable name at all times. Whether one variable name or more are listed, however, only one prompt is displayed for any given INPUT statement when the instruction is executed. In the following program, three values representing the three sides of a triangle are typed in—in response to a single INPUT statement. The area of the

triangle is then computed according to Heron's formula, which is based on the semiperimeter, represented by the variable s :

$$s = \frac{a + b + c}{2}$$

According to Heron's formula, the area of a triangle with sides a , b , and c is

$$\text{Area} = \sqrt{s(s-a)(s-b)(s-c)}$$

PROGRAM 3-8

```

100 INPUT "ENTER THE 3 SIDES: ";A,B,C
110 SEMIPER = (A + B + C) / 2
120 TEMP = SEMIPER * (SEMIPER - A) *
    (SEMIPER - B) * (SEMIPER - C)
130 IF TEMP <= 0 THEN GOTO 170
140 AREA = SQRT(TEMP)
150 PRINT "THE AREA OF THE TRIANGLE IS ";
    AREA
160 END
170 PRINT "THIS IS NOT A VALID TRIANGLE"

```

Although we stated in Chapter 2 that the END statement is ordinarily superfluous, one is included in Program 3-8 to prevent the printing of the error message in line 170. This line is necessary in the event that the values of A, B, and C selected for the triangle are invalid.

The READ, DATA, and RESTORE Statements

On occasion it is necessary to provide a program with data which do not change between different runs of the program. Such data may be stored in a DATA statement and read by a READ statement. In fact, the READ statement must always be used in conjunction with a DATA statement. As with the INPUT, the READ statement

may access more than one item. If more than one variable is listed in the READ statement or more than one item is specified in a DATA statement, they must be separated by commas.

A DATA statement may be placed anywhere at all within a program. It is a special kind of statement in that it is never executed but is merely referred to by the READ statement. A DATA statement may hold as many values as can be placed within the 80-character limit of the line. Moreover, as many DATA statements as are necessary may be included in any given program. If more than one DATA statement is present they may be conceptualized as one continuous list of data items. The DATA statements are accessed by the READ command in the order of their line numbers.

Program 3-9 demonstrates the manner in which data are accessed by a READ instruction:

PROGRAM 3-9

```
100 READ A
110 PRINT A
120 READ B,C
130 PRINT B,C
140 DATA 2,3,4
```

RUN

2

3

4

READY.

The following two programs also demonstrate different ways to structure READ and DATA statements:

PROGRAM 3-10, VERSION 1

```
100 READ A,B,C,D
110 PRINT A,B,C,D
120 DATA 2
130 DATA 6,1,8
```

```
RUN
```

```
2          6          1          8
```

```
READY.
```

```
PROGRAM 3-10, VERSION 2
```

```
100 READ A,B,C,D
110 PRINT A,B,C,D
120 DATA 2,6,1,8
```

```
RUN
```

```
2          6          1          8
```

```
READY.
```

When writing long programs containing DATA statements it is generally a good programming practice to place them at the end of the program so that they may be easily and quickly found by a programmer. Examples of the properties of READ and DATA statements are illustrated further in the following program. We shall discuss them in detail after you have had a chance to study the program.

```
PROGRAM 3-11
```

```
100 READ A,B,C
110 READ D,E,F
120 PRINT F,E,D
130 PRINT B,C,A
140 DATA 10,15
150 DATA 1,2,3,4
```

```
RUN
```

```
4          3          2
15         1          10
```

```
READY.
```

When a READ command is encountered, it accesses the first item of data in the first DATA statement of the program. In line 100 of this example, therefore, the variables A, B, and C take on the values 10, 15, and 1, respectively, since these are the first three values specified in the DATA statements. In line 110, the values 2, 3, and 4 are assigned to the variables D, E, and F, respectively. The READ statement behaves, therefore, in a manner very similar to the assignment statement; in fact, it is simply another option which may be used for assigning values to variables.

As we have said, there is a one-to-one correspondence between the list of variables in the READ statement and the items listed in the DATA statement. This fact implies that for every variable specified by the READ statement there is a value specified in the DATA statement. In Program 3-12, in which the variables A, B, and C are read, only two data items are specified—which leads to the error message

```
?OUT OF DATA.
```

PROGRAM 3-12

```
100 READ A,B,C
110 PRINT A,B,C
120 DATA 10,20
```

```
RUN
```

```
?OUT OF DATA ERROR IN 100
```

```
READY.
```

In this case, the data supplied are insufficient to satisfy the READ instruction in line 100. The program therefore terminates at that point, indicating at which line the READ statement was unsatisfied. In Program 3-13, the opposite case occurs—that is, more data exist than are necessary. As is shown, the excess data are merely ignored and no error message is generated.

PROGRAM 3-13

```

100 READ A,B,C
110 PRINT A,B,C
120 DATA 1.45,2.5,8,999,627,-5,32.345

```

```

RUN

```

```

1.45           2           5

```

```

READY.

```

The Trailer Technique

Suppose a DATA statement contains an unknown number of positive values and you want to find out exactly how many there are. It is possible to have the computer count the number of items present in the DATA statement by terminating the items with a number easily distinguishable from all the other numbers. For example, any negative number could be used. In the next program, the items stored in the DATA statements are counted and subsequently displayed on the screen. The value -1 serves as a signal that all the data items have been read. Since this technique involves a special dummy value, -1 , that “trails” all the way at the end of the data items, it is known as the *trailer technique*. The dummy value is sometimes called an *end-of-data tag*.

PROGRAM 3-14

```

100 COUNT = 0
110 READ A
120 IF A = -1 THEN GOTO 160
130 PRINT A
140 COUNT = COUNT + 1
150 GOTO 110
160 PRINT
170 PRINT "THERE ARE";COUNT;"DATA ITEMS
PRESENT."

```



```
180 DATA 2,55,82,100,2,67,999,890
190 DATA -1
```

```
RUN
2
55
82
100.2
67.999
890
```

THERE ARE 6 DATA ITEMS PRESENT.

READY.

In line 100 the variable COUNT is set to zero, representing the number of items found thus far in the DATA statement. As you will recall, this is known as initializing the variable to zero. Then the first value of A is read. By examining the DATA statement, it will be seen that the first value of A is 2. This value is immediately tested against -1. Since they are not equal, the value 2 must be one of the data items, and control drops down to line 130, where the current value of A, which is equal to 2, is printed out. The value of COUNT is then incremented by 1, and line 150 instructs the computer to go back to line 110, where the next value of A is read. This process repeats itself until the trailing item is read. The test in line 120 then succeeds, and control is sent to line 160, where a blank line is printed, followed by the final line of output, which displays the number of items stored in the DATA statements. Notice that the value -1 is *not* included in this count because when the trailing item is encountered, control is immediately sent to line 160, which skips over both the PRINT and incrementing instructions.

The DATA statement may be viewed as if there is an invisible pointer pointing to the first available item in the first (possibly the only) DATA statement. As soon as that item has been read, it moves over to the next item in preparation for the next READ instruction. It is often extremely useful to be able to reset the pointer to the beginning of the DATA items. For example, if you were interested in calculating the average of the numbers in the DATA statements (using the trailer technique), one method would be to count first

how many items there were (as was done in Program 3-14). The sum of all the items could then be calculated and the result divided by the previously determined number of items present (stored in COUNT). This method requires some way of resetting (restoring) the pointer to its initial position. In BASIC this is accomplished by means of the RESTORE statement. It is used in the next program, where the average of a list of items stored in a DATA statement is calculated.

PROGRAM 3-15

```

100 COUNT = 0
110 READ X
120 IF X = -9999 THEN GOTO 160
130 PRINT X;
140 COUNT= COUNT + 1
150 GOTO 110
160 RESTORE
170 SUM = 0
180 READ X
190 IF X = -9999 THEN GOTO 220
200 SUM = SUM + X
210 GOTO 180
220 AVERAGE = SUM / COUNT
230 PRINT
240 PRINT "THE NUMBER OF ITEMS =" ;COUNT
250 PRINT "THE AVERAGE =" ;AVERAGE
260 DATA 5,6,7,8
270 DATA 9,10
280 DATA -9999

```

RUN

```

 5 6 7 8 9 10
THE NUMBER OF ITEMS = 6
THE AVERAGE = 7.5

```

READY.

In this program, the variable being read is called X, and the trailer item has been arbitrarily defined as -9999. After COUNT has been initialized to zero and the first value of X read, X is tested against

–9999. If they are not equal, control drops to line 130, where the current value of X is printed. You will notice, however, that the PRINT instruction is terminated with a semicolon. This has the effect of printing the value of X in packed format; that is, the numbers are printed on the same line, separated by one space, rather than printed in the four zones produced by the comma. The value of COUNT is then incremented by 1, as one more data item has been read. The process is repeated within the loop until the trailer item (–9999) is encountered. At that point, control is sent to line 160, where the RESTORE instruction is located. The effect of this instruction is to restore the pointer back to the beginning of the DATA statements (line 260) and the variable SUM is then initialized to zero. Each data item is then reread and—as long as it is not equal to –9999—each one in turn is added to the total (stored in the variable SUM). Once the trailing item is encountered for the second time, the average is calculated by dividing the sum of all the items (excluding the trailer) by the number of data items present (again excluding the trailer). The number of items and their average is then printed out and the program is terminated.

Program Storage

When a BASIC program is typed into the Commodore 64, it is stored in its internal memory. However, this is only temporary memory, for if you were to switch the machine off, or if a power failure were suddenly to occur, the complete contents of memory would be lost. This, of course, could be a disastrous situation. In order to store information permanently, some other device must be used. The most common device today is the Datasette (the Commodore digital cassette recorder). In order to store a BASIC program to a cassette, it is necessary to position the tape to a vacant area that is available for storage. This step is followed by the command

SAVE "*program name*"

in which the word “SAVE” is followed by a user-defined program name enclosed in double quotation marks. Any character may be used in the program name, which may be from 1 to 16 characters in length—in contrast to variable names in which the characters are restricted to the letters of the alphabet and the digits 0 through 9. When saving to tape, the program name may be omitted, but this is not a good idea as it makes it impossible to retrieve the program by name.

Once the program has been saved to tape, the computer may be switched off without any fear of losing information. For all intents and purposes, the program is saved—forever. Any time subsequently, the saved information may be retrieved from the cassette and loaded into memory. In Commodore BASIC the command to load a previously saved program is

LOAD “*program name*”

where “program name” is the name under which the program was saved.

Saving Programs to Disk

For those users fortunate enough to own a disk drive, saving programs is even easier than to tape. The identical commands are used, LOAD and SAVE, with the exception that they must be followed by a device specification. Since the disk drive has been assigned the device number 8, the commands appear as

SAVE “*program name*”,8

and

LOAD “*program name*”,8

There are additional commands that are required for a disk drive, however. First, to save any information, the disk on which it is to

be stored must be initialized. This is accomplished by the somewhat cryptic command

```
OPEN 15,8,15,"N0:disk name,ID"
```

where "disk name" is the name you wish to call the disk and "ID" is an arbitrary number. An example of this statement is

```
OPEN 15,8,15,"N0:BING0,1"
```

which initializes the disk, preparing it to store programs and other data. This process takes some time, so a little patience is required.

Another additional and extremely useful command is

```
LOAD "*" ,8
```

which loads in the names of all the programs residing on the disk and, when listed, identifies them and the amount of space each occupies.

Review Questions

- 1 ■ What is a loop?

A loop is a segment of code that is executed repeatedly.

- 2 ■ Which instruction sends control unconditionally to another statement?

The GOTO statement.

- 3 ■ Can a GOTO statement branch to a line number greater than or less than the current one?

Certainly.

- 4 ■ What is the name given to the constant upward move of information off the screen?

Scrolling.

- 5 ■ What is an infinite loop?

An infinite loop is a loop that never ends.

- 6 ■ How may an infinite loop be terminated?

By hitting the RUN/STOP key.

- 7 ■ What is the effect of terminating a PRINT statement with a semicolon?

It suppresses the carriage return, allowing the next PRINT statement to continue at the current print position.

- 8 ■ What is a variable?

A variable is a symbolic name for a location in memory which holds a value. The value may be changed at any time.

- 9 ■ What is a constant?

A value that always remains the same—such as 5, 7.29, -2.9901, 3.14159, and “HELLO”

- 10 ■ What statement assigns the variable A to 10 in Commodore BASIC?

```
LET A = 10
```

- 11 ■ Is the word “LET” required by Commodore BASIC?

No, it is optional.

- 12 ■ In order for the computer to display a value, what instruction must be present?

The PRINT instruction.

- 13 ■ May a BASIC instruction have a line number of 0?

Yes. The valid range is 0 through 63999.

70 ■ **AT HOME WITH BASIC**

14 ■ Which of the following variable names are valid?

- a. BINGO
- b. 1984SALES
- c. PROFIT/LOSS
- d. X229FFCX
- e. SOCIAL.SECURITY.NUMBER
- f. AAABBBCCDDDEEEFFFGGGHHHIIJJJ1234567890

- a. Invalid (contains the keyword GO of GOTO).
- b. Invalid (must not begin with a digit).
- c. Invalid (no special characters allowed).
- d. Valid.
- e. Invalid (no special characters allowed).
- f. Valid (although only the first two letters are significant).

15 ■ What is the maximum length of a variable name?

A variable name has no maximum length. It must, however, fit on a line consisting of no more than 80 characters.

16 ■ What symbol is always present in an assignment statement?

The equals sign.

17 ■ What is the meaning of the instruction

```
TOTAL = TOTAL + 1
```

Add 1 to the variable TOTAL.

18 ■ What is meant by the term *initializing*?

When a variable is initially set to some value before it is used in a computation, it is said to be initialized.

19 ■ If a variable is not initialized, how does the Commodore 64 treat it?

As though it were initialized to zero.

20 ■ What is the name of the instruction that permits a decision to be made in BASIC?

The IF . . . THEN statement.

- 21 ■ Write an equivalent statement to the following:

```
IF A = B THEN GOTO 177
```

Either

```
IF A = B THEN 177
```

or

```
IF A = B GOTO 177
```

- 22 ■ What is the purpose of the END statement in a program?

To force the termination of the run at the point where it is placed, thereby avoiding executing sections of the program that should not be processed.

- 23 ■ What is printed by the following program:

```
100 READ X
110 IF X > 15 THEN 140
120 IF X > 50 THEN 150
130 IF X > 75 THEN 160
140 PRINT "BINGO"
150 PRINT "SHMINGO"
160 PRINT "ZINGO"
170 DATA 45
```

```
BINGO
SHMINGO
ZINGO
```

```
READY.
```

When *X* is tested against 15 in line 110, control is sent directly to line 140. After executing the PRINT instruction in line 140, control continues to the following statements in the ordinary way.

72 ■ AT HOME WITH BASIC

24 ■ What are the symbols for
a. greater than? b. less than? c. not equal to?

a. > b. < c. <> or ><

25 ■ What is printed by the following program:

```
100 READ X, X, X, X
110 PRINT X, X, X, X
120 DATA 1, 2, 3, 5

5            5            5            5

READY,
```

Each of the four values, 1, 2, 3, and 5, are successively read into the variable X. Each time a new X is read, the previous one is replaced. Therefore, when X is printed out it is only the last value which is displayed four times.

26 ■ Where may DATA statements be placed in a program?

Anywhere. They do not affect the way a program is executed.

27 ■ If a program contains a READ statement, what else must it contain?

A DATA statement.

28 ■ If a READ statement contains a list of five variables, how many data items should there be?

At least five.

29 ■ If a READ statement contains a list of ten variables, how many DATA statements must be present?

At least one. However, it must contain at least ten data items.

- 30 ■ What do the READ, INPUT, and assignment statements have in common?

They each assign values to variables.

- 31 ■ What is the effect of having an insufficient number of data items for a READ instruction?

An ?OUT OF DATA ERROR message is displayed and execution of the program is terminated.

- 32 ■ What is the effect of the RESTORE statement?

The RESTORE statement returns the data pointer to the first item of the first DATA statement in a program.

- 33 ■ How would you save a program called "PAYROLL" to tape?

By using the command

```
SAVE "PAYROLL"
```

- 34 ■ How would you retrieve the same program from tape?

By using the command:

```
LOAD "PAYROLL"
```

HANDS-ON PRACTICE

1. Type in the following program. Before running it, come to your own conclusion as to what the output should be.

```
100 PRINT "BASIC MEANS POWER TO THE  
PEOPLE";  
110 GOTO 100
```

2. Determine what the following program prints and why. How does it end?

74 ■ AT HOME WITH BASIC

```
100 X = 1
110 X = X + 1
120 GOTO 110
```

3. Try setting some variables in immediate mode. For example, type

```
X = 5
PRINT X
```

Reassign a new value to X and print it out again.
What do you notice?

4. Type in and run the following program:

```
100 X = 1
110 PRINT X
120 X = X + 1
130 IF X < 100 THEN GOTO 110
```

Determine ahead of time what the output should be.

TRY YOUR HAND AT THESE

1. Write a program that prints out the value of X, X², and X³, where X is a user-typed value (use the INPUT statement).
2. Write a program to print the square root of the numbers from 1 through 100 (use a loop and make sure it terminates correctly).
3. Write a program that computes the Celsius temperature, given the Fahrenheit equivalent: $C = 5/9(F - 32)$.
4. Write a program that reverses the temperatures in question 3: $F = 9/5C + 32$.
5. Write a program to compute the simple interest accrued by investing \$4,000 for 6 years at the rate of 7¼ percent per annum.

```
FV = PV * (1 + RATE/100) ↑ NUMYEARS
```

Structured Programming

In this chapter, you will learn some of the critical tools of modern computer programming, including

- the principles of structured programming
- the need to eliminate GOTO statements
- the FOR . . . NEXT loop
- the empty FOR . . . NEXT loop
- trapping invalid data
- multiple-line statements
- determining the sum of the integers from 1 to N
- the STEP clause
- internal documentation
- the Newton-Raphson iteration scheme
- introduction to subroutines

Programming has been in existence for a little less than 40 years. Before 1947 no modern electronic computer had yet been invented. Even in the early 1950s programming was an arcane art practiced by only a handful of PhD's in the most prestigious universities in the country. It is from those years that the myths of the evil, all-encompassing computer spring.

The explosive growth of computers caused by vastly improved, ever more complex, and cheaper electronic parts took the world by surprise. There developed a crucial shortage of computer program-

mers, a shortage which continues to the present day. It was in the first of the boom days that the programmers' credo, under pressure from their managers, began as "get it done—no matter what the cost."

Since those crude beginnings, a whole new science has sprung up around computers. With the evolution of computers and the passing of time, the philosophies of programming have made radical shifts.

The first of these changes is the attitude toward "getting it done." No longer is emphasis placed on getting the program to work immediately but rather on setting it up in an orderly fashion. The reason for this change is that the program should be easily amended if necessary and, equally important, be readily understood and modified by another programmer. The techniques used in modern-day programming are incorporated into a style called *structured programming*. In essence, it is a collection of suggestions and guidelines regarding programming practices.

There are several major principles of structured programming. One of the most important is that large problems should be broken up into a series of smaller problems, each of which is more easily manageable (a sort of "divide and conquer" approach). Each subproblem is coded separately into a so-called "module," which is then integrated into the program as a whole.

Another important principle, called *top-down* programming, is to make each individual module as simple as is feasible. This requirement involves a flow of control, starting from the "top" of the program and descending "down" to the bottom as directly as possible.

Although BASIC generally is not the most suitable language for structured programming, some special enhancements have been added to the Commodore dialect that promote this favored style. All these enhancements, together with the standard structured BASIC features, are covered in great detail in this chapter.

Looping

The GOTO statement has come under considerable criticism by professional programmers because, in the short history of computer

programming, it has been abused more than any other instruction. Trying to follow the logic of a program that incorporates a long sequence of GOTOs is both complex and exasperating. If you read the following program, which includes no more than five GOTO statements, you will get an idea of what confusion can result from the abuse of this instruction. The program certainly works, but that is hardly any justification for its complexity.

PROGRAM 4-1

```
100 GOTO 150
110 GOTO 130
120 GOTO 110
130 PRINT "THIS IS NOT A STRUCTURED PROGRAM"
140 GOTO 160
150 GOTO 120
160 END
```

Although this is an exaggerated example, the fact is that most programs contain loops. In fact, it is felt by many programmers that a program that does not contain a loop is not worth writing in the first place. However, a leading authority on computer programming and one of the main proponents of structured programming, Edsger Dijkstra, is of the opinion that the quality of a program is inversely proportional to the number of GOTOs present in it. How are we to reconcile his stand on the GOTO statement with the necessity for loops in every program?

As one answer to this question, Commodore BASIC provides special constructs that preclude the use of the GOTO statement in some cases. The first of these is the FOR . . . NEXT loop.

However, before getting involved with this loop, study the following program carefully and try to determine for yourself what it does.

PROGRAM 4-2, VERSION 1

```
100 I = 1
110 PRINT "HI THERE"
```

```
120 I = I + 1
130 IF I > 3000 THEN 150
140 GOTO 110
150 END
```

It is clear that the literal "HI THERE" is printed many times. The question is, how many times? Is the answer 2,999, 3,000, 3,001—or perhaps some other number? The exact number can be deduced by replacing the number 3000 by a smaller and more manageable number such as 3. The program can then be traced by "playing computer," and whatever applies to 3 will apply equally to 3000. Since it will become clear that when line 130 compares I against 3 (rather than 3000) a total of 3 lines are printed, it therefore means that when 3000 is placed in line 130, exactly 3,000 lines are printed.

The FOR . . . NEXT loop

In computer programming it is often necessary for a sequence of instructions to be executed a specific number of times. The method just employed above does not make this number of repetitions immediately apparent to the average reader. BASIC does provide, however, a streamlined equivalent to this segment of code. Instead of the six lines of code required by using an IF . . . THEN statement and worse—the dreaded GOTO statement—the following program accomplishes exactly the same purpose; but it is clearer, leaves no room for ambiguity, and moreover requires fewer lines, and therefore means less work. In short, it is a far more elegant solution to the problem.

PROGRAM 4-2, VERSION 2

```
100 FOR I = 1 TO 3000
110 PRINT "HI THERE"
120 NEXT I
130 END
```

The END statement has been included only for the sake of consistency with version 1—where it was necessary to have a line for the GOTO to send control. In version 2, the END statement is superfluous but not incorrect.

The format for the FOR . . . NEXT loop is

line number FOR *index variable* = *starting value* TO *final value*

[body of loop]

line number NEXT *index variable*

The index variable name used with the FOR statement must be the same as that used in the NEXT statement, although in Commodore BASIC the variable name in the NEXT statement is optional. All the instructions beginning with the FOR statement and ending with the NEXT statement are regarded as the body of the loop.

When a FOR . . . NEXT loop is encountered in a program, the starting and ending values are immediately computed and are stored for the duration of the loop. The starting value is then placed in the index variable. A test is then made to determine whether the index variable is greater than the ending value. If it is, the body of the loop is skipped over, and control is sent to the statement immediately following the NEXT statement. If the starting value is less than or equal to the ending value, the body of the loop is executed. On reaching the NEXT statement, the index variable (called I in Program 4-2, version 2) is automatically incremented by 1, and the process is repeated until the index variable is, indeed, greater than the ending value. As soon as this occurs, the loop is terminated and control is sent to the statement following the NEXT statement. In this way, controlled looping may occur without resort to the GOTO statement.

Here are some of the rules governing the FOR . . . NEXT loop.

1. The index variable (sometimes called the control variable) is just another variable in the program. Within the

loop its value may be used in the same manner as any other variable; it may be printed and used in a calculation. As a rule, however, its value should not be changed—this should be left to the automatic operation of the FOR . . . NEXT loop itself. By changing the value of the index (as in the following example)

```
100 FOR I = 1 TO 100
110 PRINT I;
120 I = 206
130 NEXT I
```

the value of the variable I suddenly becomes 206 after executing line 120. When 206 is compared to the final value in the FOR statement (which is 100), the loop is terminated immediately, since 206 is greater than 100. This confusing situation should always be avoided.

2. On exiting from a FOR . . . NEXT loop the value of the index variable is not the final value but the final value plus the increment. This is true for most versions of BASIC.
3. Entry to a FOR . . . NEXT loop should always be made through the FOR statement. The body of the loop should not be entered from outside the loop, as illustrated in the following example:

```
100 FOR I = 1 TO 3
110 PRINT I;
120 NEXT I
130 GOTO 110
```

RUN

```
1 2 3 4
?NEXT WITHOUT FOR ERROR IN 120
```

READY.

4. There is a nesting limit of nine levels. That is, nine FOR . . . NEXT loops may be encased within another. However, care should be taken that all the index variables used have different names. (Nested loops will be discussed in detail shortly.)
5. If nested loops are used their ranges should not overlap.

The following program illustrates a simple application of a FOR . . . NEXT loop. It prints out the value of the index I , I^2 , and the square root of I , for all values of I ranging from 1 to 10. This program shows that the index of a loop may be used in a computation, as may any other ordinary variable.

PROGRAM 4-3

```
100 FOR I = 1 TO 10
110 PRINT I,I * I,SQR(I)
120 NEXT I
```

RUN

1	1	1
2	4	1.41421356
3	9	1.73205081
4	16	2
5	25	2.23606798
6	36	2.44948974
7	49	2.64575131
8	64	2.82842713
9	81	3
10	100	3.16227766

READY.

If the starting value of the index is greater than the ending value, the loop is executed once—with the starting value. This operation may be seen in the following program, where the literal “WHAT’S GOING ON?” is printed, once.

PROGRAM 4-4

```

100 FOR I = 10 TO 1
110 PRINT "WHAT'S GOING ON?"
120 NEXT I

```

```

RUN
WHAT'S GOING ON?

```

```

READY.

```

In the next program, there are no statements forming the body of the loop. Despite the fact that this omission may seem to be an error, such loops are commonly used to create delays. For example, a delay might be necessary in a program in which the user has to view the contents of the screen before proceeding to the next step. For the message to remain on the screen long enough to be read and understood, a delay loop could be inserted in the program. As fast as the computer is, it still takes some finite time to execute a FOR . . . NEXT loop—so programmers frequently include an “empty loop” to slow the computer down when it threatens to exceed the capacity of the human eye to keep track of what’s being displayed. On the Commodore 64, a loop ranging from 1 to 10,000 takes approximately 15 seconds to execute. (Don’t forget, when writing a program never use commas to separate the thousands from the rest of the number, as in, for example, 10,000. Its inclusion will always result in an error message.)

PROGRAM 4-5

```

1000 FOR I = 1 TO 10000
110 NEXT I

```

In the following program, the user is asked to type in a positive integer, N. The program then proceeds to calculate the sum of the integers from 1 to N, using a FOR . . . NEXT loop.

PROGRAM 4-6, VERSION 1

```

100 SUM = 0
110 INPUT "ENTER YOUR VALUE FOR N: ";N
120 FOR I = 1 TO N
130 SUM = SUM + I
140 NEXT I
150 PRINT "THE SUM OF THE INTEGERS FROM 1
    TO";N;"IS:";SUM

```

RUN

```

ENTER YOUR VALUE FOR N: 6
THE SUM OF THE INTEGERS FROM 1 TO 6 IS: 21

```

READY.

This program may be criticized because it allows a user to type in a nonpositive value for N. For example, values such as -7 and -2 are not acceptable for this problem. It does not require much ingenuity, however, to include a “trap” in the program, to “catch” any such user errors. An appropriate message could be printed, stating the nature of the error, should it be made. This step has, in fact, been taken in the next version of the program, where in line 120 the value of N is tested to be sure that it is greater than zero. If it is, control passes to line 150. If it's not, a warning message is printed (line 130), and the INPUT statement is repeated.

PROGRAM 4-6, VERSION 2

```

100 SUM = 0
110 INPUT "ENTER YOUR VALUE FOR N: ";N
120 IF N>0 THEN GOTO 150
130 PRINT "SORRY, N MUST BE POSITIVE"
140 GOTO 110
150 FOR I = 1 TO N
160 SUM = SUM + I
170 NEXT I
180 PRINT "THE SUM OF THE INTEGERS FROM 1
    TO";N;"IS:";SUM

```


84 ■ AT HOME WITH BASIC

RUN

```
ENTER YOUR VALUE FOR N: -3
SORRY, N MUST BE POSITIVE
ENTER YOUR VALUE FOR N: 10
THE SUM OF THE INTEGERS FROM 1 TO 10 IS: 55
```

READY.

Once again, the program deserves some criticism. The astute reader will notice that although you have just learned how to avoid the GOTO in looping, the last program contains not one but two GOTOs. You shall now learn a new construct which eliminates one of the GOTOs. The other one cannot easily be eliminated. In general, BASIC programmers cannot totally escape the use of the GOTO statement, however much they may want to; it simply must be used sparingly and judiciously.

PROGRAM 4-6, VERSION 3

```
100 SUM = 0
110 INPUT "ENTER YOUR VALUE FOR N: ";N
120 IF N <= 0 THEN PRINT "SORRY, N MUST BE
    POSITIVE":GOTO 110
130 FOR I = 1 TO N
140 SUM = SUM + I
150 NEXT I
160 PRINT "THE SUM OF THE INTEGERS FROM 1
    TO";N;" IS:";SUM
```

Multiple-Line Statements

In version 3 of the program, which functions identically to version 2, the THEN clause in line 120 contains two BASIC statements on the same line. The word "THEN" may be followed by any number of valid statements, provided they are separated by colons and do

not exceed the maximum permissible line length of 80 characters. This multiple statement is executed only if the test proves to be true. If the test proves false, all the statements following the word THEN are skipped over. This type of *multiple-line statement* (more than one command on a given line number) is legal, not only in an IF . . . THEN statement, but also in any other normal program line. For example, the statements

```
100 PRINT "DO YOU KNOW"
110 PRINT "WHERE YOUR CHILDREN ARE?"
```

which print the two literals on separate lines, may be condensed into the single instruction

```
100 PRINT "DO YOU KNOW":PRINT "WHERE YOUR
      CHILDREN ARE?"
```

which also prints the literals on two separate lines.

You have already seen how the sum of the integers from 1 to N may be computed with a FOR . . . NEXT loop. Although the problem is useful as a demonstration of the manner in which the FOR . . . NEXT loop works, it is not the most efficient way to calculate the sum. A far better method is one attributed to the famous German mathematician and astronomer Karl Friedrich Gauss (1777 - 1855), who discovered it at the tender age of seven. He proved that the sum of the integers from 1 to N may be computed directly by the formula

$$\text{Sum} = \frac{N(N + 1)}{2}.$$

The following and final version of the program which utilizes this formula is by far the most efficient. Not only is it shorter to write but also it uses less computer time because, regardless of the value of N, only one addition, one multiplication, and one division are required. For large values of N, the saving in time over the FOR . . . NEXT method is considerable.

PROGRAM 4-6, VERSION 4

```

100 INPUT "ENTER YOUR VALUE FOR N: ";N
110 IF N <= 0 THEN PRINT "SORRY, N MUST BE
    POSITIVE":GOTO 100
120 SUM = N * (N + 1) / 2
130 PRINT "THE SUM OF THE INTEGERS FROM 1
    TO";N;"IS:";SUM

```

Again, the values produced are identical to those shown for version 2. The difference between the two versions lies in the manner (and corresponding efficiency) with which the result is calculated.

We would now like to return to the FOR . . . NEXT loop and consider several features not yet discussed. We have already explained that when the NEXT instruction is executed, the value of the index is incremented by 1. This does not always have to be the case, however. The full FOR statement contains a STEP clause following the ending value, specifying the increment that is added to the index each time around the loop. If the STEP clause is omitted, the increment is assumed to be 1. This option was deliberately provided by the designers of the language, since 99 percent of FOR . . . NEXT loops do indeed use an increment of 1.

If a FOR . . . NEXT loop uses a step other than 1, the number of times the loop is executed may be calculated exactly. Assuming the general form

$$\text{FOR } I = J \text{ TO } K \text{ STEP } L$$

the number of times the loop is executed is given by the formula:

$$\text{INT} \left(\frac{K - J}{L} \right) + 1$$

In the following program, the sum of the integers from 1 to N is first calculated by using the Gauss formula. Then by means of two separate FOR . . . NEXT loops, the sum of the odd integers from 1 to N and the sum of the even integers from 2 to N are computed. These two sums are then added together and compared to the result produced by the Gauss formula.

PROGRAM 4-7

```

100 INPUT "ENTER YOUR VALUE FOR N: ";N
110 IF N <= 0 THEN PRINT "SORRY, N MUST BE
    POSITIVE":GOTO 100
120 SUM = N * (N + 1) / 2
130 ODDSUM = 0
140 FOR I = 1 TO N STEP 2
150 ODDSUM = ODDSUM + I
160 NEXT I
170 EVENSUM=0
180 FOR I=2 TO N STEP 2
190 EVENSUM = EVENSUM + I
200 NEXT I
210 IF SUM <>EVENSUM + ODDSUM THEN PRINT
    "THERE'S AN ERROR SOMEWHERE...":END
220 PRINT "THE SUM OF THE EVEN NUMBERS =";
    EVENSUM
230 PRINT "THE SUM OF THE ODD NUMBERS =";
    ODDSUM
240 PRINT "THE GAUSSIAN TOTAL =";SUM

```

RUN

```

ENTER THE VALUE FOR N: -7
SORRY, N MUST BE POSITIVE
ENTER THE VALUE FOR N: 8
THE SUM OF THE EVEN NUMBERS = 20
THE SUM OF THE ODD NUMBERS = 16
THE GAUSSIAN TOTAL = 36

```

READY.

The STEP value need not be confined to a positive, or whole number. In the following program, a step value of .25 is used. The program prints out all the values between 1 and 3 in steps of .25.

PROGRAM 4-8

```

100 FOR K= 1 TO 3 STEP .25
110 PRINT K;
120 NEXT K

```

RUN

```
1 1.25 1.5 1.75 2 2.25 2.5 2.75 3
```

READY.

The values are printed in packed format because of the semicolon placed at the end of the PRINT statement in line 110.

The next example demonstrates the capability of the FOR . . . NEXT loop to count backwards. It merely prints out (in packed format) the values of the index as it proceeds from 5 to -4 in steps of -1.

PROGRAM 4-9

```
100 FOR Q = 5 TO -4 STEP -1
110 PRINT Q;
120 NEXT Q
```

RUN

```
5 4 3 2 1 0 -1 -2 -3 -4
READY.
```

Although the examples shown thus far have all contained specific constants for the starting, ending, and step values of the index, these values need not be specified explicitly. Each one of these three values may be replaced by an expression. For example, the statement

```
100 FOR K = A + 2 * B TO Q ↑ 2 / (4 * X
+ L) STEP .4 / 7 * J
```

is perfectly valid, as indeed is the simpler statement

```
100 FOR I = X TO Y STEP Z
```

Internal Documentation of Programs

Although the programs illustrated so far are sufficiently straightforward so as not to require any special explanation, industry-level

programs can be very long and complex. In such programs, it is the responsibility of the author to include adequate documentation of the program to enable any other programmer to modify it if necessary. We have already shown by example how to create variable names that are self-descriptive (ODDSUM rather than X22FFS12 or some such nonsense). Commodore BASIC permits the inclusion of explanatory remarks anywhere at all in the program. These remarks may be inserted by means of the REM statement, which REMinds you of the logic of the program so that you will REMember the important features. Whatever follows the REM statement is regarded as a REMark and is ignored by the computer, except that it is printed out in a listing of the program. Nothing in a REM statement affects the execution of a program in any way.

In addition to internal documentation (which does not appear during execution of the program), another commendable practice is to print a descriptive heading before any other output is produced by the program. At the option of the author, the author's name, address, and telephone number, as well as the date the program was written, may also be included. Both internal and external documentation are considered to be primary attributes of structured programming. The following program, which performs several calculations and prints the result, contains samples of such internal documentation—several REM statements.

PROGRAM 4-10

```

100 REM: DEMONSTRATION OF THE REM STATEMENT
110 A = 1
120 B = 2
130 C = 1
140 X1 = (-B + SQR(B ↑ 2 - 4 * A * C)) /
      (2 * A)
150 X2 = (-B - SQR(B ↑ 2 - 4 * A * C)) /
      (2 * A)
160 REM BOTH ROOTS HAVE NOW BEEN COMPUTED
170 REM NOW THEY ARE PRINTED
180 PRINT "THE ROOTS ARE:",X1,X2
190 REM: THE END

```

90 ■ AT HOME WITH BASIC

```

RUN
THE ROOTS ARE:      -1          -1

READY.

```

In the following program a simple and quite interesting conjecture is examined. Given any positive integer, the conjecture states that if it is even, it should be divided by 2. However, if it is odd, it should be multiplied by 3 and 1 added to the result. Whichever the case may be, the procedure is repeated in this way until 1 is reached. According to the conjecture, all positive integers converge to 1 when treated in this manner. The truth is, it has never been proven to be true or false. Perhaps you can find a value for which it will not work. Should you try, good luck! In any case, it provides an excellent showcase for illustrating the use of structured logic.

PROGRAM 4-11

```

100 PRINT "*****"
110 PRINT "*"
120 PRINT " CONVERGENCE-TO-1 CONJECTURE "
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 INPUT "PLEASE TYPE IN A POSITIVE INTEGER: ";N
170 IF N <= 0 THEN PRINT "SORRY, N MUST BE
    POSITIVE":GOTO 160
180 PRINT N;
190 IF N=1 THEN PRINT:PRINT "THE CONJECTURE
    HOLDS":END
200 IF N / 2 = INT(N / 2) THEN N = N / 2:
    GOTO 180
210 N = 3 * N + 1
220 GOTO 180

```

```

RUN
*****
*
* CONVERGENCE-TO-1 CONJECTURE *
*
*****

```

```
PLEASE TYPE IN A POSITIVE INTEGER: 14
 14 7 22 11 34 17 52 26 13 40 20 10
5 16 8 4 2 1
THE CONJECTURE HOLDS

READY.
```

The Newton-Raphson Iteration Scheme

The next program illustrates further some of the features of structured programming. As you are aware, we have often resorted to the SQR function to calculate the square root of an expression. Whenever a square root is required, the computer has to follow systematically a sequence of instructions expressly designed to calculate the square root of any given value. Such a sequence of instructions is, in fact, stored in the computer. In the next program, a nearly identical method is used to calculate the square root of any inputted number. The result is then checked against the built-in SQR function to convince the programmer that the method really does work.

The method used is attributed to Sir Isaac Newton and a contemporary of his named Raphson. According to the Newton-Raphson technique, a guess is made at the square root of the number. The closer the guess is to the actual square root, the quicker the square root is found. The process involves continually calculating new and more accurate guesses by an "iteration" process, that is, one involving a loop. The loop is terminated when the latest guess is accurate to within some predetermined accuracy, say four decimal places. If the number whose square root you wish to find is X , and you make an initial guess, GUESS, at its square root, the new guess is given by the formula

$$\text{GUESS} = (\text{GUESS} + X / \text{GUESS}) / 2$$

To find out if the newly generated value of GUESS is, in fact, the required square root, you have to test whether the absolute value

(given by the function ABS) of the guess squared divided by X minus 1 is less than some small value (usually called epsilon):

$$\text{ABS}(\text{GUESS} \uparrow 2 / X - 1) < .000001$$

The reason for this is that if GUESS is truly close to the square root, when it is squared it should be extremely close to the original value stored in X. This being the case, when 1 is subtracted from it, the result must be very close to zero. If it is so close that it is less than .000001, you accept that value of GUESS as a very good approximation of the square root—at least, to the sixth decimal place.

PROGRAM 4-12

```

100 PRINT "*****"
110 PRINT "*"
120 PRINT "*  NEWTON-RAPHSON ITERATION SCHEME  *"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 INPUT "PLEASE ENTER YOUR VALUE FOR X: ";X
170 INPUT "AND NOW YOUR INITIAL GUESS: ";GUESS
180 IF ABS(GUESS ↑ 2 / X - 1) < .000001 THEN 210
190 GUESS = (GUESS + X / GUESS) / 2
200 GOTO 180
210 PRINT "THE SQUARE ROOT OF";X;"IS:";GUESS
220 PRINT "USING THE SQR FUNCTION, THE RESULT
    IS:";SQR(X)

```

RUN

```

*****
*
*  NEWTON-RAPHSON ITERATION SCHEME  *
*
*****

```

```

PLEASE ENTER YOUR VALUE FOR X: 123
AND NOW YOUR INITIAL GUESS: 43
THE SQUARE ROOT OF 123 IS: 11.0905375
USING THE SQR FUNCTION, THE RESULT IS: 11.0905365

```

READY.

An Elementary Introduction to Subroutines

When programs are written in structured style they are divided into separate modules, each of which has its own clearly defined role to play. Generally, the first or “main” routine triggers the secondary routines. Each of these may in turn call on other routines, which may be used to subdivide further a complex problem. According to structured-programming advocates, programs written in this highly disciplined approach are error-resistant; accurate; faster to write, execute, and understand; and, therefore, easier to maintain.

In BASIC, program segments designed to be modules may be called into action in what are known as *subroutines*. A subroutine is “called” or invoked by the GOSUB statement. Following the keyword GOSUB is a line number indicating where in the program the subroutine is located. If it is at line 1000, the calling instruction would read

```
line number GOSUB 1000
```

The last executable statement in the subroutine must be a RETURN statement, which returns control to the “calling” routine, in particular to the statement immediately following the GOSUB instruction. If the same subroutine is called from different points in the program (as is often done to great advantage) the RETURN statement always returns control to the statement immediately following the GOSUB that invoked it, a feat that GOTO statements are utterly unable to emulate. The fundamental difference between the GOTO and the GOSUB statements is that the former sends control unconditionally to a given location in the program, as specified by the line number. On the other hand, when a GOSUB statement is used, the computer “remembers” the location containing the GOSUB with the understanding that it will return there (or, at least, to the statement following it) when the RETURN statement is encountered.

Subroutines are particularly useful when a segment of code is used at many different places within the program. In such situations, the subroutine need appear only once in the program, regardless of how

often it is called. This element not only lends itself to shorter programs but also speeds up the process of writing them. However, as was previously mentioned, subroutines are useful as modules, even if they are not repeatedly executed. The following program is a simple illustration of subroutines.

PROGRAM 4-13

```

100 PRINT "*"
110 PRINT "*"
120 PRINT "* FIRST ILLUSTRATION OF SUBROUTINES *"
130 PRINT "*"
140 PRINT "*"
150 PRINT
160 PRINT "A";
170 GOSUB 1000
180 PRINT "B";
190 GOSUB 1000
200 PRINT "C";
210 GOSUB 1000
220 END
1000 REM
1010 REM: SUBROUTINE PRINTS A LINE OF ASTERISKS
1020 REM
1030 PRINT "*****"
1040 RETURN

```

RUN

```

*****
*
* FIRST ILLUSTRATION OF SUBROUTINES *
*
*****

```

```

A *****
B *****
C *****

```

READY.

After printing out the heading in lines 100 through 150, this program proceeds to print the literal “A” in line 160. At that point, the subroutine that begins in line 1000 is invoked by the statement “GOSUB 1000.” Control is immediately sent to line 1000, and the subroutine prints the line of asterisks. Upon encountering the RETURN statement in line 1040, control is returned to the statement following 170, which is 180. Line 180 prints the literal “B” and is followed by line 190, which again invokes the subroutine in line 1000. The asterisks are printed out once again and control returns to the line after the calling of GOSUB—this time, line 200. The same process is repeated after printing the literal “C.” Admittedly, this is a rather contrived example, but later on, when you have covered more material, subroutines will be exploited to great advantage.

Review Questions

- 1 ■ According to the proponents of structured programming, what is the most important aspect in the creation of a program?

A program must be easy to understand and modify. Even if it takes much longer to create a program in structured style, it saves time in the long run.

- 2 ■ What is the output of the following program:

```
100 FOR W = 1 TO 3
110   PRINT W;W * 2
120 NEXT W
```

RUN

```
1 2
2 4
3 6
```

READY.

- 3 ■ In the following FOR . . . NEXT loops, how many times will the body of the loops be executed?

- a. 100 FOR ZERO = 1 TO 5
110 NEXT ZERO
- b. 100 FOR DEG = -3 TO 3
110 NEXT DEG
- c. 100 FOR J = 5 TO 1 STEP .1
110 NEXT J
- d. 100 FOR CYNDY = 3 TO 6 STEP .25
110 NEXT CYNDY
- e. 100 FOR SIT = 1 TO 5 STEP .5
110 NEXT SIT
- f. 100 FOR NYU = 3 TO 1 STEP -.75
110 NEXT NYU

- a. 5 times.
 - b. 7 times (the variable degree takes on the values -3, -2, -1, 0, 1, 2, and 3).
 - c. 1 time (the starting value of the index variable J is already greater than the ending value, and the step value is positive).
 - d. 13 times.
 - e. 9 times.
 - f. 3 times. The index takes on the values 3, 2.25, and 1.5. When it becomes .75 (which is less than 1), the loop terminates.
- 4 ■ Assuming that A, B, C, and D have the values 2, 3, 6, and .5, respectively, what output would you expect to the following program segment?

```
100 IF A < B THEN PRINT "A IS LESS THAN
    B"
110 IF B + C < D THEN PRINT A;B;C;D
120 IF A ↑ 2 < B ↑ 2 THEN PRINT "IS THAT
    SO?"
```

```
A IS LESS THAN B
IS THAT SO?
```

```
READY.
```

- 5 ■ How many lines of output are printed by the following program segment?

```
100 PRINT "T'WAS BRILLIG":PRINT "AND THE
    SLITHY TOGES"
110 PRINT "DID GYRE AND GIMBOL":PRINT
    "IN THE WABE"
```

Four lines (note the number of print statements).

- 6 ■ Write an equivalent but shorter version of the statement

```
FOR I = 1 TO 10 STEP 1
```

```
FOR I = 1 TO 10
```

- 7 ■ What is the purpose of the REM statement?

The REM statement is used for internally documenting a program. It is displayed in the listing of a program but in no way affects its execution.

- 8 ■ How is a subroutine invoked?

By means of the GOSUB statement.

- 9 ■ What is the essential difference between the GOTO and GOSUB statements?

The GOTO is an unconditional transfer of control to a given line number. From that point, execution of the program continues in its normal, sequential fashion (unless redirected by another GOTO). With the GOSUB instruction, however, a

boomerang effect takes place. After control is sent to the specified subroutine, the subroutine's instructions are executed sequentially until the RETURN statement is encountered. Then control is returned to the calling routine—in particular, to the line following the GOSUB instruction.

- 10 ■ How many times may a subroutine be called?

There is no limit to the number of times a subroutine may be called.

- 11 ■ What must be the last executable statement in a subroutine?

The RETURN statement.

- 12 ■ What is the maximum number of subroutines that may appear in a program?

The number of subroutines is restricted only by the available memory.

HANDS-ON PRACTICE

1. Type in and run the following program:

```
100 FOR I = 10 TO 1
110 PRINT I
120 NEXT I
130 PRINT "HOW ABOUT THAT?"
```

Why does it execute the body of the loop only once?

2. Examine the following program and determine the different values that the index assumes. Confirm your conclusion by running the program.

```
100 FOR I = 44 TO 110 STEP 5
110 PRINT I
120 NEXT I
```

TRY YOUR HAND AT THESE

1. Determine what the following program does and rewrite it in a simpler form. Run both programs to confirm your conclusions. (Hint: One statement is never executed.)

```

100 GOTO 150
110 GOTO 130
120 PRINT "RIDICULOUS":END
130 GOTO 120
140 PRINT "EQUALLY RIDICULOUS"
150 GOTO 110

```

2. Rewrite the following programs in a more structured form using the FOR . . . NEXT loop.

```

a. 100 I = 1
    110 I = I + 2.5
    120 PRINT I
    130 IF I <= 10 THEN 110
b. 100 R = 2.5
    110 PRINT R
    120 R = R+.5
    130 IF R > 5.5 THEN 150
    140 GOTO 110
    150 END

```

3. Write a program that asks the user to type in a number representing a height measured in inches. The program should print that number and its equivalent in feet, yards, and miles. (Note: There are 12 inches to the foot, 3 feet to the yard, and 1,760 yards to the mile.)
4. Assuming there are 2.54 centimeters to the inch, convert an inputted value of centimeters to its equivalent in yards, feet, and inches.
5. Write a program that accepts as input two values: the price and the amount of tax on a purchase. Calculate the total cost for each sale and print out the relevant details.

6. The sum of \$2,000 is invested for a period of five years in a savings account which pays annual interest of 6 percent. Write a program to calculate the interest accrued and the final value. (Hint: Use a FOR . . . NEXT loop).
7. Modify program 5 so that the user is free to type the principal, time period, and interest rate of one's choice.
8. Write a program to sum the squares of the integers from M to N, where M and N are both user-inputted values.
9. Write a program to calculate the first N terms of the Fibonacci sequence, where each number (not including the first two seed numbers) is the sum of the two preceding ones, as follows: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, and so on.
10. Modify the program specified in question 8 to print out all the numbers in the sequence that are less than N.
11. Write a BASIC program that evaluates the algebraic equation

$$y = \frac{3x^2 + 5(x + 4)^3 + 3}{3x + 19}$$

for values of x ranging from 1 to 10 in steps of 0.25. Print out the values of x and the corresponding values of y in a table containing suitable headings. Once the table is printed, print a message saying "this is the end of the table."

12. Write a program that "sings" the famous drinking song "bottles of beer." A sample stanza is given:

3 bottles of beer on the wall,
 3 bottles of beer;
 if one of those bottles should happen to fall—
 2 bottles of beer on the wall.

The program should count down from 10 to 1.

13. Write a program that computes the sum of the series

$$\frac{1}{4} + \frac{1}{7} + \frac{1}{10} + \frac{1}{13} + \dots + \frac{1}{301}$$

14. Write a program to compute the sum of the series

$$\frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \dots + \frac{1}{11!}$$

(The exclamation mark denotes factorial. Factorial 3 for example, written as 3!, is $3 \times 2 \times 1$. Similarly, 6! is $6 \times 5 \times 4 \times 3 \times 2 \times 1$. Factorial 0! is defined as 1.)

This series is equivalent to

$$\frac{1}{1} + \frac{1}{1} + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} + \dots + \frac{1}{39916800}$$

(If you are mathematically inclined, you may find that the value obtained is of special interest. It is, in fact, the universal constant e , which has a value of 2.718281828. . . .)

15. Write a program to determine if a number is even or odd.
 16. Write a program to determine if an inputted number is prime or not. (A number is considered to be prime if it is not evenly divisible by any whole number except itself and 1. So that 17, 11, and 7 are all prime numbers, whereas 28 is not, being divisible by 14, among others.)
 17. Write a program to compute the average weekly wage earned by five employees. The number of hours worked and rate per hour are given and should be stored in DATA statements within the program.

Number of Hours Worked	Rate of Pay
10	5.50
42	3.35
57	10.92
28	8.85
3	50.65

18. Modify the program you wrote for question 17 so that any employee who worked more than 40 hours is paid at time-and-

a-half. (Round off the pay to the nearest dollar before printing it.) At the end, determine the total pay earned by all the employees and display it at the end of the program after skipping four lines.

Numeric Functions and Logical Operators

In this chapter, you will become familiar with many of the mathematical tools which make Commodore BASIC a powerful programming language. Even if you are not particularly math-oriented, it is a good idea to familiarize yourself with all the math tools that the Commodore 64 provides. Many programming techniques are possible only by using these mathematical tools. If you do not, as yet, know trigonometry, some of the functions in this chapter may seem meaningless to you. However, they are included here for the sake of completeness, and if necessary, you may always refer to this chapter as the occasion demands. In particular, you will learn about

- the SQR function
- the INT function
- the ABS function
- the SGN function
- the trigonometric functions (SIN, COS, TAN, ATN)
- the logarithmic functions (EXP, LOG)
- the RND function
- the logical operators (AND, OR, NOT)
- Boolean values
- the complete hierarchy of the operators
- swapping two values
- user-defined functions

In general, certain procedures are performed so frequently that BASIC provides the programmer with built-in functions (also called library functions) that enable quick computation with a minimum of effort. You have already encountered the SQR function, which yields the square root of an expression—without a written program. All that is necessary is the library function, SQR, followed by a pair of parentheses in which is enclosed the nonnegative expression whose square root is required. For example, to store the square root of 169 into the variable X, all you need write is

```
X = SQR(169)
```

Printing X displays the square root of 169, which is 13. Fortunately for us, the Commodore 64 provides many other useful functions which can be accessed just as easily. Every function follows the same design. They all consist of three letters followed by parentheses, into which the “argument” (the value to be operated on by the function) is included. The argument can be an expression,

```
PRINT SQR(50 + 25 + 12 + 6 + 3 + 1 + 3)
10
READY.
```

or it may involve variables:

```
X = 2
READY.
Y = 3
READY.
PRINT SQR(X ↑ 4 + Y ↑ 2)
5
READY.
```

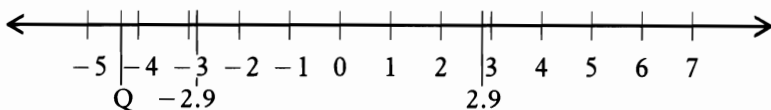
There are a total of 11 numeric functions built into the Commodore 64. One particularly useful routine (as you will discover when your knowledge of programming expands) is the INT function. This is known mathematically as the “greatest integer” function (in some other computer languages, as the “FLOOR command”) and is defined as returning the highest integer value that is less than the argument. For example,

```
PRINT INT(2.3)
```

displays the truncated value 2, because 2 is the largest integer that is less than 2.3. Similarly,

```
Q = -4.2
PRINT INT(Q), INT(-2.9), INT(2.9)
```

displays the values -5, -3, and 2 in the first three print zones. The reason for this may be seen by examining the following number line:



The values returned by the INT function can be visualized as the first integer immediately to the left of the value in question on the number line.

An example of a problem that uses the INT function is in determining how many quarters are contained in a given sum of money. You know, for example, without having to resort to a computer that \$4.50 is equivalent to 18 quarters. However, if you have an amount such as \$3.20, its equivalent value in quarters is 12, with a remainder of 20 cents. The number of quarters that can “fit” into a given sum of money is given by the general formula

```
INT (amount / .25)
```

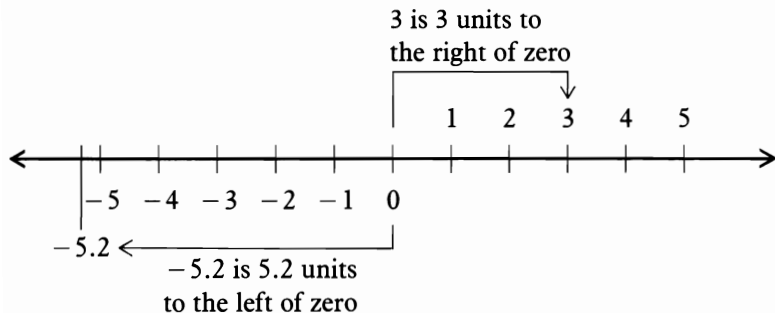
so that the number of quarters in \$3.20 is

$$3.20 / .25 = 12.8$$

Since you cannot have .8 of a quarter, the answer is arrived at by taking the greatest integer less than 12.8, which is 12. Indeed, 12 is the integer portion of 12.8.

The ABS Function

Another commonly used routine (one you already have come across but only in passing) is the ABS (absolute) function. The absolute value of a number is always positive. In fact, it represents its distance away from zero on the number line, as illustrated in the following number line where the absolute values of -5.2 and 3 are shown:



For example, if a tourist visiting New York City's famous Fifth Avenue walks from Seventeenth to Thirtieth Street, you would say that the distance covered is $30 - 17$ (which is 13) blocks. On the other hand, if the visitor were to walk back from Thirtieth to Seventeenth Street, by that argument the distance covered would be $17 - 30$ (which is -13) blocks. Obviously, the distance is the same in both directions; what we are interested in is the absolute value of

the difference—namely, 13 blocks. You may notice that an interesting property of absolute values is that the distance may be expressed in two different ways that yield the identical results:

$$\text{ABS}(\textit{start} - \textit{finish})$$

and

$$\text{ABS}(\textit{finish} - \textit{start})$$

The SGN Function

On occasion it is useful to determine the sign of an expression. This is the purpose of the SGN (signum) function, which returns the value -1 for any negative number, 0 for a value of zero, and $+1$ for any positive value. It is illustrated in the following program.

PROGRAM 5-1

```

100 PRINT "*****"
110 PRINT "*"
120 PRINT "* ILLUSTRATION OF THE SGN FUNCTION *"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 SUM = 0
170 READ A
180 IF A = -9999 THEN 210
190 SUM = SUM + SGN(A)
200 GOTO 170
210 IF SGN(SUM)=1 THEN PRINT "THERE ARE";SUM;
    "MORE POS. THAN NEG. NUMBERS":END
220 IF SGN(SUM) = -1 THEN PRINT "THERE ARE";
    -SUM;"MORE NEG THAN POS NUMBERS":END
230 PRINT "SAME NUMBER OF POS. AND NEG. NUMBERS"
240 DATA 5,4,3,-2,-1,0,-3,-4,-5,-9999
    
```



```
RUN
```

```
*****
*                                     *
*   ILLUSTRATION OF THE SGN FUNCTION *
*                                     *
*****
```

```
THERE ARE 2 MORE NEGATIVE THAN POSITIVE NUMBERS
```

```
READY.
```

In this program the first value of *A* is 5. Since 5 is positive, *SGN(A)* yields the value 1. This value is added to the value of *SUM*, which was initialized to zero. Each time a negative number is read, *SGN(A)* yields the result of -1 , which is also added to *SUM*. If more negative than positive numbers are included in the *DATA* statement (as is the case in this example) the final value of *SUM* is less than zero. To make this value more meaningful, its absolute value is taken by the *ABS* function, thus converting it to a positive number.

The Trigonometric Functions

The commonly used trigonometric functions sine, cosine, and tangent are also available by using the library functions *SIN*, *COS*, and *TAN*. In each case, the argument specifies the angle in radian measure. The only inverse function ordinarily available is the *ATN*, or arctangent function, which returns the arctangent (in radians) of the argument. In the following table, some typical trigonometric expressions are evaluated:

Expression	Value Returned
<i>SIN</i> (0)	0
<i>SIN</i> (3.1415926535)	0
<i>COS</i> (0)	1
<i>COS</i> (3.1415926535)	-1
<i>TAN</i> (0)	0
<i>TAN</i> (<i>ATN</i> (2))	2

In the last example shown, since TAN and ATN are inverse functions, they cancel each other out and return the original argument. Other less commonly used trigonometric functions have not been implemented into Commodore BASIC.

Logarithmic Functions

The natural logarithm of a number is found by the LOG function. Mathematically, it returns the power to which the universal constant e is raised to equal the argument. This may be written as

$$e^x = \text{argument}$$

or as it is often expressed

$$x = \ln(\text{argument})$$

where x is the value returned and e is approximately equal to 2.71828183. Examples of the LOG function in BASIC follow:

```
PRINT LOG(1)
Ø

READY.
PRINT LOG(2.71828183)
1

READY.
```

It is often desirable to take a logarithm to some base other than e . The so-called common logarithm, for example, has a base of 10. To convert from the natural logarithm to any other base, the following formula may be used:

$$x = \frac{\text{LOG}(y)}{\text{LOG}(b)}$$

where x is the logarithm of y to the base b . Written in BASIC, therefore, the common logarithm of 100 can be computed by

```
PRINT LOG(100) / LOG(10)
2
READY.
```

The inverse function of LOG is EXP, which raises e to the power specified by the argument, as shown in the following examples:

```
PRINT EXP(0)
1
READY.
PRINT EXP(1)
2.71828183
READY.
```

The Random Function (RND)

For many types of numerical computation and for many computer games as well, the computer's ability to generate random numbers—numbers chosen from the computer's "hat"—can be very useful. The function RND generates a random number between 0 and 1 (not including 1) to provide this important feature. With the RND function, the value of the argument is unimportant; only its sign is examined. The first and simplest form of the random function is the instruction

```
PRINT RND(X)
```

(where X is any positive number), which displays a random value such as

```
.328780872
```

The fact of the matter is, however, that these random numbers are not truly random; they are "pseudo-random." Each time the

computer is turned on and a random number is generated by the RND function, the same number will always result. This is equivalent to knowing in advance who will win a horse race or how a thrown die will roll. In fact, the series of random values is predictable not only for the first number but also for any sequence of numbers since these numbers are generated by a standard formula stored within the computer. For most purposes, the sequence of numbers generated by this method is random enough and quite satisfactory. If it is necessary to start the sequence from a different number each time the program is run, the sequence can be made to start from a different point. This is known as “reseeding” the random number generator and is accomplished by the RND function with an argument of 0:

```
PRINT RND (0)
```

This statement uses the RND(0) function to restart the random number sequence from an unknown point, making it more random since we can no longer predict where in the sequence it is. The value returned by RND(0) is also between 0 and 1 (not including 1), but its value depends on how long the computer has been switched on. This value is measured in such fine units that it is impossible for a human to predict what the number will be.

The RND function provides one other option. If the argument of the function is made negative, a certain value will be returned. The returned value differs for each argument used, but it will always be the same for the same negative argument. For example, the statement

```
PRINT RND (-1)
```

displays the value

```
2.99196472E-08
```

and will do so every time the statement is executed, whereas the statement

```
PRINT RND(-4)
```

always returns the value

```
2.99214662E-08.
```

The capability of the computer to produce random numbers is indeed one of its most fascinating features. In the following program, a series of coin flips is simulated by using the random function generator. Any number between 0 and .5 (not including .5) is regarded as "heads" and all other numbers as "tails." The user enters a value for N which specifies the number of flips to be simulated.

PROGRAM 5-2

```
100 PRINT "*****"
110 PRINT "*"
120 PRINT "          COIN FLIPPING"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 INPUT "HOW MANY FLIPS";N
170 HEADS = 0:TAILS = 0
180 FOR I = 1 TO N
190 IF RND(1) < .5 THEN HEADS = HEADS + 1:
    GOTO 210
200 TAILS = TAILS + 1
210 NEXT I
220 PRINT "THERE WERE";HEADS;"HEADS AND";
    TAILS;"TAILS."
230 PH = HEADS / (HEADS + TAILS) * 100
240 PT = TAILS / (HEADS + TAILS) * 100
250 PRINT "THE PERCENTAGE OF HEADS =" ;PH
260 PRINT "THE PERCENTAGE OF TAILS =" ;PT
```

RUN

```
*****
*
*          COIN FLIPPING
*
*****
```

HOW MANY FLIPS? 10

THERE WERE 4 HEADS AND 6 TAILS

THE PERCENTAGE OF HEADS = 40

THE PERCENTAGE OF TAILS = 60

READY.

Logical Operators

If two IF . . . THEN statements send control to the same statement number, they may be combined into one statement by the logical operators. On the Commodore 64 there are three such operators, called "AND," "OR," and "NOT." Each one performs what are known as logical or Boolean operations on numeric values. A logical operator takes either one or two true or false values and returns a single true or false result. An operand of a logical operator is considered to be "true" if it is not equal to zero. If it is equal to zero, it is considered to be "false." The following analogies will help to clarify these concepts.

The logical operator AND is used exactly as it is in everyday English. Suppose, for example, Johnny is promised a new bike if he takes out the garbage on a regular basis and also does well in school. In order to qualify for the bike he is going to have to succeed in both chores. The inclusion of the word *and* implies simultaneity—at the same time. If he does well in school but forgets to dispose of the garbage, he has to go without the promised bike. If he remembers to take care of the garbage but flunks school, again he is out of luck. Suppose, for example, you want the user to type in a positive integer value for N in response to an INPUT statement. The test ensuring that N is both positive and an integer could be done in two separate IF . . . THEN statements:

```
100 IF N > 0 THEN GOTO 250
  ⋮
250 IF N = INT(N) THEN GOTO 400
```

If both conditions are “true” simultaneously, control is sent to line 460. These separate statements may be combined into a single statement (whose meaning is more obvious) by using them with the logical operator AND in the following way:

```
100 IF N > 0 AND N = INT(N) THEN GOTO 460
```

If either N is not greater than zero or N is not equal to the integer portion of N, the statement is false, and control falls through to the next statement in line (presumably to line 110). This would also happen if both conditions were false. For the whole statement to be true, both conditions have to be true. (In logic, the AND operator is called the *conjunction operator*.)

Another frequently encountered logical situation is that of the *inclusive or*. In this case, only one of the conditions has to be true for the whole expression to be true. If they are both true, again the whole expression is true. Thus, this version of the OR statement is called the “inclusive OR”—it includes the case where both conditions are true. It is used in a manner identical to that in which the word *or* is used in English. For example, if Mary is promised a new phonograph if either (1) she does the dishes on a regular basis or (2) gets an A in all her courses in school, she only has to satisfy one of the two requirements to qualify for the phonograph. (Of course, the chances are very good that being the kind of person that she undoubtedly is, she will both do the dishes and excel in school.)

The third logical operator, NOT, unlike the others, uses a single true or false value and returns a single true or false value. The NOT operator reverses the truth value of the expression to its right. For example, the statement

```
100 IF NOT (X > 1) THEN 560
```

transfers control to line number 560 if the value of X is not greater than 1.

The manner in which all these logical operators work may be illustrated by the following truth tables. In the tables, T stands for true and F for false. If you study the table you will notice that it

reveals in a nutshell everything that we have said concerning the AND, OR, and NOT logical operators. For example, if you want to know the result of combining a false with another false, using the AND logical operator, you will see from the table under the caption AND that false and false results in false. The same is true when they are combined with the OR logical operator. You can confirm this yourself by again looking at the table. The NOT operator works in a slightly different manner since it operates on only one value. Again, by looking at the table it will immediately be seen that the NOT operator reverses the truth value of the expression to its right. So that NOT false is true and NOT true is false.

AND			OR			NOT	
A	B	Result	A	B	Result	A	Result
F	F	F	F	F	F	T	F
F	T	F	F	T	T	F	T
T	F	F	T	F	T		
T	T	T	T	T	T		

Boolean Values

Despite outward appearances, the instruction

```
PRINT 5 = 6
```

does have meaning in BASIC, which treats the statement as a command to examine the question is $5 = 6$? The answer to this question is either "yes" or "no." There is no room in computer science for "maybe." Clearly, the assertion is false because 5 is not equal to 6. Since this is so, the computer prints 0, which is the value it associates with false. The implied assertion in the command

```
PRINT 5 > 4
```

similarly must be either true or false. (It is, of course, true; 5 is greater than 4). The computer associates the value -1 with true, in

the same manner that it associates 0 with false. In fact, any nonzero value is also regarded as true on the Commodore, even though -1 is the only value that is printed. These types of values—where there are only two possible alternatives—are known as Boolean values, named after the famous British logician George Boole.

Since -1 is associated with true by Commodore BASIC, the statement

```
10 IF -1 THEN 500
```

also has meaning. This instruction always transfers control to line 500 since the IF condition is always satisfied. Although this statement may appear to be worthless, as any GOTO behaves identically, such statements can be useful in special situations where a variable is substituted for the constant. For example, the program segment

```
100 L = 2
110 X = L < 5
120 IF X THEN PRINT "L IS LESS THAN 5"
```

prints the literal "L IS LESS THAN 5" only if the variable X is true. Since L is defined as 2, and 2 is indeed less than 5, the variable X assumes the logical value to be true—causing the literal to be printed. It is pointed out that the Commodore 64 also treats any nonzero value (1, 5, -19 , and so on) as representing true.

An Overview of the Order of Operations

Thus far, we have discussed the order of operations in connection with the arithmetic operators only. The notion may be considerably expanded, however, as we consider all the types of operators we have now covered.

In all cases, parentheses override any built-in precedence. That is, putting parentheses around an expression ensures that it is evaluated first. If there is more than one set of parentheses, the innermost is evaluated first, extending outward.

After parentheses come the following:

- function calls (such as SQR, SIN, and so on)
- exponentiation (\uparrow)
- the unary minus (for example, -9)
- multiplication ($*$) and division ($/$) (on the same level)
- addition ($+$) and subtraction ($-$) (on the same level)
- relational operators [$<$, $<=$, $=<$, $=$, $>=$, $>$, $<>$, $><$]
- the NOT operator
- AND
- OR

For example, the statement

```
X = SQR(144) + SQR(3 + 6) * ABS(-6 / 3)
      - 4 ↑ (2 * SGN(11)) < 5 OR
      INT(123.4) > 10 ↑ 2
```

reduces to the following stages during execution:

```
X = 12 + 3 * 2 - 4 ↑ (2 * 1) < 5 OR 123
      > 10 ↑ 2
```

```
X = 12 + 3 * 2 - 4 ↑ 2 < 5 OR 123 > 10
      ↑ 2
```

```
X = 12 + 3 * 2 - 16 < 5 OR 123 > 100
```

```
X = 12 + 6 - 16 < 5 OR 123 > 100
```

```
X = 2 < 5 OR 123 > 100
```

```
X = -1 OR -1
```

```
X = -1
```

Since -1 is the Commodore 64's way of storing the value true, X is evaluated to be the value true. Admittedly, this is a rather involved example, but it does possess the virtue of covering the order of

operations in its entirety. Although it may at first glance seem confusing to the uninitiated, working through this example step by step will prove to be most informative.

The Pendulum Problem

The period of a pendulum, or the time it takes for one complete swing, is given by the formula

$$T = 2\pi \sqrt{\frac{L}{g}}$$

where T is measured in seconds, π is the constant 3.141593, L is the length of the pendulum (measured in feet), and g is the acceleration due to gravity (usually given as 32 feet per second per second). The longer the length of the pendulum, the longer it takes for a complete swing. (Tie a heavy object to a piece of string and confirm for yourself that the longer the string, the longer each swing takes.)

Suppose you wanted to print out a table of results showing the relationship between the length of the pendulum and its period—with the length ranging from a given minimum to a given maximum, in steps of a given increment. The following program, using the familiar FOR . . . NEXT loop, produces such a table. There are two additional points in the program worth noting. The first is the use of consecutive commas in the PRINT statement at line 200, which has the effect of skipping over the second zone, so that the column heading matches with the displayed data. The reason this step is necessary is because each zone consists of ten positions, whereas the first heading—"LENGTH (IN FEET)"—contains more than ten characters. It thus overflows the zone and causes the comma to skip to the next available zone. The second point is the curious spelling of the variable `LNPTH`. It would be more reasonable to spell this variable `LENGTH`. This is not possible on the Commodore 64, however, because it contains the keyword `LEN`—

a function you shall come across when dealing with character strings.

PROGRAM 5-3

```

100 PRINT "*"*****"
110 PRINT "*"
120 PRINT "*" THE PENDULUM PROBLEM VERSION 1 "*"
130 PRINT "*"
140 PRINT "*"*****"
150 PRINT
160 PI = 3.141593:G = 32:REM SET CONSTANTS
170 INPUT "ENTER YOUR MIN, MAX AND INCREMENT: ";
MIN,MAX,INC
180 PRINT"LENGTH (IN FEET)", "PERIOD (IN SECS)":
PRINT
190 FOR LGTH = MIN TO MAX STEP INC
200 PRINT LGTH, 1/2 * PI * SQR(LGTH / G)
210 NEXT LGTH

```

RUN

```

*****
*
* THE PENDULUM PROBLEM VERSION 1 *
*
*****

```

ENTER YOUR MINIMUM, MAXIMUM, AND INCREMENT: 1,5,.5

LENGTH (IN FEET) PERIOD (IN SECS)

1	1.11072086
1.5	1.36034967
2	1.5707965
2.5	1.75620388
3	1.92382496
3.5	2.07796845
4	2.22144171
4.5	2.35619475
5	2.48364734

READY.

It is worth contemplating what the result would be if this above program were run with the minimum and maximum typed in in reverse order. In that case, since the starting value of the FOR . . . NEXT loop would be larger than the ending value, the loop would be executed once and once only. On the Commodore 64 a FOR . . . NEXT loop is always executed at least once, regardless of the beginning and ending values specified in the FOR statement. Since it is easy to inadvertently reverse two values, such as the minimum and the maximum, a programming solution should be sought to detect and possibly correct such an error. To detect such an error, all that need be done is to include a simple IF statement to determine if MIN is greater than MAX. Should this be the case, the two values stored in MIN and MAX should be switched. At first blush, it would appear that this may be accomplished by means of the following two statements:

```
MIN = MAX
MAX = MIN
```

Unfortunately, because control always operates sequentially, this step places the value stored in MAX in both MIN and MAX, as is shown in the following “game play”:

Action Taken	MIN	MAX
(none)	10	4
MIN = MAX	4	4
MAX = MIN	4	4

In a similar manner, the instructions

```
MAX = MIN
MIN = MAX
```

stores the value that is in MIN into both MIN and MAX. Therefore, neither method is successful. What is required is for a third location to be used into which either MIN or MAX is temporarily stored. Let's call such a location “HOLD”:

HOLD = MIN
 MIN = MAX
 MAX = HOLD

This time, a copy of the value in MIN is first stored in HOLD so that it is not lost when the value of MAX is placed in MIN. The success of this technique will be seen in the following game play, which uses the same values as before:

Action Taken	HOLD	MIN	MAX
(none)	(undefined)	10	4
HOLD = MIN	10	10	4
MIN = MAX	10	4	4
MAX = HOLD	10	4	10

←————→
 MIN and MAX
 are now swapped

This swapping technique is incorporated into the next version of the program, which first insures that the three inputted values are positive and then checks to be sure that the minimum is less than the maximum. If it is not, the values of MIN and MAX are swapped, through the previous technique.

PROGRAM 5-4

```

100 PRINT "*****"
110 PRINT "*"
120 PRINT "* THE PENDULUM PROBLEM VERSION 2 *"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 PI = 3.141593:G = 32:REM SET CONSTANTS
170 INPUT "ENTER YOUR MIN, MAX AND INCREMENT: ";
    MIN,MAX,INC
180 IF MIN <= 0 OR MAX <= 0 OR INC <= 0 THEN
    PRINT "POS. VALUES ONLY":GOTO 170
190 IF MIN > MAX THEN HOLD = MIN:MIN = MAX:MAX
    = HOLD:PRINT "ERROR CORRECTED"
    
```

```

200 PRINT"LENGTH (IN FEET)" ; "PERIOD (IN
    SECS)";PRINT
210 FOR LNGTH = MIN TO MAX STEP INC
220 PRINT LNGTH ; 2 * PI * SQR(LNGTH / G)
230 NEXT LNGTH

```

This program serves several important purposes. First, it demonstrates some of the critical programming techniques that you have covered so far. These include swapping two values, the multiple-IF statement, the multiple-line statement, use of the SQR function, use of the STEP clause in a FOR . . . NEXT loop, the legitimate use of the GOTO statement, and the practice of assigning meaningful names to the constants and variables used in a program. It will become clear to you that these same techniques can be applied to any formula, regardless of the discipline from which it derives, without your even knowing the basis for the formula. No matter what the formula is, the programmer can in most cases easily and quickly write a program to produce a massive table of results, if need be.

User-Defined Functions

Although Commodore BASIC provides many useful built-in functions, occasions will arise when there is no function available for the particular operation you wish to perform. For such purposes, you are at liberty to define your own function. Once it has been defined, it may be used in much the same way as are the built-in functions. In Commodore BASIC, the definition of a function is limited to a single statement. The first step in defining a user-function is to name it. A function name consists of any valid variable name and must be placed in a definition statement of the form

DEF FN *function name* = *expression*

For example, to define a function called "CUBEROOT" that computes the cube root of 7, we may write

```
100 DEF FN CUBEROOT = 7 ^ (1 / 3)
```

Every time the cube root of 7 is required, all that is necessary is to write

```
FN CUBEROOT
```

Thus, the statement

```
PRINT FN CUBEROOT
```

displays the cube root of 7. Obviously, the usefulness of this function as written is limited by the fact that it always computes the cube root of 7; never the cube root of any other expression. Without such flexibility there is no inherent advantage to using the function instead of a variable in which the cube root of 7 has been stored. The power of user-defined functions is that they may have arguments just as built-in functions do.

The form of a function with arguments is

```
DEF FN function name (argument) = expression
```

Although in the DEF (definition) statement they must be valid variable names, the arguments do not affect the main program variables in any way; they merely provide a method for defining the computation to be performed by the function. For example, the statement

```
100 DEF FN CUBEROOT(X) = X ↑ (1 / 3)
```

does not affect the value in the variable X that appears in the main program because X is used here only to define the function. The way the function actually works is quite simple. Each time it is invoked, the argument must be supplied by the main program. For example, the instruction

```
200 PRINT FN CUBEROOT(5)
```

substitutes the constant 5 for every occurrence of the variable X in the function definition. The function may then be conceptualized as


```
DEF FN CUBEROOT = 5 ↑ ( 1 / 3 )
```

and the value is displayed.

In the next program, a user-defined function called "POLYNOME" is used to compute the value of the function for the integers from 1 to 5.

PROGRAM 5-5

```
100 PRINT "*****"
110 PRINT "*"
120 PRINT "* USER-DEFINED FUNCTIONS EXAMPLE 1 *"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 DEF FN POLYNOME(X) = X ↑ 3 - 5 * X ↑ 2 + 72
    * X - 1
170 PRINT "NUMBER", "VALUE OF FUNCTION"
180 FOR I = 1 TO 5
190 PRINT I, FN POLYNOME(I)
200 NEXT I
```

RUN

```
*****
*
* USER-DEFINED FUNCTIONS EXAMPLE 1 *
*
*****
```

NUMBER	VALUE OF FUNCTION
1	67
2	131
3	197
4	271
5	359

READY.

Although this program is somewhat artificial, it does, nevertheless, demonstrate the proper use of user-defined functions. In this

case, it would have been just as easy to place the expression where the value is required (in line 190). Ideally, the function should play a more realistic role by replacing what would otherwise entail many more involved expressions.

In the next program, user-defined functions are used to advantage because certain calculations (in this case, converting between radian and degree measure) are performed more than once. Instead of typing in the conversion factor at each point where it is required, a call to the user-defined function is made. You will notice that the constant π is used in both definitions of the functions. On the Commodore 64, the value of π is accessible directly by means of the π key.

PROGRAM 5-6

```

100 PRINT "*****"
110 PRINT "*"
120 PRINT "    MORE USER-DEFINED FUNCTIONS"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 DEF FN DEGTRAD(X) = X *  $\pi$  / 180
170 DEF FN RADTDEG(X) = X * 180 /  $\pi$ 
180 INPUT "ENTER TWO ANGLES IN DEGREES: ";A;B
190 PRINT "ANGLES ARE";FN DEGTRAD(A);"AND";FN
    DEGTRAD(B);"RADIANS."
200 PRINT"THE SINE OF";A;"DEGREES IS";SIN(FN
    DEGTRAD(A))
210 PRINT"THE COSINE OF";B;"DEGREES IS";COS(FN
    DEGTRAD(B))
220 PRINT:PRINT
230 INPUT "NOW TYPE IN TWO ANGLES IN RADIANS: ";
    C;D
240 PRINT "ANGLES ARE";FN RADTDEG(C);"AND";FN
    RADTDEG(D);"DEGREES."

RUN
*****
*
*    MORE USER-DEFINED FUNCTIONS
*
*****

```

126 ■ AT HOME WITH BASIC

```
ENTER TWO ANGLES IN DEGREES: ? 45,30
ANGLES ARE .785398163 AND .523598775 RADIANS
THE SINE OF 45 DEGREES IS .707106781
THE COSINE OF 30 DEGREES IS .866025404
```

```
ENTER TWO ANGLES IN RADIANS: ? .5,1
ANGLES ARE 28.6478898 AND 57.2957795 DEGREES
```

READY.

Review Questions

- 1 ■ What must surround the argument of any built-in or user-defined functions?

Parentheses must enclose the argument of every function.

- 2 ■ What function returns the square root of an expression?

SQR

- 3 ■ What is the effect of the statement

```
PRINT SQR(5 - 100)
```

An **ILLEGAL QUANTITY ERROR** is produced. (The SQR function accepts only nonnegative arguments.)

- 4 ■ What is printed by the following statements?

a. PRINT SQR(5 + 20)

b. PRINT INT(5.29)

c. PRINT SQR(INT(135.6 + 9.104))

d. PRINT ABS(-4), ABS(100 - 144),
ABS(INT(-2.6) * 2)

e. X = 10 - SGN(-10)

f. PRINT SGN(-52 * -2)

- a. 5 b. 5 c. 12 d. 4 44 6
- e. Nothing is printed out because the line does not include a PRINT statement. However, the value of X is 11.
- f. 1 (Two negative numbers multiplied together yield a positive result. The SGN function returns the value 1 for any positive value.)

- 5 ■ What angular measure is assumed by the trigonometric functions implemented in Commodore BASIC?

All the trigonometric functions on the Commodore 64 assume radian measure.

- 6 ■ When the square-root function is used, what is assumed?

That the argument is either zero or positive.

- 7 ■ The LOG function built into Commodore BASIC uses what base?

The base used is the mathematical constant e , or 2.718.

- 8 ■ What is the mathematical name of the preceding logarithm?

The natural logarithm.

- 9 ■ How can the logarithm of the variable X to, say, base 2 be derived in Commodore BASIC?

$$L = \text{LOG}(X) / \text{LOG}(2)$$

- 10 ■ What is the range of numbers returned by the RND function?

$$0 \leq n < 1$$

- 11 ■ How may the integers between 1 and 6 be generated with the RND function?

$$N = \text{INT}(\text{RND}(1) * 6) + 1$$

- 12 ■ Describe the actions taken by the following separate statements:

- a. 10 IF 2 < 3 AND 5 > 6 THEN PRINT "YEA"
- b. 20 IF 5.2 = INT(5.2) OR 2 < 9 THEN PRINT "YES"
- c. 30 IF NOT (4 > 10) THEN PRINT "REALLY"

- a. No output is printed (control drops to the subsequent statement).
- b. YES
- c. REALLY

- 13 ■ What is the purpose of the DEF FN statement?

To enable programmers to define their own functions.

- 14 ■ Does a “dummy” variable in a DEF FN statement affect variables in the program?

No.

- 15 ■ May a function be an argument in itself?

Yes. The innermost function is computed first, and the value returned is used by the function to calculate a second value.

HANDS-ON PRACTICE

Find the square root of

$$5^4 + 3(4 - 1.6)$$

Use the SQR function and confirm the result by raising the expression to the $\frac{1}{2}$ power.

TRY YOUR HAND AT THESE

1. Write a program to read in a value X and determine whether it is positive or negative (some appropriate message should be printed) and whether it is a whole number or not. (Hint: Use the INT function.)
2. Write a program to print out a table of logarithms in an inputted base. The table should range between some inputted minimum and maximum values.
3. Write the single line to produce a random integer between 10 and 20.
4. Write a similar line to produce a random integer between 7 and 77.
5. Write a program that prints 100 random numbers, each of which is between 3.0 and 65.2.
6. Replace the following groups of IF statements with more succinct versions by using the logical operators.

a. 100 IF X = 1 THEN 1000
 110 IF Y = 45 THEN 1000
 120 IF Z <> 22.4 THEN 1000

b. 100 IF X = 1 THEN 1000
 1000 IF Y + 3 <> 25 THEN 2000

c. 100 IF B ↑ 2 - 4 * A * C < 0
 THEN 1000
 1000 IF Q = 50 THEN 2537

d. 100 IF B = 2 THEN 120
 110 IF F = 52 THEN 1000

7. Write a program to determine which of the integers between 1 and 4,098 are perfect squares. (A perfect square is a number whose square root is an integer.) (Hint: Determine whether the square root is an integer by using the method you devised in question 1.)

8. Write a program to generate 2,500 random numbers between 1 and 1,000, but print only those that are both even and multiples of 5.
9. A number is considered “gracious” if it is both odd and a multiple of 3. If a number is negative and divisible by 101, it is “audacious.” If a number conforms to both these criteria (that is, it is both gracious and audacious) it is “great.” Write a program that prints out whether an inputted number is gracious, audacious, or great.
10. Write a program that inputs a minimum, maximum, and increment value for a radius. The program should print a table of results showing the current radius, the circumference of a circle of that radius ($2\pi r$), the area of the circle (πr^2), and the volume of a sphere of that radius ($\frac{4}{3}\pi r^3$). The program should test that the minimum value is smaller than the maximum. If it is not, a switch should be made and an appropriate comment printed before the desired table is produced.

Introduction to Character String Manipulation

In the opinion of many people, the material dealt with in this chapter is one of the most interesting aspects of BASIC programming. Since the subject is so extensive we have decided to devote this chapter to the more elementary (basic?) operations, leaving the more complex features and techniques for later on. If you have had occasion to work on a word processor, you will appreciate the immense advantage that computers have over the standard typewriter, being able to modify the material instead of retyping it. In BASIC (and with the Commodore 64's dialect in particular) it is possible to see the rudiments of a text-handling system from which a word processor can be developed. In this chapter, we shall deal not so much with the numeric manipulation capabilities of the computer as with its ability to handle textual material as data. In particular, you will be introduced to the following concepts:

- storing literals in variables
- rules for naming string variables
- the LEN function
- the null string
- testing a string for equality
- the LEFT\$ function

- the RIGHT\$ function
- the MID\$ function (both versions)
- the LEFT\$, RIGHT\$, and MID\$ commands
- concatenating strings
- converting string data to numeric data and back (VAL, STR\$)
- setting the time (TIME\$)
- the TI function
- controlling the horizontal cursor position (TAB, SPC)
- determining the horizontal cursor position (POS)

You may indeed be one of the many people who feel strongly that working with textual data not only generates a greater sense of accomplishment and reward but also is an even greater source of fun than working solely with numbers. Since the development of the word processor, the whole question of the manipulation of textual data has assumed increasing importance, and in the years to come it is inevitable that this area will become even more highly developed.

You have already dealt on numerous occasions with strings. The simple instruction

```
100 PRINT "HELLO"
```

contains the character string (a string of characters) HELLO. Just as we may print a constant,

```
100 PRINT 5
```

or store it in a variable,

```
110 X = 5
```

so we can store a character string. However, its variable name must terminate with a dollar sign. In all other respects, character string variables obey the identical naming rules as do numeric variables.

In the following program, the salutation

```
"WE WISH YOU THE BEST OF LUCK!"
```

is stored in the character string MESSAGE\$. Once it is stored there, it may be accessed as often as desired. Since the variable MESSAGE\$ is contained in a PRINT statement within a FOR . . . NEXT loop that is executed five times, the message is printed on separate lines five times.

PROGRAM 6-1

```

100 PRINT "*"*****"
110 PRINT "*"
120 PRINT "*" ASSIGNING A STRING VARIABLE "*"
130 PRINT "*"
140 PRINT "*"*****"
150 PRINT
160 MESSAGE$ = "WE WISH YOU THE BEST OF LUCK!"
170 FOR I = 1 TO 5
180 PRINT MESSAGE$
190 NEXT I

```

RUN

```

*****
*
* ASSIGNING A STRING VARIABLE *
*
*****

```

```

WE WISH YOU THE BEST OF LUCK!
WE WISH YOU THE BEST OF LUCK!
WE WISH YOU THE BEST OF LUCK!
WE WISH YOU THE BEST OF LUCK!
WE WISH YOU THE BEST OF LUCK!

```

READY.

The simplest question that can be asked about a string is: How long is it? This vital statistic may be found by using the LEN function. For example, the statement

```
PRINT LEN("THIS IS A SHORT STRING")
```

displays the value 22, which is the number of characters in the string, including spaces, for the computer considers a space to be a

character just as any letter or symbol. A special string with a length of zero is called the *null string*. It is specified as two consecutive quotation signs with no space between them and serves the same role in string manipulation that zero plays in numerical computation. In the same way that a numeric variable can be cleared by setting it to zero, so can a string variable be set to “no string” by assigning the null string to it.

Testing Strings for Equality

String values, like numeric values, may be compared. It is perfectly valid, for example, to type

```
100 IF A$ = "YES" THEN GOTO 1200
```

which tests the value of the character string stored in A\$ against the literal “YES”. If they match, the comparison is true and control is sent to line 1200; if not, control passes on to the next statement in line. Unfortunately, because BASIC differentiates between upper- and lowercase letters, the character “a” is different from the character “A”, and therefore they would not match in a comparison. The programmer must constantly watch for this error because it is extremely easy to fall prey to it.

The following program flirtatiously asks the user for a date by using the INPUT command to enter the necessary information. The program assumes a rather personal, almost conversational character because of its use of character strings.

PROGRAM 6-2

```
100 PRINT "*****"
110 PRINT "*"
120 PRINT "      SALVATION FOR THE LOVELORN"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 INPUT "HI CUTIE! WHAT'S YOUR NAME";NAME$
```

```

170 PRINT "THAT'S A NICE NAME! ARE YOU A BOY
    OR A GIRL, ";NAME$
180 INPUT SEX$
190 IF SEX$ = "BOY" THEN PRINT "GOOD, I'M A
    GIRL. BUSY TONIGHT?":END
200 IF SEX$ = "GIRL" THEN PRINT "GOOD, I'M A
    BOY. BUSY TONIGHT?":END
210 PRINT "MY, YOU'RE STRANGE..."

```

RUN

```

*****
*
*      SALVATION FOR THE LOVELORN      *
*
*****

```

```

HI CUTIE! WHAT'S YOUR NAME? JENNIFER
THAT'S A NICE NAME! ARE YOU A BOY OR A GIRL,
JENNIFER? GIRL
GOOD! I'M A BOY,
WHAT ARE YOU DOING TONIGHT?

```

READY.

RUN

```

HI CUTIE! WHAT'S YOUR NAME? DAVID
THAT'S A NICE NAME! ARE YOU A BOY OR A GIRL,
DAVID? BOY
GOOD! I'M A GIRL,
WHAT ARE YOU DOING TONIGHT?

```

READY.

RUN

```

HI CUTIE! WHAT'S YOUR NAME? NEBISH
THAT'S A NICE NAME! ARE YOU A BOY OR A GIRL,
NEBISH? ISH
MY, YOU'RE STRANGE...

```

READY.

As soon as the heading is printed out, line 160 brashly asks for the user's name. In line 170, the program incorporates the name typed (stored in the variable NAME\$) into its message, giving it a conversational quality. If in answer to the question printed in line 170, the answer "BOY" is typed, the computer responds by saying that it is a girl and vice versa. For those who are not sure what they are, an appropriate message is generated.

String Slicing Operations

THE LEFT\$ FUNCTION

It is often necessary to examine sections of a string—and perhaps change them. You might, for example, want to examine the leftmost three characters of a string. This is possible by using the LEFT\$ function, which examines the leftmost characters of a string without changing its contents at all. The format of the function is

LEFT\$(*string, number of characters to slice*)

where *string* is the string itself within quotation marks or simply the string variable. For example,

```
LEFT$( "ISN'T THIS AMAZING?" , 2 )
```

generates a copy of the two leftmost characters, which make up the word

IS

In the following program, the literal "RULE BRITANNIA" is assigned to the string variable MESSAGE\$. Within a FOR . . . NEXT loop, successively larger "slices" of the string are printed out. The last and longest slice is the original string.

PROGRAM 6-3

```

100 PRINT "*"*****"
110 PRINT "*"
120 PRINT "*" ILLUSTRATION OF LEFT$ SLICING "*"
130 PRINT "*"
140 PRINT "*"*****"
150 PRINT
160 INPUT "ENTER YOUR MESSAGE ";MESSAGE$
170 FOR I = 1 TO LEN(MESSAGE$)
180 PRINT LEFT$(MESSAGE$,I)
190 NEXT I

```

RUN

```

*****
*
* ILLUSTRATION OF LEFT$ SLICING *
*
*****

```

ENTER YOUR MESSAGE: ? RULE BRITTANIA

```

R
RU
RUL
RULE
RULE B
RULE BR
RULE BRI
RULE BRIT
RULE BRITT
RULE BRITTA
RULE BRITTAN
RULE BRITTANI
RULE BRITTANIA

```

READY.

This program references the user-typed string stored in MESSAGE\$. Within the FOR . . . NEXT loop extending from line 170

to 190, the index *I* varies from 1 to the length of the string MESSAGE\$ (which is 14). Since the second parameter of the function LEFT\$ is *I*, the length of the slice examined begins with 1 and ends with 14.

The RIGHT\$ Function

The companion to the LEFT\$ function is the RIGHT\$. As you will have guessed from its name, its role is to generate a copy of the rightmost *n* characters, where *n* is the second of the parameters that have to be supplied by the programmer. For example, the instruction

```
PRINT RIGHT$( "COMMODORE EATS APPLES", 6)
```

references the six rightmost characters of the string, "APPLES". In the following program the literal "COMMODORE EATS APPLES" is stored in the string variable PHRASE\$. By means of a FOR . . . NEXT loop, the rightmost slice of the string is printed out, beginning with a slice of 1 character (the rightmost) and ending with the entire string, which has a length of 21.

PROGRAM 6-4

```
100 PRINT "*" * 21
110 PRINT "*"
120 PRINT "*" ILLUSTRATION OF RIGHT$ SLICING "*"
130 PRINT "*"
140 PRINT "*" * 21
150 PRINT
160 INPUT "ENTER YOUR MESSAGE "; MESSAGE$
170 FOR I = 1 TO LEN(MESSAGE$)
180 PRINT RIGHT$(MESSAGE$, I)
190 NEXT I
```

```
RUN
```

```
*****
*
*  ILLUSTRATION OF RIGHT$ FUNCTION  *
*
*****
```

```
ENTER YOUR MESSAGE ? COMMODORE EATS APPLES
S
ES
LES
PLES
PPLES
APPLES
  APPLES
S APPLES
TS APPLES
ATS APPLES
EATS APPLES
  EATS APPLES
E EATS APPLES
RE EATS APPLES
ORE EATS APPLES
DORE EATS APPLES
ODORE EATS APPLES
MODORE EATS APPLES
MMODORE EATS APPLES
OMMODORE EATS APPLES
COMMODORE EATS APPLES

READY.
```

The MID\$ Function

The last of the slicing functions available is the MID\$ function, which as you might imagine, generates a copy of the middle section of a string. In order to provide additional options, the MID\$ function is available in two forms. The first has the format

MID\$(string, starting position)

which references all the characters beginning at the specified starting position and extending to the end of the string. Although it may seem as if the MID\$ function behaves in a manner identical to that of the RIGHT\$ function, this is true only for special cases. The following example illustrates the difference between the MID\$ and RIGHT\$ functions:

```
A$="FOR WHOM THE BELL TOLLS"
PRINT MID$(A$,14), RIGHT$(A$,14)
BELL TOLLS          THE BELL TOLLS

READY.
```

In the case of MID\$, the characters starting with the fourteenth position and extending all the way to the end of the string are printed, whereas in the case of RIGHT\$, the rightmost 14 characters are printed. It is clear that these string slices are not the same. By contrast, however, the strings referenced by MID\$ and RIGHT\$ in the following special case are indeed the same, but only because the string happens to have an odd number of characters and the string slice begins at the midpoint:

```
PRINT MID$("HALLELUJAH!",7), RIGHT$(
"HALLELUJAH!",7)
LUJAH!          LUJAH!

READY.
```

In the following program (which uses the only version of MID\$ described so far) slices of the literal "HI THERE!" are printed within a FOR . . . NEXT loop. The slices get smaller and smaller because, as the index gets larger, the characters after the position specified by the index become fewer and fewer.

PROGRAM 6-5

```
100 PRINT "*****"
110 PRINT "*"
120 PRINT "* ILLUSTRATION OF MID$ VERSION 1 *"
130 PRINT "*"
140 PRINT "*****"
```

```

150 PRINT
160 INPUT "ENTER YOUR MESSAGE ";MESSAGE$
170 FOR I = 1 TO LEN(MESSAGE$)
180 PRINT MID$(MESSAGE$,I,2)
190 NEXT I

```

RUN

```

*****
*
* ILLUSTRATION OF MID$ VERSION 1 *
*
*****

```

```

ENTER YOUR MESSAGE: ? HI THERE!
HI THERE!
I THERE!
  THERE!
THERE!
HERE!
ERE!
RE!
E!
!

```

READY.

The second version of the MID\$ function uses a third parameter, which specifies the length of a substring of the original character string so that

```
MID$("FRATERNIZATION",2,3)
```

references the substring that begins with the second character and extends for a length of three characters. The substring referenced is therefore the word "RAT." In the following program, the user is asked to input a phrase. Once this has been done, the phrase is printed out in its entirety and then in adjacent pairs of characters.

PROGRAM 6-6

```

100 PRINT "*****"
110 PRINT "*"
120 PRINT "*   ILLUSTRATION OF MID$ VERSION 2   *"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 INPUT "ENTER YOUR MESSAGE ";MESSAGE$
170 FOR I = 1 TO LEN(MESSAGE$)
180 PRINT MID$(MESSAGE$,I,1)
190 NEXT I

```

RUN

```

*****
*
*   ILLUSTRATION OF MID$ VERSION 2   *
*
*****

```

ENTER YOUR MESSAGE ? COMPUTER

CO
OM
MP
PU
UT
TE
ER
R

READY.

Printing a String in Reverse

You will recall that adding a semicolon to the end of a PRINT statement inhibits the generation of a carriage return. The next program exploits this fact, and with the use of a FOR . . . NEXT loop in which the index counts backwards, an inputted phrase is printed out in reverse.

PROGRAM 6-7

```

100 PRINT "*****"
110 PRINT "*"
120 PRINT "*" PRINTING A STRING 'SDRAWKCB' "*"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 INPUT "PLEASE ENTER A PHRASE: ";PHRASE$
170 PRINT "THANK YOU. I WILL NOW PRINT IT OUT
    BACKWARDS:"
180 PRINT:PRINT
190 FOR I=LEN(PHRASE$) TO 1 STEP -1
200 PRINT MID$(PHRASE$,I,1);
210 NEXT I
220 PRINT
230 INPUT "DO YOU WANT ANOTHER ROUND (YES OR
    NO): ";AGAIN$
240 IF AGAIN$ = "YES" THEN 150

```

RUN

```

*****
*
* PRINTING A STRING 'SDRAWKCB' *
*
*****

```

```

PLEASE ENTER A PHRASE: ? MUST I?
THANK YOU. I WILL NOW PRINT IT OUT BACKWARDS:
?I TSUM

```

```

DO YOU WANT ANOTHER ROUND? (YES OR NO): ? YES

```

```

PLEASE ENTER A PHRASE: ? I LIKE THIS
THANK YOU. I WILL NOW PRINT IT OUT BACKWARDS:
SIHT EKIL I

```

```

DO YOU WANT ANOTHER ROUND? (YES OR NO): ? NO

```

```

READY.

```

After each phrase has been printed out in reverse the user is asked if another round is desired. Another character string variable (in this case, `AGAIN$`) is used to store the user's response of "YES" or "NO". This response is then tested within the program against the literal "YES". If they match, control is sent immediately to line 150, which allows the process to be repeated. Any response other than YES is equivalent to typing NO.

This program illustrates a little of the conversational mood it is possible to create between the computer and the user. After the phrase is typed in, the computer courteously thanks the user and tells him or her what it is about to do next. This so-called friendliness is rapidly becoming a standard in modern computer programming circles since it relieves the user of much unnecessary anxiety. You will frequently hear the phrase *user friendly* being applied to programs that explicitly ask for the required information and address the user in an informative, disarming way.

Returning now to the program, you will see that the actual mechanism by which the program prints out the string in reverse order is contained in lines 190 through 210. Successive slices of one-character strings are printed starting at position

```
LEN(PHASE$)
```

which is the end of the string, and extending to position 1, the beginning of the string.

Concatenation of Character Strings

It is also possible to attach two strings end-to-end. If for example,

```
A$ = "BREAK"
```

and

```
B$ = "FAST"
```

it is possible to store the single literal "BREAKFAST" into the string variable C\$ by a process known as *concatenation*, as we can see in the following example:

```
C$ = A$ + B$
```

Here the concatenation symbol is the plus sign, the same symbol that is used for addition in arithmetic. In a sense, the plus sign plays the role of "adding together" two literals.

The concatenation operator also allows the computer to store strings in reverse order (as opposed merely to printing in reverse order). This is done in the following program, where the phrase stored in MESSAGE\$ is reversed into REVERSE\$.

PROGRAM 6-8

```
100 PRINT "*****"
110 PRINT "*"
120 PRINT " REVERSING STRINGS VERSION 1"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 INPUT "ENTER YOUR MESSAGE PLEASE: ";MESSAGE$
170 REVERSE$ = ""
180 PRINT "YOUR MESSAGE IN REVERSE IS:"
190 FOR I = LEN(MESSAGE$) TO 1 STEP -1
200 REVERSE$ = REVERSE$ + MID$(MESSAGE$,I,1)
210 NEXT I
220 PRINT REVERSE$
```

RUN

```
*****
*
* REVERSING STRINGS VERSION 1
*
*****
```

```
ENTER YOUR MESSAGE PLEASE: ? ALL'S WELL THAT
ENDS WELL!
```

```
YOUR MESSAGE IN REVERSE IS:
!LLEW SDNE TAHT LLEW S'LLA

READY.
```

In this program the user is asked to enter a phrase, which is stored in MESSAGE\$. The variable REVERSE\$ is then set to the null string so that later on it can be “added” to. Within the FOR . . . NEXT loop, one character at a time (beginning with the last) is successively added to the end of REVERSE\$. At this point, both the original and the reversed strings are printed out. The sequence of events that occurs is illustrated in the following action play, based on the assumption that the phrase typed in is the word “BINGO”.

Action Taken	PHRASE\$	REVERSE\$
(none)	“BINGO”	“”
REVERSE\$ = REVERSE\$ + MID\$(MESSAGE\$,5,1)	“BINGO”	“O”
REVERSE\$ = REVERSE\$ + MID\$(MESSAGE\$,4,1)	“BINGO”	“OG”
REVERSE\$ = REVERSE\$ + MID\$(MESSAGE\$,3,1)	“BINGO”	“OGN”
REVERSE\$ = REVERSE\$ + MID\$(MESSAGE\$,2,1)	“BINGO”	“OGNI”
REVERSE\$ = REVERSE\$ + MID\$(MESSAGE\$,1,1)	“BINGO”	“OGNIB”

In computer programming in general, but in Commodore BASIC in particular, there are often several ways to solve the same problem. A result identical to that just obtained is produced by the next version of the program, where the index of the FOR . . . NEXT loop counts up from 1 to the length of the inputted string, each time adding a single-character “slice” to the front of REVERSE\$. This process is illustrated in the following action play, where the inputted phrase is “CAT”.

Action Taken	MESSAGE\$	REVERSE\$
(none)	"CAT"	""
REVERSE\$ = MID\$(MESSAGE\$,1,1) + REVERSE\$	"CAT"	"C"
REVERSE\$ = MID\$(MESSAGE\$,2,1) + REVERSE\$	"CAT"	"AC"
REVERSE\$ = MID\$(MESSAGE\$,3,1) + REVERSE\$	"CAT"	"TAC"

A close scrutiny of Program 6-9 will confirm that the action play just indicated does in fact reflect that used in the program.

PROGRAM 6-9

```

100 PRINT "*****"
110 PRINT "*"
120 PRINT "      REVERSING STRINGS VERSION 2      *"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 INPUT "ENTER YOUR MESSAGE: ";MESSAGE$
170 REVERSE$ = ""
180 PRINT "YOUR MESSAGE IN REVERSE IS:"
190 FOR I = 1 TO LEN(MESSAGE$)
200 REVERSE$ = MID$(MESSAGE$,I,1) + REVERSE$
210 NEXT I
220 PRINT REVERSE$
    
```

RUN

```

*****
*
*      REVERSING STRINGS VERSION 2      *
*
*****
    
```

```

ENTER YOUR MESSAGE: ? MARY HAD A LITTLE LAMB
YOUR MESSAGE IN REVERSE IS:
BMAL ELTTIL A DAH YRAM
    
```

READY.

The VAL and STR\$ Functions

Although numeric and string variables are useful, there are certain restrictions on each. For example, a string can never be operated upon arithmetically, so that

```
2 * "DOG"
```

is not equal to "2 DOGS". Even when the literal contains just a number, such as in the expression

```
4 * "127"
```

it cannot be evaluated since the first operand is numeric and the second is a character string. Sometimes, however, it becomes necessary to operate arithmetically on a string that contains a numeric quantity. In such cases, the string may be converted to its equivalent numeric value by means of a function called "VAL." The VAL function ignores all spaces and initiates a left-to-right scan for a numeric value within the string. If the first nonspace character is not a digit or a sign (+ or -) the value returned is zero. Here are some examples:

Command	Value Returned
PRINT VAL(" 123")	123
PRINT VAL(" -2")	-2
PRINT VAL("2.65")	2.65
PRINT VAL("1 + 9")	1
PRINT VAL("ABRACADABRA")	0
PRINT VAL("+5")	5
PRINT VAL("3") + VAL("4")	7
PRINT VAL("5E3")	5000
PRINT VAL("5.6E - 2")	.056
PRINT VAL("123ZERO")	123
PRINT VAL(1)	? TYPE MISMATCH ERROR (argument must be a string)

In the following program the VAL function is used to extract the numeric digits of a string and add them together.

PROGRAM 6-10

```

100 PRINT "*****"
110 PRINT "*"
120 PRINT "* ILLUSTRATION OF THE VAL FUNCTION *"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 INPUT "ENTER YOUR STRING: ";S$
170 SUM = 0
180 FOR I = 1 TO LEN(S$)
190 SUM = SUM + VAL(MID$(S$,I,1))
200 NEXT I
210 PRINT "THE SUM OF THE DIGITS IS: ";SUM

```

RUN

```

*****
*
* ILLUSTRATION OF THE VAL FUNCTION *
*
*****

```

```

ENTER YOUR STRING: ? 3 98 AF16C2-4
THE SUM OF THE DIGITS IS: 33

```

READY.

After printing the heading the user is asked to enter a literal, which is stored in the variable S\$. (Avoid using the more reasonable variable name "STRING\$" because it contains the reserved word "STR"). On the assumption that the string contains embedded numerics, the program scans each character of the string, going from left to right. Each time a digit is encountered, the VAL function converts it to its numeric equivalent, which is added to SUM (initialized to zero at the beginning of the program). If on the other hand, the character is not a digit, the value zero is returned by the

VAL function and the value of SUM is left unchanged. In this program, the inputted string is

```
3 98 AF16C2-4
```

The sum of 33 is obtained by adding the digits 3, 9, 8, 1, 6, 2, and 4. Since each substring is only one character long, the substrings "98" and "-4" are treated as the separate entities "9", "8", "-", and "4", which have values of 9, 8, 0, and 4, respectively.

The inverse of the VAL function is called "STR\$." The role played by STR\$ is to convert its numeric argument to a character string. Some examples are given in the following table.

Command	Value Returned
PRINT STR\$(5)	"5"
PRINT STR\$(3 + 4)	"7"
PRINT STR\$(1E4)	"10000"
PRINT STR\$(1.23)	"1.23"
PRINT STR\$(-4.2E-20)	"-4.2E-20"

In the following program the STR\$ function is used to determine the number of digits in its numeric argument. The variable name selected for the length of the string is called "LNGETH" rather than the more reasonable name "LENGTH" because, as you will recall, the latter contains the reserved word "LEN."

PROGRAM 6-11

```
100 PRINT "*****"
110 PRINT "*"
120 PRINT "* ILLUSTRATION OF THE STR$ FUNCTION *"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 INPUT "ENTER YOUR NUMBER PLEASE: ";NUMBER
170 LNGETH = LEN(STR$(NUMBER))-1
180 PRINT "YOUR NUMBER,";NUMBER;"CONTAINS";LNGETH;"
      "DIGITS"
```

```
RUN
```

```
*****
*
* ILLUSTRATION OF THE STR$ FUNCTION *
*
*****
```

```
ENTER YOUR NUMBER PLEASE: ? 5744
YOUR NUMBER, 5744 CONTAINS 4 DIGITS
```

```
READY.
```

In line 170, “- 1” is necessary because the STR\$ function provides an extra space for the sign of the number (positive or negative).

You might be interested in a totally different way to solve this problem. The number of digits contained by any number may be found by computing the logarithm to the base 10 (the common logarithm) of that number, adding 1, and taking the integer portion of the result.

The Built-in Variable TI\$

One of the most useful features of the Commodore 64 is its ability to keep track of the time once it has been typed into the computer. The manner in which the time may be set is the same as any ordinary string variable. The form of the literal between the quotation marks is

```
“hhmmss”
```

where hh represents the number of hours in military time (from 0 to 23), mm represents the minutes (from 0 to 59), and ss the seconds (from 0 to 59). To set the time to 10:41 A.M., for example, the command

```
TI$ = "1041"
```

is used. This command automatically sets the seconds to zero. In a similar manner, the time may be set to 5:24:30 P.M. by the command

```
TI$ = "172430"
```

where the number 1724 is the 24-hour form for 5:24 P.M. and 30 is the number of seconds.

Once TI\$ has been set it may be accessed any time, just as may any ordinary string variable. In the following program the time (in 24-hour format) is continuously displayed on the screen.

PROGRAM 6-12

```
100 PRINT "*****"
110 PRINT "*"
120 PRINT "      ILLUSTRATION OF TI$ FUNCTION"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 INPUT "ENTER THE TIME IN THE FORM HHMMSS ->";
    TI$
170 PRINT TI$
180 GOTO 170
```

In this program, the time is entered as a character string in the form "hhmmss". However, since any character whatever is accepted into a character string, this method of entering the time tends to be error-prone. For example, there is nothing to prevent a careless user from typing in something like 4:46pm which results in an ?ILLEGAL QUANTITY ERROR message.

To avoid this possibility, the following approach can be adopted. Instead of inputting a character string, the user types in the hours, minutes, and seconds individually—in response to individual prompts from the program. All the values may then be validated within the program so that any illegal values typed in may be rejected instantly. Furthermore, Program 6-12 suffers from the defect of displaying the time continuously down the lefthand margin of the screen—at its own frenetic electronic speed. In this way, more than one entry is displayed per second since the computer completes the

loop more than once per second. In the next, improved version of the program, not only is the time validated but also it is updated (uptimed?) on the screen once per second by comparing the current time with the last printed time (stored in the variable TEMP\$).

PROGRAM 6-13

```

100 PRINT "*"
110 PRINT "*"
120 PRINT "*" DISPLAYING THE TIME ONCE PER SEC "*"
130 PRINT "*"
140 PRINT "*"
150 PRINT
160 A$="0"
170 INPUT "ENTER THE HOUR IN MILITARY TIME: ";HRS
180 IF HRS < 0 OR HRS > 23 OR HRS <> INT(HRS)
    THEN PRINT "MUST BE 0-23":GOTO 170
190 H$ = MID$(STR$(HRS),2)
200 IF LEN(H$) < 2 THEN H$ = A$ + H$
210 INPUT "ENTER THE MINUTE (0-59): ";MIN
220 IF MIN < 0 OR MIN > 59 OR MIN <> INT(MIN)
    THEN PRINT "ONLY 0-59":GOTO 210
230 M$ = MID$(STR$(MIN),2)
240 IF LEN(M$) < 2 THEN M$ = A$ + M$
250 INPUT "ENTER THE SECONDS (0-59): ";SEC
260 IF SEC < 0 OR SEC > 59 OR SEC <> INT(SEC)
    THEN PRINT "ONLY 0-59":GOTO 250
270 S$ = MID$(STR$(SEC),2)
280 IF LEN(S$) < 2 THEN S$ = A$ + S$
290 TIME$ = H$ + M$ + S$
300 TEMP$ = ""
310 IF TEMP$ <> TIME$ THEN TEMP$ = TIME$:PRINT
    TEMP$
320 GOTO 310

```

In lines 160 to 210 of this program the user's typed responses for the hours, minutes, and seconds are entered and validated. If any one of them is found to be invalid, it is immediately rejected and the user is given the opportunity to retype that value. In line 220, the three numeric variables HRS, MIN, and SEC are converted to their equivalent strings and concatenated to form the time. Once in this readable form the resulting six-character string is stored in TIME\$,

simply another variable whose first two letters are TI and is thus equivalent to TI\$ (TIGER\$ would work just as well but would be less mnemonic). The infinite loop in lines 310 through 320 has the effect of printing out the time only once per second. As already said, this operation is accomplished by virtue of the fact that the time is displayed only when it is different from the previously printed time stored in TEMP\$.

The TI Numeric Function

From the moment the Commodore 64 is switched on, the number of sixtieths of a second it has been on is constantly updated and stored in the numeric variable TI. For many applications it is useful to be able to measure the time between two events in a program. The TI function can be used directly in calculations involving intervals of time. It accesses the same timer as does the TI\$ function, even though it returns the time in a different form. One situation in which it is particularly useful is timing a certain segment of a program.

For example, suppose you wish to determine the exact amount of time that a segment of code (such as the following segment, which contains a double nest of loops) takes to execute

```

140 S = 0
150 FOR I = 1 TO 100
160 FOR J = 1 TO 10
170 X = I ↑ 2 * J ↑ 3
180 S = S + X
190 NEXT J
200 NEXT I

```

To time this segment of code you can simply insert the following two statements:

```

135 T = TI
205 PRINT (TI - T) / 60;"SECONDS WERE
    USED"

```

Just insert a statement storing the time into a variable before the segment, and a statement printing the time minus the stored time

after the event. The number printed is the length of time taken to execute the segment of code (in sixtieths of a second). In order to return the number in seconds, simply divide by 60. Later on you shall use this function to great advantage.

The TAB Function

Now that you have covered some aspects of string manipulation, the time has arrived to learn how to print strings in an elegant fashion. For this purpose there is the TAB function, which behaves in a manner very similar to the TAB key on an ordinary typewriter. PRINT TAB(*n*) tabs over to position number *n* of the screen. The contents of the parentheses may be a specific number or an expression that is evaluated before it is acted on. The expression must be an integer, as it is impossible to TAB over a fraction of a space. If the expression turns out not to be an integer, BASIC automatically rounds it off to the nearest whole number. If the value of the rounded integer is not within the range zero through 255, an error results. If the current print position is already beyond space *n*, the TAB function applies to the position *n* on the subsequent line. TAB(0) is considered the leftmost position. In the same way that a PRINT statement may terminate with a semicolon (which affects the printing of the following items) so the appearance of a TAB specification at the end of a PRINT statement positions the subsequent print at position *n* of the current line. In the following program, the TAB function is used to print a literal diagonally down the screen.

PROGRAM 6-14

```

100 PRINT "*"
110 PRINT "*"
120 PRINT "* ILLUSTRATION OF THE TAB FUNCTION *"
130 PRINT "*"
140 PRINT "*"
150 PRINT
160 FOR I=10 TO 0 STEP -1
170 PRINT TAB(I);"HOW NOW BROWN COW?"
180 NEXT I

```


RUN

```
*****
*
* ILLUSTRATION OF THE TAB FUNCTION *
*
*****
```

```
        HOW NOW BROWN COW?
        HOW NOW BROWN COW?
        HOW NOW BROWN COW?
        HOW NOW BROWN COW?
        HOW NOW BROWN COW?
        HOW NOW BROWN COW?
        HOW NOW BROWN COW?
        HOW NOW BROWN COW?
        HOW NOW BROWN COW?
        HOW NOW BROWN COW?
        HOW NOW BROWN COW?
```

READY.

In the next example, both the TAB and MID\$ functions are illustrated. After the user has typed in a phrase, it is first printed out in regular fashion. Then successive pairs of characters are printed out diagonally, the index value of the loop being used as the argument of the TAB function to position the printing of each line.

PROGRAM 6-15

```
100 PRINT "*****"
110 PRINT "*"
120 PRINT "* ILLUSTRATION OF MID$ AND TAB"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 INPUT "ENTER A PHRASE PLEASE: ";PHRASE$
170 PRINT "YOUR PHRASE IS: ";PHRASE$
180 PRINT
190 FOR I = 1 TO LEN(PHRASE$)
200 PRINT TAB(I + 16);MID$(PHRASE$,I,2)
210 NEXT I
```

```
RUN
```

```
*****
*
*   ILLUSTRATION OF MID$ AND TAB   *
*
*
*****
```

```
ENTER A PHRASE PLEASE: ? CABBAGE PATCH
YOUR PHRASE IS: CABBAGE PATCH
```

```

CA
 AB
  BA
   AG
    GE
     E
      P
       PA
        AT
         TC
          CH
           H
```

```
READY.
```

You will notice that the final line of output consists of the single letter “H”. Unlike the “E” found six lines above it, the character immediately to its right is not a space—even though the difference between the two is not immediately visible. There is, however, no character whatever to the right of the letter “H” because it is the last character of the string. Nevertheless, the MID\$ function attempts to reference a character to its right and, finding none, treats it as a null string.

The SPC Function

The TAB instruction just described positions the cursor at a given location relative to the lefthand margin of the screen. The argument of the TAB function, therefore, refers to an absolute position on the current line. Rather than specify the position in this way, it is often

more convenient to specify a position relative to the current position. This is precisely the role played by the SPC (space) function, which in a PRINT statement, causes displayed information to skip a specified number of spaces to the right. For example, the instruction

```
PRINT "HELLO"; SPC(5); "GOODBYE"
```

displays

	<u>H</u>	<u>E</u>	<u>L</u>	<u>L</u>	<u>O</u>					<u>G</u>	<u>O</u>	<u>O</u>	<u>D</u>	<u>B</u>	<u>Y</u>	<u>E</u>	
column numbers	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

with five blank spaces between the two literals.

The specified number of spaces must be within the range zero through 255 inclusive. If the number is not an integer, it is rounded to the nearest one.

The POS Function

It is occasionally useful to determine the position at which the cursor is located in addition to specifying what it should be set to. This is done by means of the POS function, which returns a number, between 0 and 79, specifying the POSition at which the next PRINT statement will begin. POS takes only a dummy argument; that is, the argument must be there but its value does not matter. A typical command for the POS function is

```
1000 X = POS(0)
```

You have by now been exposed to some of the more important building blocks with which all text-handling systems (such as word processors) are constructed. You will now appreciate that a computer can cope with textual data almost as easily as with numeric data. To fully exploit the power of the computer, however, you must learn how to handle mass data, which in BASIC is done by the use of what are called arrays, the subject of the next chapter.

Review Questions

- 1 ■ What is the last character of every string variable name?

A dollar sign (\$).

- 2 ■ What is the length of the string

"WELCOME ALL YOU MERRY PEOPLE"

It is 28 characters long, including spaces.

- 3 ■ What is the name of the function that returns the length of a string?

The LEN function.

- 4 ■ What is the shortest string possible?

The null string, which has a length of zero.

- 5 ■ What is the value of

- `LEFT$("COMPUTER", 2)`
- `LEFT$("CURI0US", 3)`
- `LEFT$("RHYME", 0)`
- `LEFT$("STRANGE", 20)`
- `LEFT$("MYSTERIOUS", LEN("MYSTERIOUS") - 3)`

- CO
- CUR
- (the null string)
- STRANGE
- MYSTERI (The length of the string is 10. Subtracting 3 yields 7. `LEFT$("MYSTERIOUS", 7)` yields the answer.)

- 6 ■ What does the RIGHT\$ function do?

It slices off a copy of the rightmost characters as specified by the argument.

7 ■ What values are returned by the following?

- a. `RIGHT$("INTRIGUING", 4)`
- b. `RIGHT$("SHIMMER", 3)`
- c. `RIGHT$("TECHNIQUE", 5)`
- d. `RIGHT$("HUMOROUS", LEN("MAN") - 2)`
- e. `RIGHT$(LEFT$("MISSISSIPPI", 7), 3)`

- a. UING
- b. MER
- c. NIQUE
- d. S
- e. ISS

8 ■ What is returned by

- a. `MID$("I CAME, I SAW, I CONQUERED", 9)`
- b. `MID$("I CAME, I SAW, I CONQUERED", 16)`
- c. `MID$("PEDANTIC", 4)`
- d. `MID$("FRIENDLY", 4, 3)`
- e. `MID$("CONFUSING", LEN("CHAPTERS") / 2)`
- f. `MID$(LEFT$("UNCOMMON", 5) + "P" + RIGHT$("DISLEXIC", 5), 3, 7)`

- a. I SAW, I CONQUERED
- b. I CONQUERED
- c. ANTIC
- d. END
- e. FUSING
- f. COMPLEX

9 ■ What is the concatenation symbol?

The plus sign (+).

10 ■ What is printed by the statement:

```
PRINT "BLACK" + "JACK"
```

BLACKJACK

- 11 ■ What role is played by the VAL function?

It strips its string argument of any included spaces and returns its numeric value. If a letter is encountered in the string, the embedded number is considered to terminate at that point.

- 12 ■ What does the STR\$ function accomplish?

It converts its numeric argument to a character string.

- 13 ■ What command is used to set the time in Commodore BASIC?

TIME\$ = "hhmmss"

where the time is expressed in military (24-hour) format and hh represents the hours, mm the minutes, and ss the seconds.

- 14 ■ What statement is always associated with the TAB and SPC functions?

The PRINT statement.

HANDS ON PRACTICE

1. Type in the following program and try to determine what it will do before running it.

```

100 A$ = ""
110 FOR I = 1 TO 10
120   A$ = A$ + MID$("LONGSTRING",I)
130 NEXT I
140 PRINT A$

```

TRY YOUR HAND AT THESE

1. Write a program that requests the full name of the user and displays the length of that name in characters (including spaces and any punctuation).
2. Write a program that requests the user's full name and prints the first name, the middle initial, and the last name.

3. What does the following statement do:

```
IF LEN("THIS IS A LONG STRING ") = 22  
THEN PRINT "YES"
```

4. Write a program that requests a message and a character to search for. The program should print the number of times the character was found in the message.
5. Write a program that allows for the inputting of a sweepstake entry's name and address. The program should use this information to generate a form letter such as this:

Name of person: Ms. Diana Windsor
Address: 140 NW Wash D.C 10234

January 1, 1999
Ms. Diana Windsor
140 NW Wash D.C. 10234

Dear Ms. Windsor:

We are pleased to inform you that you have a chance to win a grand prize in our sweepstakes—\$10,000,000 paid to you at the rate of \$1000.00 a week. You must enter within a week to be eligible for this prize, so please hurry.

Sincerely yours,
James Hamill

Arrays

Computers are renowned for their remarkable ability to handle large masses of data swiftly, efficiently, and accurately. In order to do all this, BASIC provides the programmer with what are known as *arrays*—entities that permit the manipulation of long lists of numbers. The concepts described in this chapter are extremely important for anyone who wants to do serious, useful programming, and it is quite likely that this part of the text will have to be read several times before the material is completely absorbed. Among the new concepts you will be introduced to are the following:

- subscripts
- an array as an ordered list of numbers
- the DIMension statement
- filling an array with data
- generating a random array
- finding the maximum and minimum of an array, together with their positions
- calculation of Pearson's correlation coefficient
- the STOP instruction
- multidimensional arrays
- the CLR statement
- finding out the amount of free memory (FRE)

In all the programs you have examined so far, there was no difficulty in assigning suitable names to the different variables because

they were so few in number. The situation would be considerably different, however, if you had to assign a different variable name to each of, say, 100 students in a class—perhaps to compute their average class weight. Assigning 100 different names is not only extremely tedious and burdensome but even worse, is highly error prone. The problem is further compounded when the numbers involved are even larger than 100. With, say, 1,000 people in a school, the problem ceases to be at all manageable if tackled in this manner. A much more elegant way to approach the problem is to assign one name for all the values. However, at first glance it would seem that if this were done, it would be impossible to distinguish between one item and another. There is some truth to this answer, but you can borrow a technique used frequently in mathematics—that is, to specify the name of the item together with its position in the list of numbers. For example, the first item has position 1; the second, position 2; and so on. The hundredth item would therefore have the position 100. In mathematics, the position number (called the index) is written as a subscript just below the line in a smaller typeface than the name associated with it. For example, we could write

$$S = a_1 + a_2 + a_3$$

where the small numbers are the subscripts. Since on most computers it is impossible to indicate a subscript in this manner, the convention is to enclose it in parentheses. Let us now assume that all the 100 items referred to are assigned the common variable name X. The first value of X may then be denoted as X(1); the second may be written as X(2); and so on.

When the same variable name is used to describe a list of different numbers or values, the list is called an *array*. An array carries the notion that the items are listed in a definite order. This concept is not exactly new. Students are often ranked according to how high their average is. The top student is regarded as “number 1” (he or she has a subscript of 1). The student who comes in second correspondingly has a subscript of 2. In Commodore BASIC, an array of more than ten elements (actually 11, as you shall soon see) must be specifically “dimensioned” by means of a DIM statement. That is,

the programmer must state how much room is to be reserved for the elements of the array so that the computer can reserve memory space for them. For example, the statement

```
100 DIM X(500)
```

sets aside 501 locations for the subscript variable X. The reason there are 501 locations as opposed to 500 is because allowance is made for X(0), which is useful in some mathematical and scientific applications. The statement

```
200 DIM Y(600), Z(750)
```

sets aside 601 locations for the array Y and 751 locations for the array Z. If a variable (such as X) is subscripted before a corresponding DIM statement is encountered, Commodore BASIC automatically sets aside 11 locations for the array, X(0) through X(10). Attempting to access an element beyond its dimensioned value causes a

```
?BAD SUBSCRIPT ERROR
```

to be displayed and the program is terminated.

Once a variable has been dimensioned, it cannot be redimensioned within the body of the program (unless it is first cleared, a feature we shall discuss shortly). If a variable such as X has been subscripted without a corresponding dimension statement, it is considered as though it were dimensioned with

```
DIM X(10)
```

and so it may not be redimensioned explicitly. If this is attempted, the message

```
?REDIM'D ARRAY ERROR
```

appears and the program is terminated.

In the following program, an array of ten numbers is typed in by the user. These numbers are summed, and their average is found. Then the difference between the average and each of the elements in the array is computed and printed out. Notice that once the elements have been stored in the array, they may be accessed as often as is necessary, making the problem considerably more manageable.

PROGRAM 7-1

```

100 PRINT "*****"
110 PRINT "*"
120 PRINT "          INPUTTING AN ARRAY"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 DIM X(10)
170 PRINT "TYPE IN 10 VALUES, ONE AT A TIME"
180 FOR I = 1 TO 10
190 INPUT X(I)
200 NEXT I
210 REM
220 REM: NOW COMPUTE THE AVERAGE
230 REM
240 SUM = 0
250 FOR I = 1 TO 10
260 SUM = SUM + X(I)
270 NEXT I
280 AVERAGE = SUM / 10
290 REM
300 REM: PRINT THE DIFFERENCE BETWEEN AV AND
    EACH ELEMENT
310 REM
320 PRINT "THE AVERAGE IS:";AVERAGE
330 PRINT:PRINT "AND HERE ARE THE DIFFERENCES
    FROM THE AVERAGE:"
340 FOR I = 1 TO 10
350 PRINT AVERAGE - X(I);
360 NEXT I

```

```
RUN
```

```
*****
*
*           INPUTTING AN ARRAY           *
*
*
*****
```

```
TYPE IN 10 VALUES, ONE AT A TIME
```

```
? 23
? 4
? 14
? 23
? 21
? 13
? 3
? 2
? 25
? 19
```

```
THE AVERAGE IS: 14.7
```

```
AND HERE ARE THE DIFFERENCES FROM THE AVERAGE:
```

```
-8.3 10.7 .699999999 -8.3 -6.3 1.7 11.7
12.7 -10.3 -4.3
```

```
READY.
```

An interesting point to note is the slight error that is introduced when the third value (14) is subtracted from the average (14.7). You would expect $14.7 - 14$ to yield the result of 0.7. However, the computer actually displays the result as .699999999, a value slightly less than that expected. This is due to the fact that the Commodore 64 performs all its computations in binary (base 2). Just as there are certain fractions that cannot be expressed as finite decimals (for example, the value $\frac{1}{3}$), so are there numbers that cannot be exactly expressed in binary. The small resulting errors are called *round-off errors*.

Although this program performs its task flawlessly, it can, nevertheless, be criticized for the fact that it works for one case and one case only—when the number of elements in the array is equal to ten. If the number is anything but ten, the program as shown is

quite useless. The matter is easily rectified, however, by deciding on some maximum value for the dimensioned array, say 100. Then the specified number of elements can be typed in and tested to be sure that it is a positive integer not exceeding the value 100. Every time the constant 10 appears in the last program it may be replaced by the variable N (representing the number of elements in the array), thereby providing a much more general program.

This is precisely what is done in the next version of the program, where the user is asked to type in the desired value for N. This value is first tested to be sure that it is a valid number. If it is, the user is then asked to type in each element of the array, one at a time, each time the question mark prompt appears. Once the N numbers have been inputted, the array is printed out in packed format, followed by the difference of each element from the average. However, if the value of N is invalid, after a mild rebuke the user is asked to type in another value. No matter how often an invalid N is typed in, the computer merely provides an opportunity to correct the inputted value without ever losing its temper. If only some of our teachers had this kind of unlimited patience!

PROGRAM 7-2

```

100 PRINT "*"*****"
110 PRINT "*"
120 PRINT "* INPUTTING A VARIABLE LENGTH ARRAY *"
130 PRINT "*"
140 PRINT "*"*****"
150 PRINT
160 INPUT "HOW MANY ELEMENTS";N
170 IF N <= 0 OR N > 100 OR N <> INT(N) THEN
    PRINT "TRY AGAIN":GOTO 160
180 DIM X(N)
190 PRINT"TYPE IN";N;"NUMBERS ONE AT A TIME"
200 FOR I = 1 TO N
210 INPUT X(I)
220 NEXT I
230 REM
240 REM: CALCULATE AVERAGE

```

```

250 REM
260 SUM = 0
270 FOR I = 1 TO N
280 SUM = SUM + X(I)
290 NEXT I
300 AVERAGE = SUM / N
310 REM
320 REM: CALCULATE DIFFERENCES FROM AVERAGE
330 REM
340 PRINT "THE AVERAGE IS:";AVERAGE
350 PRINT "AND HERE ARE THE DIFFERENCES FROM
    THE AVERAGE:"
360 FOR I = 1 TO N
370 PRINT AVERAGE - X(I);
380 NEXT I

```

RUN

```

*****
*                                     *
* INPUTTING A VARIABLE LENGTH ARRAY *
*                                     *
*****

```

```

HOW MANY ELEMENTS ? 0
TRY AGAIN
HOW MANY ELEMENTS ? -7
TRY AGAIN
HOW MANY ELEMENTS ? 3
TYPE IN 3 NUMBERS ONE AT A TIME
? 10
? 5
? 27
THE AVERAGE IS: 14
AND HERE ARE THE DIFFERENCES FROM THE AVERAGE:
 4  9 -13

```

READY.

Dynamically Allocated Arrays

To make a program more flexible, the size of the dimension for an array may be left to the user to decide at the time of execution (as in Program 7-2, above). A previously defined variable name, such as N, may be substituted for the dimensioned value. Whatever valid number is typed into N is regarded as the dimension of the array. The ability to dynamically allocate room for arrays is made possible by a statement of the following type:

```
DIM X(N)
```

This capability is not common to all versions of BASIC, although increasingly, recent releases of the more sophisticated versions of the language have this useful feature. The maximum allowable value of N is 32,767, but the actual limitation is a somewhat lower number since the amount of memory available to the programmer is limited.

Finding the Maximum and Minimum of an Array

There are several methods used for finding the lowest and highest numbers of an array. The method described here may be regarded as the “natural” method, since it is probably the way the human mind goes about it. To find the minimum of a list of numbers, an assumption is made that the first element is the smallest. This temporary minimum is then compared successively with each element of the array until the last one has been compared. Whenever an element is found which is smaller than the temporary minimum, that element replaces the value stored as the temporary minimum. This strategy may be seen in the following illustration, where an array, X, is composed of five elements:

X	(1)	(2)	(3)	(4)	(5)
Value	4	6	8	2	1

Assigning to MIN the first element as the temporary minimum value (4), these steps follow:

```

X(2) < MIN ? (no)
X(3) < MIN ? (no)
X(4) < MIN ? (yes) therefore MIN = 2
X(5) < MIN ? (yes) therefore MIN = 1

```

The maximum of the array is found by a similar method in which the first element of the array is regarded as the temporary maximum. It is successively compared with each element of the array, replacing the contents of the temporary maximum each time an element is found that is larger than the current maximum. In the following program, the minimum, the maximum, and their locations within the array are calculated and printed. For the purpose of this program we shall arbitrarily limit the length of the array to 1,000 elements. Subroutines are used to break up the problem into its various natural modules.

The first subroutine, beginning in line 1000, simply generates N random integers between 1 and 1,000. Once the array has been generated and stored, the RETURN statement is encountered and control is sent back to the statement following line 190. Here, another GOSUB statement is encountered and control is sent to the subroutine beginning in line 2000, which finds the minimum, the maximum, and their locations. The strategy used in the subroutine is to assume that the first element of the array is both the minimum and the maximum. This being the case, the locations of the minimum and the maximum are therefore both 1. The FOR . . . NEXT loop begins with 2 rather than 1 because the first element has already been stored in MIN and MAX; they are merely being tested against the rest of the elements, which have the subscripts 2 . . . N.

Each time a new minimum or maximum is found, the corresponding value of the index is stored in SLOC (for small location) or BLOC (for big location) so that they always reflect the position of the current minimum and maximum respectively. When the loop is satisfied, the contents of MIN and MAX are the true minimum and maximum, and SLOC and BLOC will ultimately contain the loca-

tions at which they were found. (It would be clearer to use the names MINLOC for SLOC and MAXLOC for BLOC, but the computer would not distinguish them from MIN and MAX, since the first two letters are identical.)

The difference between the minimum and the maximum is called the *range* of the array. This is computed in a separate subroutine beginning in line 4000.

PROGRAM 7-3

```

100 PRINT "*"
110 PRINT "*"
120 PRINT "*" FINDING MIN, MAX AND LOCATIONS "*"
130 PRINT "*"
140 PRINT "*"
150 PRINT:PRINT
160 INPUT "ENTER THE ARRAY SIZE: ";N
170 IF N <= 0 OR N > 1000 OR N <> INT(N) THEN
    PRINT "MUST BE 0<N<=1000":GOTO 160
180 DIM ARRAY(N)
190 GOSUB 1000
200 GOSUB 2000
210 GOSUB 3000
220 GOSUB 4000
230 END
1000 REM
1010 REM:SUBROUTINE TO GENERATE THE ARRAY
1020 REM
1030 FOR I = 1 TO N
1040 ARRAY(I) = INT(RND(1) * 1000) + 1
1050 NEXT I
1060 RETURN
2000 REM
2010 REM:SUBROUTINE TO PRINT OUT THE ARRAY IN
    PACKED FORMAT
2020 REM
2030 FOR I = 1 TO N
2040 PRINT ARRAY(I);
2050 NEXT I
2060 PRINT:PRINT

```

```

2070 RETURN
3000 REM
3010 REM: SUBROUTINE TO FIND THE MIN-MAX AND
      THEIR LOCATIONS
3020 REM
3030 MIN = ARRAY(1):MAX = ARRAY(1):SLOC = 1:
      BLOC = 1
3040 FOR I = 2 TO N
3050 IF ARRAY(I) < MIN THEN MIN = ARRAY(I):
      SLOC = I
3060 IF ARRAY(I) > MAX THEN MAX = ARRAY(I):
      BLOC = I
3070 NEXT I
3080 PRINT "THE MINIMUM OF THE ARRAY IS";MIN;
      "IN LOCATION";SLOC:PRINT
3090 PRINT "THE MAXIMUM OF THE ARRAY IS";MAX;
      "IN LOCATION";BLOC:PRINT
3100 RETURN
4000 REM
4010 REM: PRINT THE RANGE
4020 REM
4030 PRINT "THE RANGE IS:";MAX-MIN
4040 RETURN

```

RUN

```

*****
*
* FINDING MIN, MAX AND LOCATIONS *
*
*****

```

```

ENTER THE NUMBER OF ELEMENTS IN THE ARRAY: ? 0
MUST BE 0<N<=1000
ENTER THE NUMBER OF ELEMENTS IN THE ARRAY: ? -5
MUST BE 0<N<=1000
ENTER THE NUMBER OF ELEMENTS IN THE ARRAY: ? 5
5 942 241 345 661

```

```

THE MINIMUM OF THE ARRAY IS 5 IN LOCATION 1
THE MAXIMUM OF THE ARRAY IS 942 IN LOCATION 2
THE RANGE IS: 937

```

READY.

Pearson's Correlation Coefficient

In statistics, it is often important to calculate the correlation between two sets of variables. It is well known that there is a high positive correlation between height and weight; that is, the taller a person is, the more that person probably weighs. Students are told that the more they study, the better their grades will be. If this advice is true, it would indicate a positive correlation. Examples of negative correlations abound. The more money you spend, the less you have left. Dentists tell us that the more candy we eat, the fewer natural teeth we will have left in adult life.

The Pearson correlation coefficient formula is used to calculate the degree of positive or negative correlation between two sets of data. What is interesting about it is that regardless of the data used, the formula always produces a result that lies between -1 and 1 inclusive. The only exception to this general rule is if one of the two variables under study is constant. Should this be the case, the program would find itself trying to take the square root of a negative quantity. Since this would lead to an error message in BASIC, the program tests for this contingency, and if found, the program is halted immediately with an explanatory message.

It should be noted that the correlation coefficient can be calculated without resorting to arrays. However, the reason for doing so here is that once the correlation coefficient has been calculated, the elements of both arrays are left intact and further statistical analyses may then be performed on them. Indeed, this is one of the major reasons for using arrays.

Suppose then, we have the two arrays X and Y,

X	Y
4	2
5	7
6	4
10	8
20	16

where an X value of 4 corresponds to a Y value of 2, and an X of 5 with a Y of 7, and so on. Without needing to know the mathematical

rationale, we note that in order to calculate the correlation coefficient, three more columns must be constructed: one for $X \times Y$ (denoted in mathematics as XY), another for X^2 , and a third for Y^2 .

X	Y	XY	X ²	Y ²
4	2	8	16	4
5	7	35	25	49
6	4	24	36	16
10	8	80	100	64
20	16	320	400	256

Now each of these five columns is summed:

45 37 467 577 389

The five sums are now substituted into the somewhat forbidding-looking formula

$$r = \frac{n\sum xy - \sum x \sum y}{\sqrt{(n\sum x^2 - (\sum x)^2)(n\sum y^2 - (\sum y)^2)}}$$

from which the correlation coefficient, r , is calculated. Do not be intimidated by the recurring use of the Greek sigma, written Σ . It simply stands for "the sum of."

Notice the immense amount of work performed by the FOR . . . NEXT loops, especially if N is very large. In fact, the FOR . . . NEXT loop was designed expressly for this kind of operation.

PROGRAM 7-4

```

100 PRINT "*"*****"
110 PRINT "*"
120 PRINT "*" PEARSON'S CORRELATION COEFFICIENT "*"
130 PRINT "*"
140 PRINT "*"*****"
150 PRINT:PRINT
160 INPUT "WHAT IS YOUR VALUE OF N:";N
170 IF N < 0 OR N <> INT(N) THEN PRINT "SORRY,
    TRY AGAIN":GOTO 160

```

```

180 DIM X(N),Y(N)
190 FOR I = 1 TO N
200 INPUT "X,Y";X(I),Y(I)
210 NEXT I
220 PRINT
230 FOR I = 1 TO N
240 XSUM = XSUM + X(I)
250 YSUM = YSUM + Y(I)
260 XYSUM = XYSUM + X(I) * Y(I)
270 XXSUM = XXSUM + X(I) * X(I)
280 YYSUM = YYSUM + Y(I) * Y(I)
290 NEXT I
300 NUMER = N * XYSUM - XSUM * YSUM
310 TEMP = (N * XXSUM - XSUM ↑ 2) * (N * YYSUM
- YSUM ↑ 2)
320 IF TEMP <= 0 THEN PRINT "DATA INVALID":STOP
330 DENOM = SQR(TEMP)
340 R = NUMER / DENOM
350 PRINT "CORRELATION COEFFICIENT =" ; R

```

RUN

```

*****
*
* PEARSON'S CORRELATION COEFFICIENT *
*
*****

```

```

WHAT IS YOUR VALUE OF N? .7
SORRY, TRY AGAIN
WHAT IS YOUR VALUE OF N? -123
SORRY, TRY AGAIN
WHAT IS YOUR VALUE OF N? 5
X,Y? 4,2
X,Y? 5,7
X,Y? 6,4
X,Y? 10,8
X,Y? 20,16

```

CORRELATION COEFFICIENT = .951950432

READY.

After the user has typed in the value of N , representing the number of elements in the X and Y arrays, N is tested to be sure it is both positive and an integer. If either of these conditions fails, a message is displayed telling the user to try again. If N is valid, control is sent to the `DIM` statement, which sets aside enough space for the two arrays X and Y . Within the `FOR . . . NEXT` loop, extending from line 190 to line 210, each element of X together with the corresponding element of Y is entered. As soon as N pairs of values have been typed in, the loop is satisfied, a blank line is printed, and the major `FOR . . . NEXT` loop, beginning in line 230 and ending in 290, computes the five sums used in the computation of the correlation coefficient. Once this loop is satisfied, the value of `NUMER` (standing for “numerator”) is calculated. The expression under the square-root sign is then computed and placed in the variable `TEMP`.

Provided that the value of `TEMP` is greater than zero, the data are considered valid. The square root of `TEMP` is then taken and stored in `DENOM`, and the value of `R` is computed. If it turns out that `TEMP` is less than or equal to zero, the message “`DATA INVALID`” is displayed, and execution of the program is stopped by means of the `STOP` instruction. The `STOP` differs from the `END` statement in that it displays the line number at which the termination occurred. It has much the same effect as does the `RUN/STOP` key and is treated in the same way. Since it produces what is, in effect, an error message, it is considered inelegant and is not used except where the user’s attention is being directed to the line in which the run was terminated.

Multidimensional Arrays

The arrays discussed so far were, without exception, one-dimensional. That is, they consisted of a list of numbers in a specific order. However, it is often convenient to arrange data in terms of rows and columns, in much the same way that a railway schedule is laid out. Such an arrangement of data is called a *matrix*. If you have difficulty picturing a matrix, think of it as an array, each of whose elements is

also an array. In Commodore BASIC, a matrix (two-dimensional array) is dimensioned in much the same way as is a one-dimensional array. The only difference is that for each dimension, a maximum subscript is supplied. For example, in order to reserve room for the two-dimensional array X consisting of three rows and four columns, the dimension statement

```
DIM X(3,4)
```

is used. As with one-dimensional arrays, if the dimension statement is omitted, it is as if the maximum value of each subscript is 10. In other words, typing

```
PRINT W(4,4)
```

without first dimensioning the matrix W, automatically performs the implied statement

```
DIM W(10,10)
```

Commodore BASIC does not limit the programmer to two-dimensional arrays. As a matter of fact, there may be up to 255 dimensions, although this maximum seems like a practical impossibility in light of the restrictions imposed by the length of a BASIC program line and the amount of memory available.

Memory Considerations and Arrays

Arrays, as we have already mentioned, use up memory. In fact, they are the primary users of memory in BASIC. After all, whereas it is hard to write a 6,000-line program, it takes only a second to write the statement

```
DIM A(6000)
```

which instantly sets aside 6,001 memory locations. When dimensioning an array, each location takes up memory space, which is

measured in what are known as *bytes*. As you may already know, the Commodore 64 (as its name implies) has a maximum capacity of 64K (more than 64,000 bytes). Any request for space beyond the physical capacity of the system generates the message

OUT OF MEMORY ERROR

After some memory overhead (a section of memory that the computer needs for internal processing) is subtracted, the remaining number of free bytes (38911) is displayed on the screen when BASIC is first entered. Each element of an array uses up 5 bytes of memory. Therefore, the preceding example uses $6,001 \times 5$, or 30,005 bytes to store the data, plus another 8 bytes for overhead. In other words, this single innocuous little dimension statement uses approximately 75 percent of the available memory.

Suppose you need to dimension an array of size 7500 in a program. At a later point in the program, another array of size 6000 is needed—for a totally different purpose. If the data stored in the first array are still needed, you are stuck—you will have run headlong into the memory barrier. If, however, there is no longer any need to access the data in the first array, there are two alternatives. The first possibility is to reuse the array by reinitializing it in a loop (erasing all 6,000 elements creates a substantial delay) and then to use the array for the new purpose. The major objection to this course of action is not the time factor, however, but the fact that it is bad programming practice and confusing to use the same array for two totally different purposes. The other, more preferable choice, is to erase the entire array from memory and start from scratch by dimensioning a new one. This may be done by using the CLR (clear) command, which simply takes the form

line number CLR

Unfortunately, the CLR statement is not selective; it destroys all variables (arrays, strings, normal variables, records required for FOR . . . NEXT loops that are not yet satisfied, and so on), not just the desired array. It is therefore very important to be sure that

the CLR instruction is not invoked within a FOR . . . NEXT loop or a subroutine or when the value of a variable is needed for further computation.

It is worth noting that arrays consisting solely of integers use only 2 bytes per element. Therefore it is worthwhile to use integer arrays wherever they are feasible in order to conserve memory space. In this way, the problem of running out of memory may often be totally avoided.

In the following two pseudo-programs, identical arrays are used. In the first, however, an

?OUT OF MEMORY ERROR

is generated, whereas the second avoids the problem by erasing arrays that are no longer required by the program.

PROGRAM 7-5, VERSION 1

```

100 PRINT "*****"
110 PRINT "*"
120 PRINT "      MEMORY HOG VERSION 1"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 DIM A(4000),B(3000)
170 DIM X(5000)

```

RUN

```

*****
*
*      MEMORY HOG VERSION 1
*
*****

```

?OUT OF MEMORY ERROR IN 170

READY,

PROGRAM 7-5, VERSION 2

```

100 PRINT "*"
110 PRINT "*"
120 PRINT "*"          MEMORY HOG VERSION 2          "*"
130 PRINT "*"
140 PRINT "*"
150 PRINT
160 DIM A(4000),B(3000)
170 REM: HERE IT IS ASSUMED THAT THERE IS NO NEED
180 REM: FOR THE DATA STORED IN ARRAYS A AND B
190 CLR
200 DIM X(5000)

```

The FRE(x) Function

Although arrays are, in the average program, the largest consumers of memory, they are by no means the only ones. The program itself takes up memory space, and naturally, the larger and more complex it is, the more memory it uses. Since it is sometimes important to know precisely how much memory is free, Commodore BASIC provides a function that returns this important value. It takes a dummy numeric argument, so that the statements

```

X = FRE(0)
X = FRE(98.6)

```

each produce the identical result—the amount of free memory (specified in units called *bytes*, which are roughly equivalent to characters). In this example, this number is stored into the variable X. The FRE function may be used interchangeably with any numeric value, so that it may be used in statements such as the following:

```

Q = FRE(0)
PRINT FRE(1)
TRUE = FRE(0) > 2000

```

Review Questions

- 1 ■ What is an array?

An array is a sequence of items that share the same name and are distinguished from each other by a subscript that defines their positions in the array.

- 2 ■ What statement reserves room for an array?

The DIM statement.

- 3 ■ Is a dimension statement necessary for every array?

No; any array that is not dimensioned is automatically assigned 11 locations by the system. However, it is good programming practice to dimension every array used.

- 4 ■ What is the effect of accessing an element of an array beyond its dimensioned range?

The error message

```
?BAD SUBSCRIPT ERROR
```

is generated.

- 5 ■ May more than one array be dimensioned by the same DIM statement?

Yes. The array names in the DIM statement must merely be separated by commas.

- 6 ■ What is a dynamic array?

A dynamic array is one whose dimensioned size is specified by a variable and is assigned during execution of the program.

- 7 ■ What convention has been adopted in BASIC to indicate a subscript?

The subscript is enclosed within parentheses.

- 8 ■ What does the CLR command do?

It erases all variables, arrays, active FOR . . . NEXT loop storage, and so on.

- 9 ■ What is the meaning of the value returned by the FRE function?

It is the number of unused bytes left in BASIC memory.

Advanced String Manipulation

Now that you have covered FOR . . . NEXT loops and some of the more common string functions, the time has come to learn some more about manipulating characters contained within strings. In this chapter you will learn

- how to read character strings from DATA statements
- string arrays
- the ASC and CHR\$ functions
- the ASCII character code
- a simple method of testing for palindromes
- the GET statement

Reading String Values from DATA Statements

The READ . . . DATA combination, which allows data to be listed in a DATA statement and read by the READ statement, can store literals, as well as numeric items. The strings may be enclosed in quotation marks (or not) at the discretion of the programmer.

In the following program the DATA statements contain an employee's name and social security number. As each is read, it is simply printed out.

PROGRAM 8-1

```

100 PRINT "*"*****"
110 PRINT "*"
120 PRINT "*" READ/DATA WITH STRINGS "*"
130 PRINT "*"
140 PRINT "*"*****"
150 PRINT
160 READ EMPNAME$,SSN
170 IF EMPNAME$ = "THE LAST ONE" THEN END
180 PRINT EMPNAME$,SSN
190 GOTO 160
200 DATA "ABRAHAM LINCOLN",132444583
210 DATA "STEPHEN DECATUR",999777555
220 DATA "GEORGE WASHINGTON",123456789
230 DATA "NANCY REAGAN",100000001
240 DATA "THOMAS JONES",111222333
250 DATA "THE LAST ONE",0

```

RUN

```

*****
*
* READ/DATA WITH STRINGS *
*
*****

```

```

ABRAHAM LINCOLN 132444583
STEPHEN DECATUR 999777555
GEORGE WASHINGTON 123456789
NANCY REAGAN 100000001
TOM JONES 111222333

```

READY.

Within the loop, successive pairs of names and corresponding social security numbers are read from DATA statements. The trailing items are "THE LAST ONE", 0. The 0 is added at the end because two values are read every time the READ statement is executed. The last time around, when the trailer items are read, two values must be supplied, or the program is terminated with an

ROUT OF DATA ERROR

It should be pointed out that the DATA statements could have been merged into a single statement. However, since two items are read at a time, it is much clearer to a reader of the program if they are listed in the manner shown. For further clarity, the quotation marks are placed around each of the character strings to reinforce the notion that they are, indeed, strings.

String Arrays

Each of the arrays we have used so far contained elements that were numeric. This need not always be the case, however. Commodore BASIC supports arrays in which each element is a character string rather than a numeric quantity. Just as with numeric arrays, if no dimension statement is provided before a reference is made to the string array, the computer sets aside 11 locations (including the zero) for the string array. The only difference between the two types of arrays is that the name selected for the string array must, as you probably will have guessed, terminate with a dollar sign, in the usual way. The statement

```
DIM A$(100)
```

sets aside room for 101 string elements for the array A\$.

To help you become thoroughly familiar with the concept of string arrays, the following program is presented. It stores in the array DAY\$ the names of the seven days of the week. The user is asked to type in an integer N ranging from 1 to 7; the computer then prints out the the name of the corresponding day.

PROGRAM 8-2

```
100 PRINT "*****"
110 PRINT "*"
120 PRINT "* ILLUSTRATION OF STRING ARRAYS"
130 PRINT "*"
140 PRINT "*****"
```

```

150 PRINT
160 DIM DAY$(7)
170 DAY$(1) = "SUNDAY"
180 DAY$(2) = "MONDAY"
190 DAY$(3) = "TUESDAY"
200 DAY$(4) = "WEDNESDAY"
210 DAY$(5) = "THURSDAY"
220 DAY$(6) = "FRIDAY"
230 DAY$(7) = "SATURDAY"
240 PRINT
250 INPUT "WHICH DAY OF THE WEEK DO YOU WANT";N
260 PRINT "DAY NUMBER";N;"IS: ";DAY$(N)
270 PRINT:INPUT"MORE? (TYPE YES OR NO):";AGAIN$
280 IF AGAIN$ = "YES" THEN 240
290 IF AGAIN$ <> "NO" THEN PRINT "I BEG YOUR
    PARDON; TRY AGAIN.":GOTO 270

```

RUN

```

*****
*                                     *
*   ILLUSTRATION OF STRING ARRAYS   *
*                                     *
*****

```

```

WHICH DAY OF THE WEEK DO YOU WANT? 2
DAY NUMBER 2 IS MONDAY

```

```

MORE? (TYPE 'YES' OR 'NO'): ?YUP
I BEG YOUR PARDON; TRY AGAIN.

```

```

MORE? (TYPE 'YES' OR 'NO'): ?YES

```

```

WHICH DAY OF THE WEEK DO YOU WANT? 5
DAY NUMBER 5 IS THURSDAY

```

READY.

In this program, each element of the string array `DAY$` was defined by seven successive assignment statements. An alternative is to store the strings in `DATA` statements and `READ` them into the array from within a loop.

The ASC and CHR\$ Functions

Although on the surface it may appear that the computer is quite adept at manipulating characters, this is not exactly true. The computer actually treats all data—whether numeric or not—as numbers. Every symbol recognized by the Commodore 64 is assigned a specific numeric representation according to a modified version of the so-called ASCII (pronounced *as-key*) code. The acronym stands for American Standard Code for Information Interchange and is by far the most popular standard among all those used throughout the computing world. For your reference a complete table of the characters that can be generated on the Commodore 64 is included in Appendix C at the back of this book.

In BASIC, the ASCII value of any character may be obtained by the ASC function which takes the form

variable name = ASC(string value)

For example, if you were interested in the ASCII code for the letter “C” you could write

```
CODE = ASC("C")
PRINT CODE
67
```

READY.

or

```
PRINT ASC("C")
67
```

READY.

As do VAL and STR\$, ASC has a complementary, or reverse, function. It is called the “CHR\$ function” and returns the character string representation of any given ASCII value. Since the ASCII

code for capital letter "D" is 68, it may be deduced and confirmed on the computer that

```
CHR$(68)
```

returns the capital letter "D".

Testing for Palindromes

We shall now turn to matters of literature, instructing the computer to operate not on numeric data but on textual material. For your first venture into this field you shall type in an English phrase (the phrase could be expressed in any language, really, as long as it is written with the Latin alphabet, the letters shown on the keys of your Commodore computer).

A palindrome is a phrase that reads the same backwards and forwards. There are many such words in the English language, including

I, eye, peep, level, madam, and pop,

to name a few. The classical phrase that is a true palindrome is

ABLE WAS IERE I SAW ELBA

where without any change whatever, the phrase reads exactly the same backwards and forwards. The purpose of the next program is to identify such palindromes.

PROGRAM 8-3

```
100 PRINT "*"*****"
110 PRINT "*"
120 PRINT "*" SIMPLE PALINDROME TESTER "*"
130 PRINT "*"
140 PRINT "*"*****"
```

```

150 PRINT
160 INPUT "PLEASE ENTER YOUR PHRASE:";PHRASE$
170 FOR I = LEN(PHRASE$) TO 1 STEP -1
180 REVERSE$ = REVERSE$ + MID$(PHRASE$,I,1)
190 NEXT I
200 IF PHRASE$ = REVERSE$ THEN PRINT "YOUR PHRASE
    IS A PALINDROME":END
210 PRINT "YOUR PHRASE IS NOT A PALINDROME"

```

RUN

```

*****
*                                     *
*      SIMPLE PALINDROME TESTER      *
*                                     *
*****

```

```

PLEASE ENTER YOUR PHRASE: ? MADAM
YOUR PHRASE IS A PALINDROME

```

READY.

RUN

```

PLEASE ENTER YOUR PHRASE: ? REALLY
YOUR PHRASE IS NOT A PALINDROME

```

READY.

In this program the user types in a phrase which is stored internally under the string variable name PHRASE\$. Within the FOR . . . NEXT loop, extending from line 170 to 190, each character of the phrase (beginning with the last) is added to the string variable REVERSE\$, which since it hasn't been initialized, is initialized by default to the null string. When the FOR . . . NEXT loop has done its work, the reverse of the original string will be stored in REVERSE\$. In line 200 this reversed version of the string is tested for equality with the original string residing in PHRASE\$. If they prove to be equal to each other, the phrase must be a palindrome; otherwise it is not. In either case, an appropriate statement is printed.

Elementary Textual Analysis

With all its mathematical abilities the Commodore 64 can also actually help you become aware of—and improve—your writing. One way you can do so is to perform a statistical analysis of your text, which can reveal whether you use certain words too frequently, your sentences are of a (too?) consistent length, and many other interesting stylistic features.

Analyses of textual material often demand the ability to count the number of words in a given passage. The best strategy for counting the words is simply to count the number of intervening spaces. Then 1 is added to the result because the first word is not preceded by a space. In the following program, the average word length of an inputted phrase is computed and printed out.

PROGRAM 8-4

```

100 PRINT "*****"
110 PRINT "*"
120 PRINT "      COUNTING WORDS IN A PASSAGE"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 INPUT "PLEASE ENTER YOUR TEXT: ";PHRASE$
170 LENGH = LEN(PHRASE$):COUNT = 0
180 FOR I = 1 TO LENGH
190 IF MID$(PHRASE$,I,1) = CHR$(32) THEN COUNT
    = COUNT + 1
200 NEXT I
210 AVLNG = (LENGH - COUNT) / (COUNT + 1)
220 PRINT
230 PRINT "IN THE PHRASE: ";PHRASE$
240 PRINT "THERE ARE:";COUNT + 1;"WORDS WITH AN
    AVERAGE LENGTH OF:";AVLNG

RUN
*****
*
*      COUNTING WORDS IN A PASSAGE
*
*****

```

```
PLEASE ENTER YOUR TEXT: ? WHERE THERE'S A WILL
THERE'S A WAY
```

```
IN THE PHRASE: WHERE THERE'S A WILL THERE'S A WAY
THERE ARE: 7 WORDS WITH AN AVERAGE LENGTH OF: 4
```

```
READY.
```

In this program, the user is invited to input any phrase at all. First, the length of the phrase is determined by the LEN function, and COUNT, representing the count of spaces found, is set to zero. The number of words present is then found by examining the length of the string for intervening spaces. Once the number of spaces has been found, 1 is added to reflect the number of words present. Line 190 performs this test by comparing each character of the string to the character with the ASCII code 32, which represents a space. You might well ask why the comparison wasn't made with a space directly—in other words, with the literal " ". The answer is that the Commodore 64 behaves rather strangely on this score and, counter to convention, treats the space within quotes as the lower-case space (ASCII code 160) rather than the normal 32. When ASCII character 160 is compared to what was typed in (ASCII character 32) the comparison fails; the two are not the same. We therefore must resort to the CHR\$ function.

Once the total number of words present is calculated, the average length per word is found by subtracting the number of spaces (stored in COUNT) from LENGTH (as they are not part of the word) and dividing by the number of words found (COUNT + 1).

A common feature of word processing is the ability to replace all occurrences of a given string with another string. For example, a typist not very expert at spelling might have erroneously spelled the word *receive* as *recieve*. If this error has been committed consistently throughout a long report, a considerable amount of time and effort may be saved by resorting to this "search and replace" feature. In the following program, a simple illustration of this type of problem is presented. This rudimentary program only works on one string's worth of text (up to 255 characters). In all the essentials, however, it duplicates this most useful feature of full-fledged word processors.

PROGRAM 8-5

```

100 PRINT "*"
110 PRINT "*"
120 PRINT "SEARCH AND REPLACE"
130 PRINT "*"
140 PRINT "*"
150 PRINT
160 INPUT "ENTER SENTENCE TO BE SCANNED:";SENT$
170 INPUT "REPLACE ALL OCCURRENCES OF:";REP$
180 INPUT "WITH:";SUB$
190 LENGTH = LEN(SENT$):REPLNGTH = LEN(REP$)
200 OUT$ = ""
210 COUNT = 0
220 I = 1
230 IF REP$ = MID$(SENT$,I,REPLNGTH) THEN GOSUB
    1000:GOTO 250
240 OUT$ = OUT$+ MID$(SENT$,I,1)
250 I = I + 1: IF I <= LENGTH THEN 230
260 PRINT
270 PRINT "THE AMENDED SENTENCE IS NOW:"
280 PRINT OUT$
290 PRINT COUNT;"REPLACEMENT(S) WERE MADE"
300 PRINT
310 INPUT "DO YOU WANT TO GO AGAIN (YES OR NO)";
    AGAIN$
320 IF AGAIN$ = "YES" THEN 150
330 IF AGAIN$ <> "NO" THEN PRINT "TYPE YES OR NO,
    NOT: ";AGAIN$:GOTO 310
340 END
1000 OUT$ = OUT$ + SUB$
1010 I = I + REPLNGTH - 1
1020 COUNT = COUNT + 1
1030 RETURN

```

RUN

```

*****
*
*          SEARCH AND REPLACE
*
*****

```

ENTER SENTENCE TO BE SCANNED: ? MAY THE BEST MAN
WIN

REPLACE ALL OCCURRENCES OF: ? BEST
WITH: ? WORST

THE AMENDED SENTENCE IS NOW:
MAY THE WORST MAN WIN.

1 REPLACEMENT(S) WERE MADE

DO YOU WANT TO GO AGAIN (YES OR NO)? YES

ENTER SENTENCE TO BE SCANNED: ? HE WHO LAUGHS
LAST LASTS LAST

REPLACE ALL OCCURRENCES OF: ? LAST
WITH: ? FIRST

THE AMENDED SENTENCE IS NOW:
HE WHO LAUGHS FIRST FIRSTS FIRST

3 REPLACEMENT(S) WERE MADE

DO YOU WANT TO GO AGAIN (YES OR NO)? MAYBE

TYPE YES OR NO, NOT: MAYBE

DO YOU WANT TO GO AGAIN (YES OR NO)? NO

READY.

In this program, the user is invited to enter any sentence, which is stored in the variable SENT\$. The user is then asked to type in the string of characters for which SENT\$ is to be scanned, which is stored in REP\$. As a third input, a request is made to type in the desired replacement for any occurrences found in SENT\$. From this point on, the program runs under its own steam. The length of SENT\$ is stored in LENGTH, and the length of REP\$ is stored in the variable REPLNGTH. COUNT, which stands for the number of times a replacement is made, is initially set to zero. Then, within a loop, successive slices of SENT\$ are compared to REP\$. If a match is successful, the replacement string, SUB\$, is tacked on to the end of OUT\$, which starts out as the null string. One is also

added to the contents of the variable COUNT, meaning that a replacement has been made. Finally, REPLENGTH - 1 is added to the index I to skip over the body of the matching substring. If the match is unsuccessful, the letter that did not match is added to the end of OUT\$ and the substring (beginning at the subsequent letter) is examined. The process is shown in stages in the following example:

```
SENT$ = "THE CAT SAT ON THE MAT"
REP$ = "AT"
SUB$ = "LEW"
COUNT = 0
OUT$ = ""
```

is:

```
"TH" = "AT" ? (no)   OUT$ = "T"
"HE" = "AT" ? (no)   OUT$ = "TH"
"E  " = "AT" ? (no)   OUT$ = "THE"
" C" = "AT" ? (no)   OUT$ = "THE "
"CA" = "AT" ? (no)   OUT$ = "THE C"
"AT" = "AT" ? (yes!)  OUT$ = OUT$ + "LEW"
    and COUNT = COUNT + 1 (now it is 1)
    I = I + REPLNGTH - 1 (now it is 7)
    I = I + 1 (now it is 8)
" S" = "AT" ? (no)   OUT$ = "THE CLEW "
[and so on]
```

By the time this program is finished, of course, the sentence will be the nonsensical "THE CLEW SLEW ON THE MLEW".

Another feature of this program is that the answer to the "go again" question is tested for both "YES" and "NO". Previously, we assumed that any response other than "YES" was "NO". In other words, a response of "Y", "YEA", or "YUP", all of which are probably intended as a positive response, were taken to be negative. Even an ambivalent answer of "MAYBE" was regarded as a negative response. In this program, however, if AGAIN\$ is not

equal to "YES", a test is made as to whether it is "NO". If it is neither "YES" nor "NO", it is clear that an improper response has been made. Whatever this response was—be it "MAYBE", "YEP", or something like "GO JUMP IN A LAKE"—it is stored in AGAIN\$ and is printed out in an admonishing tone in the sentence

```
TYPE YES OR NO, NOT: (whatever was typed in)
```

In this way, the programmer insures that only a "YES" or "NO" response is accepted. Even at the expense of some extra labor, this type of "error trapping" is an excellent idea.

The GET Command

When a user types data into the computer interactively, time is not usually important, the primary consideration being accuracy. However, there are many classes of problems in which the time taken for a response is critical. For example, a timed-reponse quiz must wait for the allotted time only and then move on to the next question—whether or not a response has been entered. In "shoot-'em-up" games, the opposing electronic army must move to the attack whether or not the player responds. Obviously, the INPUT statement—which waits indefinitely for a response, followed by a RETURN, before resuming execution—cannot be used for this purpose. The GET command is the BASIC programming tool designed to solve these and other similar problems. Before illustrating this command, however, we must introduce a new concept—namely, the *keyboard buffer*.

Before being accessed by the computer, all typed characters are stored into the keyboard buffer, an intermediate storage area which temporarily holds up to ten typed characters until the computer is ready to operate on them. Usually the computer accesses the typed characters so rapidly that it is not apparent that they are stored at all. To demonstrate that the buffer exists, therefore, we shall artificially keep the computer "busy" by executing a delay loop 10,000 times. If characters are typed in during execution of the loop, no

action is taken, but they will appear almost instantly as soon as the INPUT statement is executed. If more than ten characters are typed in, only the first ten will appear—the rest being lost because of “buffer overload.”

PROGRAM 8-6

```

100 PRINT "*****"
110 PRINT "*"
120 PRINT "*" ILLUSTRATION OF INPUT BUFFER "*"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 PRINT "TYPE IN SOME LETTERS NOW"
170 FOR DELAY = 1 TO 10000
180 NEXT DELAY
190 INPUT A$

```

As opposed to the INPUT statement, GET scans the keyboard buffer for one character. If it is there, the character is returned into the specified variable name. For example, the instruction

```
100 GET A$
```

removes the first character from the buffer and stores it into A\$. Similarly, the statement

```
100 GET B
```

removes the first character from the buffer and places it in the numeric variable B. It is not advisable to use this version of the GET statement, however, as a ?SYNTAX ERROR is generated when a character other than a digit is typed in.

In the next program, the GET statement is used to scan the keyboard repeatedly and terminate the loop when the first key is pressed. At that point, the message “GOODBYE FOR NOW” is printed.

PROGRAM 8-7

```

100 PRINT "*****"
110 PRINT "*"
120 PRINT "* ILLUSTRATION OF THE GET COMMAND *"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 PRINT "HIT ANY KEY TO CONTINUE"
170 GET GARBAGE$: IF GARBAGE$ = "" THEN 170
180 PRINT "GOODBYE FOR NOW"

```

In the next program, advantage is taken of the TIME function in conjunction with the GET statement. The purpose of the program is to simulate a sophisticated stopwatch. The user is asked to hit the S key to start and stop the clock. In addition, lap time may be obtained by pressing the L key, the clock may be reset to zero by pressing C, and the program may be terminated by typing Q once the stopwatch is started. Since the elapsed time is stored in sixtieths of a second, it is first divided by 60 before being printed out.

PROGRAM 8-8

```

100 PRINT "*****"
110 PRINT "*"
120 PRINT "* STOPWATCH SIMULATOR *"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 PRINT "HIT S TO START AND STOP THE STOPWATCH"
170 T = 0
180 GET COMMND$
190 IF COMMND$ = "C" THEN PRINT "CLEAR":GOTO 170
200 IF COMMND$ <> "S" THEN 180
210 BEGIN = TIME: PRINT "START": PRINT
220 GET COMMND$
230 IF COMMND$ = "L" THEN PRINT "LAP:";T + (TIME
- BEGIN) / 60:GOTO 220
240 IF COMMND$ = "S" THEN T = T+(TIME-BEGIN)/60
PRINT:PRINT T; "STOP":GOTO 180

```

```
250 IF COMMND$ <> "Q" THEN 220
260 PRINT "QUIT"
```

By now you will have a fairly good idea of what string manipulation is all about. By combining these techniques with other programming features, quite sophisticated and useful effects can be accomplished. The results are really remarkable because, after all, the computer is completely inanimate, composed as it is of metal, plastic, and wires. It relies on your creativity to be truly useful and entertaining.

Review Questions

- 1 ■ What kinds of data may be stored in a DATA statement?

Both numeric and alphanumeric.

- 2 ■ Must alphabetic data stored in a DATA statement be enclosed in quotation marks?

No. The comma separating the data items functions as a delimiter (separator) and therefore the quotes are optional. If a comma is desired as part of the string, however, that string must be placed in quotation marks.

- 3 ■ What is implied by the following statement?

```
DIM Z$(5000)
```

That room has been reserved in memory for 5,001 elements of the character string array Z\$.

- 4 ■ Is the following statement valid?

```
DIM A(T)
```

No. The subscript of an array must have a numeric value.

- 5 ■ What is the purpose of the ASC function?

It returns the ASCII value of the given character. If a string of more than length 1 is specified, the value returned is the ASCII value of the first character.

- 6 ■ What does the following statement do?

```
PRINT CHR$(ASC("D"))
```

It displays the capital letter "D" on the screen because the ASC function first returns the ASCII code of the literal "D" (which is 68). The CHR\$ function then accepts the given number and returns the corresponding character (in this case, the original character).

- 7 ■ Why is COMMND\$ used instead of COMMAND\$ in Program 8-8?

Because the name COMMAND contains the keyword AND.

HANDS-ON PRACTICE

1. If a DATA statement contains literals, they may or may not be enclosed in quotation marks at the discretion of the programmer. Write a program to convince yourself of the truth of this statement.

TRY YOUR HAND AT THESE

- 1a. Write a program that reads the twelve months of the year from DATA statements into an array.
- 1b. Modify this program so that it allows the user to type in the number of a month and returns the name of the month.
- 1c. Allow for repeated use and accept only "OUI" and "NON" as valid responses.
2. Write a program that counts the number of vowels in any inputted string.

3. Write a program that accepts any input until the RETURN key is hit and stores it in the variable A\$. (Note: INPUT does not work in this manner, since it recognizes a comma as a separator, so use GET in a loop.)

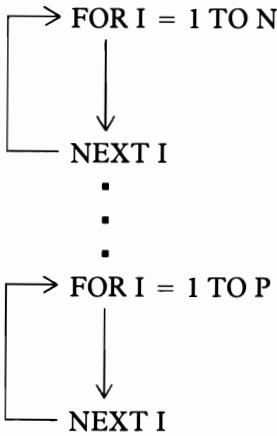
Nesting Loops

In this chapter you will be introduced to some of the most powerful features found in the BASIC language and some entertaining examples of their applications. These include

- nests of FOR . . . NEXT loops
- the sum-of-the-digits-cubed problem
- the word-rearrangement problem
- the telephone problem
- nesting other loop constructs
- nesting subroutines
- nesting subroutines and loops

Nested FOR . . . NEXT Loops

In all the programs illustrated so far in which several loops appeared, there was no connection between one loop and the other. Each FOR . . . NEXT loop was executed whenever it appeared in the normal sequence of the program and was “satisfied” before the subsequent FOR . . . NEXT loop was entered. Thus they could be described as being independent of each other. Being independent, they could all have the same index variables; it really did not matter because each FOR statement initialized the value of the index variable when it was first encountered. The following diagram illustrates this concept:



[and so on]

It is possible in BASIC, however, to enclose completely one FOR . . . NEXT loop within another. Such an arrangement is called a *nest* of loops because it is reminiscent of the way a bird builds its nest—layer on layer. As you shall soon see, a nest of loops is a very powerful programming tool.

In the following program, a nest of loops is created with an outer index I and an inner index J.

PROGRAM 9-1, VERSION 1

```

100 PRINT "*"*****"
110 PRINT "*"                               "*"
120 PRINT "*"      NESTING LOOPS VERSION 1  "*"
130 PRINT "*"                               "*"
140 PRINT "*"*****"
150 PRINT
160 PRINT " I", " J"
170 PRINT
180 FOR I = 1 TO 3
190 FOR J = 1 TO 2
200 PRINT I, J

```



```
210 NEXT J
220 NEXT I
```

```
RUN
```

```
*****
*                                     *
*      NESTING LOOPS VERSION 1      *
*                                     *
*****

  I          J

  1          1
  1          2
  2          1
  2          2
  3          1
  3          2
```

```
READY.
```

A careful look at the output of this program shows that the inner loop is satisfied each time the outer loop goes around once. This concept is easily visualized when the identical program is put into the following form:

```
PROGRAM 9-1, VERSION 2
```

```
100 PRINT "*****"
110 PRINT "*"
120 PRINT "      NESTING LOOPS VERSION 2"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 PRINT " I", " J"
170 PRINT
180 FOR I = 1 TO 3
190 FOR J = 1 TO 2:PRINT I,J:NEXT J
200 NEXT I
```

In this form, it is evident that line 200 is executed each time around the loop. Each time the line is executed, the FOR . . . NEXT loop with the index variable J goes through a complete cycle. Since many people feel that this form is intuitively easier to follow, this might be the preferred way.

Whenever loops are nested, their index variables must have different names. If they are the same, the following situation arises. (We use variable indents in these samples to illustrate the beginning and end of nested loops; the computer, however, will delete the extra spaces.)

```

100 FOR I = 1 TO 3
110   FOR I = 1 TO 2
120     PRINT I
130   NEXT I
140 NEXT I

```

RUN

```

1
2

```

```

?NEXT WITHOUT FOR ERROR IN 140
READY.

```

As stated earlier, the NEXT statement need not take a variable; indeed it is increasingly common not to include the index variable name. This omission makes for slightly increased speed, even if it loses a little self-documentation. For purposes of clarity, however, beginners should always include the index variable name. If the index variable is omitted from the NEXT statement, it is associated with the last unclosed FOR statement. Thus, for example, the following program segment is perfectly legal on the Commodore 64:

```

1000 FOR I = 1 TO 995
1010   PRINT I
1020 NEXT

```

It is equivalent to

```

1000 FOR I = 1 TO 995
1010   PRINT I
1020 NEXT I

```

The same principle applies to nested loops, as may be seen from the following example:

```

560 FOR I = 1 TO 2
570   FOR J = 1 TO 4
580     PRINT "GET IT?"
590     FOR K = 1 TO 15
600       PRINT I,J,K
610     NEXT K
620   NEXT J
630 NEXT I

```

As before, omitting the index variable from the NEXT statement is perfectly acceptable and is equivalent to the following:

```

560 FOR I = 1 TO 2
570   FOR J = 1 TO 4
580     PRINT "GET IT?"
590     FOR K = 1 TO 15
600       PRINT I,J,K
610     NEXT K
620   NEXT J
630 NEXT I

```

If nested loops have the same endpoint, a single NEXT statement may be used for all of them. In the following program, such a NEXT statement is used, saving a line.

PROGRAM 9-2

```

100 PRINT "*"*****"
110 PRINT "*"
120 PRINT "*"   ILLUSTRATION OF MULTIPLE NEXTS
130 PRINT "*"
140 PRINT "*"*****"

```

```

150 PRINT
160 FOR I = 1 TO 4
170 FOR J = 2 TO 3
180 FOR K = 5 TO 6
190 PRINT I,J,K
200 NEXT K,J,I

```

Lines 170 through 200 are exactly equivalent to

```

170 FOR I = 1 TO 4
180   FOR J = 2 TO 3
190     FOR K = 5 TO 6
200       PRINT I,J,K
210     NEXT K
220   NEXT J
230 NEXT I

```

The order of the index variables as listed in the NEXT statement is of vital importance since, when loops are nested, they must not intersect. The reason for this restriction is shown in the following flawed program segment, where an error message is triggered:

```

100 FOR X = 1 TO 3
110   FOR Y = 1 TO 2
120     PRINT X,Y
130   NEXT X
140 NEXT Y

```

RUN

```

1      1
2      1
3      1
4      2

```

```

?NEXT WITHOUT FOR ERROR IN 130
READY.

```

For those who prefer graphic illustrations to the numeric examples shown, the following program containing a nest of loops might be of interest.

PROGRAM 9-3

```

100 PRINT "*****"
110 PRINT "*"
120 PRINT "* GRAPHICS ILLUSTRATION OF NESTING *"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 FOR I = 1 TO 3
170 PRINT "*****"
180 FOR J = 1 TO 2
190 PRINT "      :::::::::::"
200 NEXT J,I

```

RUN

```

*****
*
* GRAPHICS ILLUSTRATION OF NESTING *
*
*****

*****
      :::::::::::
      :::::::::::
*****
      :::::::::::
      :::::::::::
*****
      :::::::::::
      :::::::::::

```

READY.

Nesting has many practical applications. For example, in the following program a multiplication table of the numbers between any two inputted integers is printed out. (In the two sections that follow we will explore how nests are used to solve a math problem and to manipulate character strings.)

PROGRAM 9-4

```

100 PRINT "*****"
110 PRINT "*"
120 PRINT "* PRINTING A MULTIPLICATION TABLE *"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 INPUT "ENTER YOUR TWO INTEGERS:";A,B
170 IF A > B THEN TEMP = A: A = B: B = TEMP
180 PRINT
190 FOR I = A TO B
200 FOR J = A TO B
210 PRINT I;"*";J;TAB(10);"=";I * J
220 NEXT J
230 NEXT I

```

RUN

```

*****
*
* PRINTING A MULTIPLICATION TABLE *
*
*****

```

ENTER YOUR TWO INTEGERS: ? 5,1

```

1 * 1    = 1
1 * 2    = 2
1 * 3    = 3

```

[and so on]

READY.

The Sum-of-the-Cube-of-the-Digits Problem

This is an interesting mathematics problem that requires nesting to solve. The problem is to print out all the three-digit numbers that equal the sum of the cubes of the individual digits. One such number is 153 because

$$153 = (1 \uparrow 3) + (5 \uparrow 3) + (3 \uparrow 3)$$

One method of solving the problem is to set up a nest of three loops, where the index of each loop corresponds to one of the digits of the number. This approach is taken in the following program, which prints out four such numbers.

PROGRAM 9-5

```

100 PRINT "*****"
110 PRINT "*"
120 PRINT "  THE SUM-OF-THE-CUBES PROBLEM  "
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 FOR H = 1 TO 9
170 FOR T = 0 TO 9
180 FOR U = 0 TO 9
190 IF H*H*H + T*T*T + U*U*U = 100*H + 10*T + U
    THEN PRINT 100*H + 10*T + U
200 NEXT U,T,H

```

RUN

```

*****
*
*   THE SUM-OF-THE-CUBES PROBLEM   *
*
*****

```

```

153
370
371
407

```

READY.

The control variables H, T, and U correspond to the hundreds, tens, and units digits of all possible three-digit numbers. Since the numbers below 100 do not qualify as three-digit numbers, the digit

stored in H ranges from 1 to 9 (rather than 0 to 9), which would yield all the numbers from 0 to 999. For each increment in the H loop, the T digit moves from 0 to 9, as does the U digit for each increment of the T digit. For each of the 900 combinations, the cubes of the H, T, and U digits are added and compared to the number itself—100 times the hundreds digit, plus 10 times the tens digit, plus the units digit. If the two values are equal, the three-digit number is printed out by storing the value of the number in TEMP and printing it out. This step is taken because if the individual digits were printed out separately, they would not be displayed contiguously.

The Word-Combination Problem

In the following program, a five-letter word is rearranged into every possible combination of its component letters by means of five nested FOR . . . NEXT loops. You might be interested in running the program yourself and examining the output.

PROGRAM 9-6

```

100 PRINT "*****"
110 PRINT "*"
120 PRINT "* REARRANGING A FIVE-LETTERED WORD *"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 INPUT"ENTER YOUR FIVE-LETTERED WORD:";WD$
170 IF LEN(WD$) <>5 THEN 160
180 FOR I = 1 TO 5
190 FOR J = 1 TO 5
200 IF J = I THEN 320
210 FOR K = 1 TO 5
220 IF K = J OR K = I THEN 310
230 FOR L = 1 TO 5
240 IF L = K OR L = J OR L = I THEN 300
250 FOR M = 1 TO 5

```



```

260 IF M = L OR M = K OR M = J OR M = I THEN 290
270 PRINT MID$(WD$,I,1);MID$(WD$,J,1);MID$(WD$,
    K,1);
280 PRINT MID$(WD$,L,1);MID$(WD$,M,1);"  ";
290 NEXT M
300 NEXT L
310 NEXT K
320 NEXT J
330 NEXT I

```

The inputted word is first checked to be sure that it is composed of five letters. If it isn't, it is rejected and another word is requested. Once it has passed the five-letter test, each letter of the word is successively jumbled by printing the first, second, third, fourth, and fifth letter in each of the possible positions. This step is accomplished by means of a nest of five FOR . . . NEXT loops in which provision is made to ignore all those combinations where the value of the indexes are the same. Whenever this happens (for example, when J is equal to I) control is sent to the corresponding NEXT statement, effectively ignoring such combinations.

The Telephone Problem

When new telephone numbers are issued, they are invariably given in the form of seven-digit numbers rather than the old system of two letters followed by five digits. The change was necessitated because of the vast increase in telephone numbers that were required across the United States. Contrary to what the telephone authorities might say, it is not always easy to remember a seven-digit number. The purpose of the following program is to try to alleviate this difficulty. If you examine the dial on the typical telephone, you will notice that the number 2 is associated with the letters A, B, and C; 3 with D, E, and F; and so on. The digits 1 and 0 have no letters associated with them, nor do the letters Q or Z appear on the dial.

In the program that follows, a user is asked to type in a seven-digit telephone number. The seven-character number is accepted as a string and is then sliced into its component parts, each of which is

stored into the array PHNE. (PHNE is not called PHONE because the latter contains the keyword ON.) The conversion from the string to its numeric representation is accomplished by slicing it with the MID\$ function, taking the ASCII value, and subtracting 48. Since the ASCII code for 0 is 48, 49 is the code for 1, and so on, this operation places the value of the digit typed into the array PHNE.

Once the seven digits have been entered into the array PHNE, a nest of seven loops is used to compose seven-character words consisting of every possible combination of the letters associated with the digits. Since there are $3 \uparrow 7$ combinations, the output consists of well over 2,000 words—from which, it is hoped, at least one suitable name can be selected that can be more easily remembered than the seven-digit number. Since the digits 0 and 1 have no alphabetic equivalents, they are retained as is. One warning though: If you run this program, be prepared for voluminous output.

PROGRAM 9-7

```

100 PRINT "*"
110 PRINT "*"
120 PRINT "    THE TELEPHONE NUMBER PROBLEM"
130 PRINT "*"
140 PRINT "*"
150 PRINT
160 DIM REF$(9),PHNE(7)
170 FOR I = 0 TO 9
180 READ REF$(I)
190 NEXT I
200 INPUT "ENTER YOUR TELEPHONE NUMBER:";PHNE$
210 FOR I = 1 TO 7
220 PHNE(I) = ASC(MID$(PHNE$,I,1)) - 48
230 NEXT I
240 PRINT:PRINT
250 FOR A = 1 TO 3
260 FOR B = 1 TO 3
270 FOR C = 1 TO 3
280 FOR D = 1 TO 3
290 FOR E = 1 TO 3
300 FOR F = 1 TO 3
310 FOR G = 1 TO 3

```

```

320 PRINT MID$(REF$(PHNE(1)),A,1);
330 PRINT MID$(REF$(PHNE(2)),B,1);
340 PRINT MID$(REF$(PHNE(3)),C,1);
350 PRINT MID$(REF$(PHNE(4)),D,1);
360 PRINT MID$(REF$(PHNE(5)),E,1);
370 PRINT MID$(REF$(PHNE(6)),F,1);
380 PRINT MID$(REF$(PHNE(7)),G,1) " ";
390 NEXT G,F,E,D,C,B,A
400 DATA "000","111","ABC","DEF","GHI","JKL",
"MN0","PRS","TUV","WXY"

```

Nesting Other Structures

Not only can FOR . . . NEXT loops be nested, but so may GOTO loops and IF . . . THEN statements. For example, the following program segment performs a “counting” FOR . . . NEXT loop for as long as the user enters a “C” for “continue.”

```

1000 FOR I = 1 TO 10
1010   PRINT I;
1020 NEXT I
1030 PRINT
1040 INPUT "HIT C TO CONTINUE: ";AGAIN$
1050 IF AGAIN$ = "C" THEN 1000

```

Loops, moreover, are not the only structures that can be nested. Subroutines may also be nested—and frequently, are, as shown in the following program segment:

```

1000 GOSUB 2000
1010 PRINT "BACK IN THE MAIN ROUTINE"
1020 END
2000 PRINT "SUBROUTINE A"
2010 GOSUB 3000
2020 PRINT "BACK IN SUBROUTINE A"
2030 RETURN
3000 PRINT "SUBROUTINE B"
3010 PRINT "STILL IN SUBROUTINE B"
3020 RETURN

```

```

RUN
SUBROUTINE A
SUBROUTINE B
STILL IN SUBROUTINE B
BACK IN SUBROUTINE A
BACK IN THE MAIN ROUTINE

READY.

```

In this program segment, control is initially sent to the subroutine beginning in line 2000. Once there, the literal "SUBROUTINE A" is printed. At that point, the subroutine beginning in line 3000 is invoked, which has the effect of printing out the literals "SUBROUTINE B" and "STILL IN SUBROUTINE B". Execution of the RETURN statement in line 3020 sends control back to line 2020, which prints the message "BACK IN SUBROUTINE A". Control then drops to the next line, which returns control to line 1010. Subsequently the message "BACK IN THE MAIN PROGRAM" is printed out, at which time the program segment terminates.

It is also possible to call a subroutine from within a FOR . . . NEXT loop. This is not regarded as an exit from the loop. For example, the following segment of code is perfectly legal:

```

1000 FOR I = 1 TO 5
1010 GOSUB 2000
1020 NEXT I
1030 END
2000 PRINT I, 2 * I
2010 RETURN

```

```

RUN
1          2
2          4
3          6
4          8
5         10

```

```

READY.

```

Review Questions

- 1 ■ What is a nest of loops?

A nest of loops is a structure in which at least one loop is completely enclosed within another.

- 2 ■ What must be true about the names of the index variables in nested FOR . . . NEXT loops?

They must each have a different name.

- 3 ■ What is the result if nested FOR . . . NEXT loops are given identical index-variable names?

A ?NEXT WITHOUT FOR ERROR diagnostic message is displayed and the program terminated.

- 4 ■ What happens if the order of the indexes as specified in the NEXT statement or statements does not reflect the order as specified in the FOR statements?

A ?NEXT WITHOUT FOR ERROR message is displayed and the program is terminated.

- 5 ■ Is it mandatory for the NEXT statement to include the index variable name?

No, it is perfectly correct to omit it.

- 6 ■ What is the effect of omitting the index-variable name in a NEXT statement?

The NEXT automatically references the last unclosed FOR statement.

- 7 ■ What will the following program print?

```
100 FOR I = 1 TO 3
110   PRINT "I ";
```

```

120   FOR J = 1 TO 4
130     PRINT "J ";
140   NEXT J
150 NEXT I

I J J J J I J J J J I J J J J

READY.

```

HANDS-ON PRACTICE

1. Write a one-line instruction to print out the pairs of numbers as follows:

```

(1, 1) (2, 2) (3, 3)
(2, 1) (2, 2) (2, 3)
(3, 1) (3, 2) (3, 3)

```

2. Write equivalent statements to

```

FOR I=1 TO 3:FOR J=1 TO 3:FOR K=4 TO 3.2
STEP -.1:NEXT K,J,I

```

TRY YOUR HAND AT THESE

1. Write a program to generate the sum of the factorials from 1 to N, where N is a user-inputted integer. (The x factorial is defined mathematically as $x(x - 1)(x - 2) \dots (1)$. For example, 5 factorial (also written 5!) is equal to $5(4)(3)(2)(1) = 120$.)
2. Write a program to print out the prime numbers from 3 to 1,000. (A prime number is an integer that is not evenly divisible by any integer other than itself and 1, for example, 17.)

Audio-Visual Program Enhancement

Up to now we have explored the way BASIC handles numeric and textual data on the Commodore 64. In this chapter we will introduce what are known as the graphics and sound-generation features of the Commodore 64—those features that affect the screen display and the waveforms produced by the highly sophisticated music synthesizer. In particular you will learn about

- the special graphics characters shown on many of the keys
- the Commodore ASCII character set
- converting a character to its ASCII equivalent
- converting an ASCII number to its equivalent character
- embedding control characters in quotation marks
- clearing the screen
- controlling the colors
- the PEEK function and the POKE command
- setting the background color of the screen
- incorporating sound into a program
- the general characteristics of a sound wave
- the four waveform types
- the Commodore SID chip
- setting the sound parameters: frequency, volume, envelope (ADSR), waveform

The Commodore 64 can produce a wide range of special characters known as *graphics characters*. Many of these characters can be combined on the screen (or printed out) to make designs that are both useful and beautiful. You will notice that on the front side of most of the keys, there are two strange-looking symbols enclosed in boxes. The righthand one is obtained by holding down the SHIFT while pressing the appropriate key, and the left one is obtained by simultaneously holding down the Commodore key and the required key. By using combinations of these characters it is possible to display a wide variety of shapes and designs that may enhance a program's display.

There are also lowercase characters and special "mode-changing" symbols that control the manner in which the entire display is represented. With so many characters available and (for technical reasons) with only 256 codes to represent them, the designers of the Commodore 64 resorted to the clever technique of having two different character sets. The difficulty is that characters from both sets cannot be displayed simultaneously. It is possible to switch between the two character sets by pressing the Commodore and SHIFT keys simultaneously. Normally, code number 65 represents the character "A", but when the Commodore-SHIFT combination is used every capital A on the screen instantly changes to lower case and all subsequently printed A's are also lower case. All graphics characters accessed by the SHIFT key are converted into upper case in a similar manner.

You are already familiar with the standard uppercase symbols used for writing programs since these are the only ones we have used in the text. The graphics and lowercase symbols are used for decorative purposes. The mode-changing characters, by contrast, have a definite effect in the way characters are displayed on the screen. For example, pressing the CLR/HOME key generates the special HOME character, which brings the flashing cursor to the top left of the screen. Pressing SHIFT-CLR/HOME clears the screen before moving the cursor to the home position. Other mode-changing characters include CTRL-1 to CTRL-8 and Commodore-1 to Commodore-8, which control the colors of all characters subsequently displayed. Others are CTRL-9 (RVS ON), which causes all subse-

quently displayed characters to appear in reverse image (switches the foreground and background colors), and CTRL-0 (RVS OFF), which cancels the action of CTRL-9.

It is unfortunate that all these mode-changing characters take effect immediately after being typed. If they did not, you could include them in a program for subsequent activation. But if you type, for example,

```
10
```

and then press the SHIFT-CLR/HOME combination, all that happens is that the screen clears before line 10 can be entered. The solution is to include the desired character in a PRINT statement. When an opening quotation sign is typed, indicating the start of a literal, the Commodore 64 automatically defers action until the literal is printed. For example,

```
A$="
```

followed by the SHIFT-CLR/HOME combination does not instantly clear the screen but instead displays on the screen the character representing CLR (an inverse heart). Typing the end quote and then pressing RETURN has the effect of setting A\$ to the character that clears the screen and sends the cursor to the home position. The screen is not affected, however, until the value in A\$ is printed out. The general principle may be observed in the following program, which clears the screen and prints a row of asterisks diagonally down the display. Immediately after the screen is filled, it is cleared and the same action takes place in the other diagonal direction. This process continues indefinitely.

PROGRAM 10-1

```
100 PRINT "*"
110 PRINT "*"
120 PRINT "*" ILLUSTRATION OF CLEARING SCREEN
130 PRINT "*"
140 PRINT "*"

```

```

150 PRINT
160 PRINT " ♥ "
170 FOR I = 1 TO 24
180 PRINT TAB(I); "*****"
190 NEXT I
200 PRINT " ♥ "
210 FOR I = 24 TO 1 STEP -1
220 PRINT TAB(I); "*****"
230 NEXT I
240 GOTO 160

```

Each of the characters on the Commodore 64 is represented internally (as already mentioned) by the ASCII code, which ranges from zero to 255. Since it is often easier to manipulate data in the form of numbers, two built-in functions exist to translate between the two representations. The first function, ASC, takes a string and converts it to its numeric form. You will remember that the statement

```
PRINT ASC("A")
```

displays the value 65, which is the Commodore's way of storing the character A. If the argument of the ASC function is longer than one character, only the first character is converted. The statement

```
PRINT ASC("CONTEMPLATION")
```

displays the value 67 because the internal representation of the letter C (the first letter of the literal) is 67. Also recall that the complementary function to ASC is the CHR\$ function. It converts the numeric representation of a character to the character itself. For example, the statement,

```
PRINT CHR$(66)
```

displays the character B.

In the following program some of the printable characters available on the Commodore 64 are displayed together with their numeric codes. The codes representing the characters displayed range

from 160 to 255. After each character is printed the computer waits until a key is pressed, which gives the viewer a chance to examine the character set. For fast viewing, the user may hold down the space bar (or any other repeating key, for that matter).

PROGRAM 10-2

```

100 PRINT "*****"
110 PRINT "*"
120 PRINT "          THE CHARACTER CODES"
130 PRINT "*"
140 PRINT "*****"
150 PRINT
160 FOR I = 160 TO 255
170 PRINT I,CHR$(I)
180 GET X$:IF X$ = "" THEN 180
190 NEXT I

```

The Commodore 64's ability to display fascinating color is illustrated in the next program, where some interesting patterns are generated by printing two given characters of the character set in the complete spectrum of colors. The first line of the program clears the screen, as character 147 is none other than the CLR character. This may be verified by typing the instruction

```
PRINT ASC(")
```

followed by the SHIFT-CLR/HOME combination, the end quote, and the final parenthesis. Thus, when you print the character whose code number is 147, you are actually displaying the character for clearing the screen. After the screen is cleared and the standard heading printed out, the user is asked to enter two character codes, which are stored in the variables A and B. After initializing A\$ to the null string, the program proceeds to concatenate the characters represented by the codes A and B into TEMP\$. Twenty copies of TEMP\$ are then copied end-to-end into A\$, and B\$ is set to A\$ shifted over by one position. At this point, alternating lines of A\$ and B\$ are displayed in the various colors available on the Commo-

dore, filling the screen with triangles in a pleasing pattern. To view the pattern in a more leisurely manner, the user holds down the control key. The DATA statement contains the equivalent codes to the characters controlling the color.

PROGRAM 10-3

```

100 PRINT CHR$(147)
110 PRINT "*****"
120 PRINT "*"
130 PRINT "      GRAPHIC DESIGN GENERATOR      *"
140 PRINT "*"
150 PRINT "*****"
160 PRINT
170 INPUT "ENTER TWO NUMERIC CHARACTER CODES: ";
    A,B
180 A$ = "" : TEMP$ = CHR$(A) + CHR$(B) : FOR I = 1
    TO 20 : A$ = A$ + TEMP$ : NEXT I
190 B$ = MID$(A$,2) + LEFT$(A$,1)
200 RESTORE
210 FOR I = 1 TO 16
220 READ COL
230 IF I / 2 = INT( I / 2) THEN PRINT CHR$(18) ;
    CHR$(COL) ; A$ ; : GOTO 250
240 PRINT CHR$(18) ; CHR$(COL) ; B$ ;
250 NEXT I
260 GOTO 200
270 DATA 144,15,28,159,156,30,31,158,129,149,150,
    151,152,153,154,155

```

(We recommend that you try character codes 169 and 223 for the first values for A and B, as they are particularly pleasing to the eye.)

There is virtually no end to the variety of the combinations possible by using different characters and colors. The following program, for example, displays a multicolored pattern by printing triangles with RVS alternately ON and OFF—by printing CHR\$(18) and CHR\$(146) respectively—with the colors changing randomly and continuously. The strategy employed this time is reading the color codes from a DATA statement into the array COL.

Only 15 (rather than 16) are read into the array because color number 7 (which is omitted) is the background color and so creates a "hole" in the sequence of patterns.

PROGRAM 10-4

```

100 PRINT CHR$(147)
110 PRINT "*****"
120 PRINT "*"
130 PRINT "      ANOTHER INTERESTING PATTERN"
140 PRINT "*"
150 PRINT "*****"
160 PRINT
170 INPUT "ENTER TWO CHARACTER CODES: ";C1,C2
180 DIM COL(15)
190 FOR I = 1 TO 15:READ COL(I):NEXT I
200 PRINT CHR$(18)+CHR$(C1)+CHR$(146)+CHR$(C2)
      +CHR$(COL(INT(RND(1)*15)+1));
210 PRINT CHR$(18)+CHR$(C2)+CHR$(146)+CHR$(C1)
      +CHR$(COL(INT(RND(1)*15)+1));
220 GOTO 200
230 DATA 144,5,28,159,156,30,31,158,129,149,150,
      151,152,153,154,155

```

Direct Machine Access with PEEK and POKE

Extensive as Commodore BASIC is, there are many features of the machine that are not supported by it. Many of the advanced features such as sprite control, changing the character set, and sound generation can be effected only by special commands that are able to access selectively parts of the memory that BASIC ordinarily is not able to reach. Unfortunately, the scope of this book prevents us from entering into this area in detail. Indeed, a sizable work could be written solely on this point.

As was mentioned previously, the Commodore 64 has 65,536 distinct memory locations, each of which is capable of holding a number from 0 to 255. By using the POKE command in BASIC any one of these memory locations may be set to any legal value. In

addition, any one of these locations may be examined by use of the PEEK function. For example,

```
POKE 5000,3
```

followed by

```
PRINT PEEK(5000)
```

displays the value 3 that was stored in location 5000 by the POKE command. Certain locations (particularly those in the high areas of memory—above 50,000) create special effects. By taking advantage of these features it is possible to exploit the computer to its fullest.

One particular instance in which POKE allows for manipulation of the machine in a way that is impossible otherwise is changing the background color of the screen to any specified color. The command needed is

```
POKE 53281,N
```

where N is the color code ranging from 0 to 15.

Sound Generation

Even though the material needed to generate sound is somewhat more involved than any yet encountered, nevertheless, the incredible excitement that can be incorporated into programs—especially games—by taking advantage of this feature makes mastering it worth the effort. Moreover, it is not necessary to have any previous knowledge of music to create some very interesting sounds on the Commodore 64.


The Commodore sound chip (called the “SID,” for Sound Interface Device) is actually a sophisticated peripheral processor, that is, a computer in its own right. The SID is connected to the heart of the Commodore 64 by the 25 locations from 54,272 to 54,296. It is by manipulating the contents of these addresses that the spectacular

variety of effects are accomplished. As you will have guessed, in BASIC this can be done only through POKEs.


The sounds we hear (no matter how complex they may be) are really combinations of vibrations impinging on the ear. There are, however, relatively pure vibrations which we recognize as notes. These notes have specific frequencies, which are the first values that must be set when creating a sound on the Commodore 64. The whole Western system of music is based on ratios between these frequencies.

In order to create a sound, it is necessary to specify some kind of waveform, the basic vibrational energy which more complicated parameters shape into the final sound. The SID chip allows for four distinct waveforms:

White noise (for sound effects)

Pulse (a square wave) 

Sawtooth 

Triangular 

Two other important parameters are the attack/decay and sustain/release values, which determine what is referred to as the “envelope” of the wave that makes up the tone. When one of the four simple waveform types is emitted, it starts out quietly and rises in volume to a maximum predetermined intensity. The rate at which this process occurs is known as the *attack*. It then dies down from its peak volume to some lower volume. The fall rate is known as the *decay*. The lower volume to which the note drops is called the *sustain* level. Finally, when the note stops, it falls from the sustain level to zero volume, at a rate which is called the *release*.

Using these four values, the timbre, or musical quality, of the note may be controlled. The difference between a piano, an electric guitar, and a clarinet is entirely due to the shape of the waveforms they emit. By properly modifying the ADSR and other envelope parameters, it is possible to simulate a wide variety of musical

sounds. The waveform also affects the sound, but in a more subtle way; the envelope changes the general characteristics of the tone—sharp, gradual, and so on—whereas the waveform makes the sound more like the pluck of a harp string or a piano—slight changes in the musical quality of the tone.

When generating sound on the Commodore 64, the first action is to erase all previous values from the SID chip so that any residual values there will not affect the generated tone. This step is accomplished by setting locations 54272 through 54296 to zero by the following line:

```
160 FOR I = 54272 TO 54296:POKE I,0:NEXT
  I
```

Next, set the desired ADSR values. Each of these is a number from 0 to 15, where 15 is the highest legal value and 0 is the lowest. These values may be set by the following statements:

```
170 POKE 54277, A * 16 + D
180 POKE 54278, S * 16 + R
```

where the variables A, D, S, and R contain the ADSR values (0 through 15). Next, the volume (the overall loudness of the emitted sound) must be set, again to a number between 0 and 15. This step is accomplished by the statement

```
190 POKE 54296,V
```

where V is the volume. Then the frequency must be set to a number between 0 Hz and 65,535 Hz. Since it cannot all fit into a single byte, it is divided into high- and low-order portions. If the frequency is stored in the variable F, the statement

```
200 HF=INT(F / 256):POKE 54273, HF:POKE
  54272, F - HF * 256
```

sets both portions correctly. (HF is an arbitrary variable name that stands for the high part of the frequency. The low part of the fre-

quency is computed by using the values of F and HF.) Finally, the waveform must be set. This value may be calculated by looking up the corresponding value in the following table. Note that two waveforms may not be combined. The result which is obtained by adding their corresponding values does not sound very pleasant at all.

Waveform	Value
White noise	128
Square	64
Sawtooth	32
Triangular	16

A value is entered by specifying one of these waveforms by the statement

```
210 POKE 54276, W + 1
```

where the variable W contains the waveform value (one of the four numbers listed in the table). The 1 that is added at the end of the statement is included to trigger the start of the A/D phase. After a user-defined delay (which is usually accomplished with a delay loop, where the length of the delay represents the duration that the note is to be sustained);

```
220 FOR DELAY = 1 TO 50:NEXT DELAY
```

the release phase is begun by means of the statement

```
230 POKE 54276,W
```

The following program implements these features. It requests a frequency from the user, proceeds to play the note in the form of a triangular wave, and then goes back to request another note in an infinite loop. We recommend frequencies above 220, since anything lower might be inaudible. (Remember the upper limit is 65,535.)

PROGRAM 10-5

```

100 PRINT CHR$(147)
110 PRINT "*****"
120 PRINT "*"
130 PRINT "          NOTE GENERATION"
140 PRINT "*"
150 PRINT "*****"
160 PRINT
170 FOR I = 54272 TO 54296:POKE I,0:NEXT I
180 POKE 54277,0 * 16 + 9:REM A/D
190 POKE 54278,0 * 16 + 9:REM S/R
200 POKE 54296,15:REM MAXIMUM VOLUME
210 INPUT "ENTER THE DESIRED FREQUENCY:";F
220 HF = INT(F / 256):POKE 54273,HF:POKE 54272,
    F - HF * 256
230 POKE 54276,17
240 FOR T = 1 TO 50:NEXT T
250 POKE 54276,16
260 INPUT "ANY MORE";AGAIN$
270 IF AGAIN$ = "YES" THEN 210
280 POKE 54296,0

```

There are many more features associated with the SID chip. The Commodore 64 is truly a highly sophisticated sound generator. However, taking full advantage of its versatility requires a great deal more time and effort and programming difficulty. As you can see from the preceding example, which merely sounds out a single note, a more thorough treatment of the subject is well beyond the scope of this book. Still, even a limited amount of sound can infuse a program with excitement.

Here is another sound program, which simulates the sound of a jet plane.

PROGRAM 10-6

```

100 PRINT CHR$(147)
110 PRINT "*****"
120 PRINT "*"
130 PRINT "                JET"
140 PRINT "*"
150 PRINT "*****"
160 PRINT
170 S = 54272:Q = .97
180 FOR I = S TO S + 24:POKE I,0:NEXT I
190 POKE S + 5,89:POKE S+6,240
200 POKE S+24,15
210 POKE S + 4, 129
220 INPUT "ENTER YOUR HIGH AND LOW FREQUENCY:";
    HI,L0
230 IF HI < L0 THEN TEMP = HI:HI = L0:L0 = TEMP
240 F = HI
250 HF = INT(F / 256):LF = F - HF * 256
260 POKE S+1,HF:POKE S,LF
270 F = F * Q
280 IF F < L0 OR F > HI THEN Q = 1/Q:FOR T = 1
    TO 50:NEXT T
290 GOTO 250

```

White Noise

This program is very similar to the previous one in that it produces sounds by using most of the same POKEs. However, it differs, first, because it does not stop after one tone but sweeps a range between two specified frequencies in an infinite loop. Second, it produces the chaotic roar of a jet plane overhead rather than a pure musical tone; this sound is obtained by setting the waveform to what is known as “white noise” rather than a pure tone. White noise is analogous to white light—which is a mixture of all the colors of the spectrum. In a similar manner, white noise is a mixture of many random frequencies, which combine to sound like a “hissing” noise. The hiss can be turned into an explosive sound by setting the attack and decay values for very sharp rise and fall (15 for each).

Multiple Voices

The Commodore 64 supports more than just one voice at a time. In fact, it permits three different simultaneous sounds, each with its own ADSR, frequency, waveform, and other values. The locations that are POKEd for the additional voices are quite similar; just add seven to each location, as will be observed from the following table. The one exception, you will note, is the volume location, which is common to each of the voices. Therefore, the intensity of the tone for each of the voices is the same; all that can be controlled is the general loudness of the sound produced—not the mix.

Some of the Most Popular Sound Memory Locations

Location	Explanation
54272/3	Frequency of tone to be produced.
54274/5	Specifies the shape of the wave if it's of the pulse (square) type.
54276	Specifies (among other things) the type of the waveform and switches between the A, D, and S phases of the envelope and the R.
54277/8	Specifies the A, D, S, and R.
54296	Sets the volume.

Review Questions

- How are the righthand graphics symbols on most of the keys obtained?
By holding down SHIFT while pressing the appropriate key.
- How are the lefthand graphics symbols accessed?
By holding down the Commodore key and the required key.
- What is the effect of the instruction PRINT CHR\$(147)?
It clears the screen.

232 ■ AT HOME WITH BASIC

- 4 ■ What is the effect of the PEEK function?

It returns the value of the specified memory location.

- 5 ■ What is the purpose of the POKE command?

It sets the value of a given memory location to a specified value from 0 to 255.

- 6 ■ Given the statements

```
POKE 2048,59  
PRINT PEEK(2048)
```

what (if anything) is displayed on the screen?

The number 59.

- 7 ■ Given the program segment

```
POKE 2055,96  
PRINT PEEK(5502)
```

what is printed?

There is not enough information given to determine what would be printed; it depends on the value stored in location 5502 at the time.

- 8 ■ What is the effect of the statement

```
POKE 53281,1
```

The background color of the screen changes to white.

- 9 ■ What range of locations controls the sound capabilities of the Commodore 64?

Locations 54272 to 54296.

- 10 ■ What do the letters ADSR stand for?

Attack (the rise rate), decay (the fall rate), sustain (the length of time the note takes before it begins to trail off), and release (the rate of the trail-off).

- 11 ■ How is the SID chip cleared?

By setting all the locations in the range 54272 to 54296 to 0.

- 12 ■ What is white noise?

White noise is a random mixture of frequencies.

- 13 ■ How many voices are permitted on the Commodore 64?

A maximum of three voices are permitted.

TRY YOUR HAND AT THESE

1. Write a program that plays a concert A (frequency 440).
2. Write a program that plays an A in four different octaves. (Hint: A note one octave above another has a frequency of twice the lower octave.)
3. Write a program to simulate a wailing siren.
4. Write a program to take a message typed in English, convert it to international Morse code, and sound it out. The standard to be used is that a dash is three times as long as a dot.
5. Using the white-noise generator, devise a program to simulate the sound of a machine gun.

Debugging

In computer programming it is a universal truth that no program worthy of the name runs the way one expects it to the first time, no matter how experienced the programmer. What distinguishes the good programmer from the bad, however, is the approach adopted in finding and correcting the errors. The proper way to go about eradicating errors is to expect the program to contain them at the outset and so be prepared for them when they arise.

The aim of this chapter is to give you the tools to prevent some of the most common errors from arising in the first place, and when they occur, (rather than should they occur), to help find and eliminate them. Among programmers, all errors are known as *bugs* and the process of getting rid of them is called *debugging*.

There are two major types of errors. The first falls under the broad category of syntax errors. The second group involves errors of logic. Syntax errors are by far the easiest to detect. In fact, the computer does it for you automatically by printing a diagnostic message advising you of the nature of the error and pinpointing the offending line. All the programmer need do is understand the nature of the error and correct it. Errors of logic, on the other hand, cannot be detected by the computer and are sometimes quite difficult to track down. They are often caused by misconceptions of the problem and unforeseen special cases.

Syntax Errors

Just like natural languages, BASIC adheres to grammatical rules. In this section we will explore the most frequently encountered syntax errors. One of the commonest types is the mistyping of a keyword, for example, PRUNT for PRINT. This error is detected by the system during execution of the program and not when the line is entered. If the offending statement is in line 20 the message

```
?SYNTAX ERROR IN 20
```

appears. At this point, the proper procedure is to LIST the offending line by typing in

```
LIST 20
```

After the offending line has been scrutinized and the error found, the line is modified, after which the program can be run again. Some of the most common syntax errors occur repeatedly in a program. Misspelling of keywords is probably the most common, followed closely by variable names containing keywords, some of which might be unfamiliar to you and are thus difficult to detect. Some innocent-looking though invalid variable names follow:

Variable Name	Keyword Included
WIDGET	GET
LENGTH	LEN
BINGO	GO
FORGOTTEN	FOR, GO
STINGY	ST (An abbreviation for STATUS)
STUFF	ST
MORE	OR
COMMAND	AND
FRIEND	END
PHOSPHORUS	OR
TONE	TO
FOREIGN	FOR

Variable Name	Keyword Included
PALLETTE	LET
DIME	DIM
TREAD	READ
SPOON	ON
TOGETHER	TO, GET
THOROUGH	OR
NOTHING	NOT
SHIFT	IF
FORGET	FOR, GET
DEFINITION	DEF, ON

Now that you have been made aware of this feature, can you see any problems that might be caused by the variable name “OPPOSITION”? It happens to contain the keyword “ON” and so is invalid.

A more obscure type of syntax error is an arithmetic expression in which the parentheses are not balanced. The Commodore 64 simply does not tell you that the parentheses are unbalanced. Instead, it rudely gives the standard ?SYNTAX ERROR message, leaving the bewildered programmer to wonder what is wrong. An example of this type of error follows:

$$A = (3 * 1) / (2 / 5 + 16 \uparrow (3 - 2))$$

Although a quick glance at this statement might not reveal anything wrong, a closer scrutiny will show that there are three left parentheses, and only two right ones. Of course, just having an equal number of opening and closing parentheses is no guarantee that the expression is acceptable. The next example does, in fact, contain two left and two right parentheses. Since they face the wrong direction, however, the computer flags it as a syntax error.

$$B =)3 + 2(* (4 + 5)$$

Mathematical expressions must also be written in a certain way, even if other ways are valid in “normal” math. For example, although implied multiplication is perfectly valid in algebra and ordi-

nary arithmetic, any attempt to use implied multiplication in a BASIC statement will lead to problems. Thus, although

$$x = 5a + 6bc$$

is a fine example of an algebraic expression, it cannot be written in BASIC in that form. Instead, it must be written as

$$X = 5 * A + 6 * B * C$$

Errors of Logic

Once all the syntax errors have been corrected, some programs still stubbornly refuse to run to completion. A typical example follows.

PROGRAM 11-1

```
10 FOR I = -5 TO 5
20 PRINT I/I
30 NEXT I
RUN
1
1
1
1
1
1
?DIVISION BY ZERO ERROR IN 20
READY.
```

This kind of error can be infuriating because the program seems to work for some values but not for others. As usual, the offending line should be listed out. It reads

```
20 PRINT I / I
```

By taking advantage of the information contained in the error message, it becomes obvious that the program must be dividing by zero,

which is an illegal operation. At this point, the value of I can be printed out in immediate mode. It will then be seen that its value indeed is zero. Upon examination of the loop, it becomes clear that its index, I, does indeed take on the value of 0 when going from -5 to 5 and hence, division of 0 by 0 is performed. Now that the nature of the problem is understood, it remains for the programmer to correct the error. One way would be to include an IF statement before the calculation of I / I is performed. For example,

```
15 IF I = 0 THEN PRINT "VALUE
    UNDEFINED":GOTO 30
```

which avoids the special case of division by zero.

Another common type of error, but fortunately one that is easily detected, is the ?TYPE MISMATCH error. It is caused by assigning a value of one type to a variable of a different type, for example, in the following lines of code:

```
COM# = 426
MOD = "DORE"
W(REF#) = 6
FOR INDEX# = 1 TO 5
```

Examine the following program and try to determine if it will run without difficulty.

PROGRAM 11-2

```
10 DIM X(100)
20 FOR I = 1 TO 100:X(I) = INT(RND(1) * 100)
    + 1:NEXT I
30 FOR J = 1 TO 100
40 NEXT J
50 PRINT X(J)
```

Even though the array X is generated by the index I, when printing it out there is no reason at all to stay resolutely with the index I. Using the index J is perfectly acceptable, as long as J ranges from 0

to 100 (the dimensioned size of X) In this case, however, X(J) is printed outside of the FOR . . . NEXT loop. In such a situation, the value of the index J is 101, 1 more than the dimensioned value, the reason being that a FOR . . . NEXT loop behaves identically to the following section of code:

```

10 J = 1
   [body of loop]
100 J = J + 1
110 IF J <= 100 THEN 20

```

The only way this loop is exited is when J becomes equal to 101. In a similar fashion, the index of a FOR . . . NEXT loop that has been “satisfied” will always be 1 greater than the ending value specified in the FOR statement. Therefore, an error is generated when the attempt is made to print out the value of X(101), which is nonexistent. The obvious correction to make here is to place the PRINT instruction inside the loop, thereby performing what was originally desired, namely, the printing out of each of the 100 elements of the array, one beneath the other.

A much more subtle error is the kind that does not generate a diagnostic message at all. For example, if instead of writing

```
10 PRINT "HI THERE"
```

you omitted the quotation marks entirely, the variable HITHERE would be printed. In the assumption that it had never been assigned a value, the number printed out would be 0.

Another common mishap involving the omission of quotation signs in literals is forgetting the trailing quotation sign. Any further commands on that line become part of the literal and, as a result, are not executed as intended. For example,

```
20 PRINT "HELL0:IF X > Y THEN GOTO 50
```

is treated by the computer as the literal

```
"HELLO:IF X > Y THEN GOTO 50"
```

In the event that the programmer forgets the trailing quotation sign it is supplied automatically by the system and no error message is displayed.

When dealing with loops, certain errors repeatedly crop up. For example, consider the following program:

PROGRAM 11-3

```
10 DIM X(100)
20 FOR I = 1 TO 100:X(I) = I:NEXT I
30 PRINT X(J)
40 FOR J = 1 TO 100
50 NEXT J
```

Here the intention is to print out all 100 elements of the array X. Notice, however, that the PRINT instruction is located outside the loop rather than within. The subscript J is therefore not defined and defaults to the value 0. Printing out X(0) (which is undefined) produces the value 0. As a result, 0 is displayed by the PRINT statement. You will notice that this particular error, although similar to the one made in Program 11-2, does not generate an error message, whereas Program 11-2 did. In practice, this type of error is much harder to detect because the program does generate results. Compare this situation with the following case:

PROGRAM 11-4

```
10 DIM X(100)
20 FOR I = 1 TO 100:X(I) = I:NEXT I
30 FOR J = 1 TO 100
40 PRINT X(P)
50 NEXT J
```

Here the value of P (never having been defined) is zero. Line 40, which prints X(P), is then equivalent to PRINT X(0). Since X(0) has also never been set, it too is treated as zero, and as a result, 100 zeros are printed out.

Sometimes a nest of loops is set up with identical index names. This error is not actually detected by Commodore BASIC except during execution, when the second NEXT is reached. The error message: ?NEXT WITHOUT FOR is displayed. The following program would create such an error.

PROGRAM 11-5

```

10 FOR I = 1 TO 5
20 FOR I = 1 TO 10
30 PRINT I
40 NEXT I
50 NEXT I

```

When the first FOR I is encountered in line 10, I is initially set to 1. Then the inner loop is entered and again I is set to 1. This in itself does not violate any rule of BASIC. At this point, however, all record of the original I is lost. When the inner loop is satisfied and control drops to line 50, the system detects what it regards as a NEXT I without its corresponding FOR statement, generating the error message quoted.

A more subtle error is engendered when two variables, both of which begin with the same two characters, are used in a program. For example,

PROGRAM 11-6

```

10 WAGES1 = 10
20 WAGES2 = 20
30 PRINT WAGES1,WAGES2

```

prints the value 20 for each of the variables, WAGES1 and WAGES2. Since both variable names begin with the letters WA, they are considered to be identical. Therefore, it is tantamount to writing the program as

```

10 WA = 10
20 WA = 20
30 PRINT WA,WA

```

In this form, it becomes obvious that the value 10 is displaced by the second assignment statement. The value of WA becomes 20 and is printed twice. Watch out for this one—it is very, very subtle and can be very hard to trace.

Trial Runs

Even when a program seems to be running smoothly (the surest sign that something important has been overlooked, according to one of the many versions of Murphy's Law) and results are printing out, it is no guarantee whatever that the results are, in fact, correct. Suppose, for example, you wish to calculate the average of two user-supplied numbers A and B and have written the following program to do it:

PROGRAM 11-7

```
10 INPUT A,B
20 AVERAGE = A + B / 2
30 PRINT A,B,AVERAGE
```

This program prints the values of A and B, followed by not the average of A and B but rather the value of $A + (B / 2)$, which is definitely not the average. This error is caused by the algebraic order of operations, in which, you will remember, division takes precedence over addition. The correct way to write line 20 is

```
20 AVERAGE = (A + B) / 2
```

In a more complex situation, the only way such an error can be detected is by comparing the printed results with some previously calculated answers. The program should be trusted only after it has passed a rigorous series of tests.

Sometimes a program will work perfectly for some data but not for others. The following program, for example, is designed to solve quadratic equations.

PROGRAM 11-8

```

10 PRINT "TYPE IN THE VALUES OF A, B AND C"
20 INPUT A,B,C
30 X1 = (-B + SQR(B ↑ 2 - 4 * A * C)) / (2 * A)
40 X2 = (-B - SQR(B ↑ 2 - 4 * A * C)) / (2 * A)
50 PRINT "ROOT1 =" ; X1
60 PRINT "ROOT2 =" ; X2

```

SAMPLE OUTPUT RUN 1

```

TYPE IN YOUR VALUES OF A, B, AND C
? 1,2,1
ROOT1 = -1
ROOT2 = -1
READY.

```

SAMPLE OUTPUT RUN 2

```

TYPE IN YOUR VALUES OF A, B, AND C
? 1,2,5
? ILLEGAL QUANTITY ERROR
READY.

```

The error created in the second run is caused by the fact that for the values chosen for A, B, and C, the discriminant— $B^2 - 4 * A * C$ —is negative. Trying to take the square root of a negative quantity causes the diagnostic message to be displayed and the run terminated immediately. The correct approach is to define the discriminant separately and to test it before taking its square root. If the discriminant is a negative quantity, the square root cannot ordinarily be calculated. Only if it is equal to or greater than zero should the square-root function be applied. A correct program would be of the following form.

PROGRAM 11-9

```

10 PRINT "TYPE IN YOUR VALUES FOR A, B AND C"
20 INPUT A,B,C

```



```

30 DISC = B ↑ 2 - 4 * A * C
40 IF DISC < 0 THEN PRINT "ROOTS ARE
   IMAGINARY":END
50 X1 = (-B + SQR(DISC)) / (2 * A)
60 X2 = (-B - SQR(DISC)) / (2 * A)
70 PRINT "ROOT1 =" ; X1
80 PRINT "ROOT2 =" ; X2

```

There will almost always be data that will cause a hastily or carelessly written program to “bomb.” There is only one sure way to be convinced that the program will always work as designed and that is to test it out with every conceivable type of data. Unfortunately, this is impossible because there are usually far too many possibilities to test. Therefore, you have to settle for the most reasonable type of errors and construct test data accordingly. The best way to do so is to try deliberately to concoct data that will stop your program from running and counter every possibility you have thought of. Even so, experience has shown that some errors will remain, no matter how diligent you are, since we all have our own preconceived notions of approaching a problem. For close to absolute certainty many different people must test the program over a considerable period of time.

An excellent way to minimize debugging time and the creation of errors is to adopt what has come to be known as “defensive” programming. Once you are resigned to the inevitability of making mistakes, you may insert instructions to print out intermediate results at critical points. This step helps to localize any errors found to within a few lines of code. After the program has been checked out, the print instructions can be deleted.

Another strategy that works very well in conjunction with the two methods just described is to arrange error traps for the various types of erroneous input. If a program is designed to accept numbers in the range 1 through 100, you should set up a “filter” or “trap” to ensure that no input outside that range is accepted. By stopping bad values right at the source, you avoid a major source of error and headache.

Occasionally, the nature of an error will be so puzzling that no amount of systematic debugging helps to reveal its source. Even in

such cases, however, all is not lost. The best course for the programmer to follow in such situations is to simply “play computer.” By carrying out each of the programming instructions—using a pocket calculator if necessary—the programmer can see what goes wrong as it happens.

In this chapter, we have tried to give you the tools and techniques to find and eliminate bugs, if and when they occur. Even though the techniques may be known, good debugging requires a firm command of the language and a lot of experience, some of it quite bitter. Often, the programming novice tends to become discouraged all too quickly. A good practice is to stop work on the problem after a certain frustration level has been reached; it will do no good to continue past that point. Returning to the problem at some later time when you are more refreshed often helps. Another helpful hint is to ask a friend to take a look at your program, even if he or she is not particularly adept at programming. It is usually easier for someone else to spot your elementary mistake than it is for you, in the same way that it is often easier to give advice than to receive that very same advice yourself.



Glossary

- Access mode:** A technique used to obtain a specific logical record from, or put a logical record into, a file.
- Access time:** The length of time it takes for information to be written to or read from a diskette.
- Accoustic coupler:** A special type of modem which allows a standard telephone headset to be attached to a terminal or computer to allow for transmission of data.
- Accumulator:** A holding register in the computer's arithmetic logic unit that holds instructions for input/output operations. It performs arithmetic operations.
- Accuracy:** The quality of being free from error. On a machine this is actually measured and refers to the size of the error between the actual number and its value as stored in the machine.
- Address:** A number or name that identifies a particular location in memory, in a register, or in some other data source.
- Algorithm:** A finite set of well-defined rules for the solution of a problem in a finite number of steps.
- Allocate:** To assign a resource, such as a diskette file or a part of memory, to a specific task.
- Alphabetic:** A letter of the alphabet.
- Alphanumeric or Alphameric:** Data presented in both alphabetic and numeric form, such as in a mailing list. These data may contain the digits 0 through 9 and the letters A through Z in any combination.
- ALU:** Arithmetic Logic Unit.
- ANSI:** American National Standards Institute, an association of computer manufacturers and users whose purpose is to standardize computer languages.
- Applications software:** A program or group of programs written in high-level languages which perform specific tasks, such as word processing, general ledger and mailing list, and so on.
- Architecture:** The actual physical layout and construction of a microcomputer.
- Argument:** A value that is passed from a calling program to a function.
- Arithmetic expression:** An expression consisting only of numbers and operators, such as $4 + 7 * C$.
- Arithmetic logic unit:** The device within the CPU that performs all the arithmetic operations, such as addition, subtraction, multiplication, and division.

- Arithmetic operator:** A symbol that tells the computer to perform an arithmetic operation. The operators are + for addition, - for subtraction, * for multiplication, / for division and ↑ for exponentiation (raising a number to a power).
- Arithmetic overflow:** When the result of an operation exceeds the capacity of the intended unit of storage.
- Arithmetic variable:** A location in memory where a numeric variable is stored.
- Array:** A one-dimensional set of elements arranged in tabular form.
- ASCII:** A simple code system that converts symbols and numbers into numeric values the computer can understand. For example, when uppercase A is typed in at the keyboard it is converted to the number 0010001 before being sent to the CPU. (The binary number 0010001 is equivalent to the decimal number 17.) The acronym (pronounced *askey*) stands for American Standard Code for Information Interchange.
- Assembly language:** A programming language that uses mnemonic symbols. An assembler converts the mnemonics into machine language.
- Background:** The area which surrounds the subject; in particular, the part of the display screen surrounding a character.
- Backup:** A copy of any program or other information stored on a cassette or disk. Backups usually are stored in a safe place and are used if a bug develops in the original. Cassettes and disks tend to wear out with frequent use.
- BASIC:** An acronym for Beginner's All-purpose Symbolic Instruction Code, a high-level computer language designed for beginners. The most common microcomputer language, it was developed at Dartmouth College by Dr. Thomas Kurtz and Dr. John Kemeny in 1965.
- Batch processing:** The breaking down of items or jobs into different groups (batches) which are processed together at one time—for example, in a payroll system, instructing the computer to print paychecks for all administrative personnel at one time.
- Baud:** A unit of information transfer. In microcomputers, the baud is defined as bits per second.
- Baud rate:** The rate at which information is transferred from one computer to another over telephone lines or cables. When telephone lines are used, most computers transfer information at the rate of 300 baud (300 bits per second), or about 37 characters a second.
- Binary number:** A number system with the base 2 that uses only two digits, 0 and 1, to express all numeric values.
- Bit:** The basic unit of computer memory. It is short for *binary digit* and can take on a value of either 0 or 1. Computers "think" in bits. Each number and letter that goes into a computer is translated into a unique series of electronic impulses. Each impulse, actually a level of voltage

coursing through the computer's circuitry, is usually represented on paper by a 0 or 1. Each combination of bits, representing a letter or number, is called a *byte*. There are eight bits in a byte.

Blank: A part of a data medium in which no characters are recorded. Also, the space character.

Boolean value: A numeric value that is interpreted as "true" (if it is not zero) or "false" (if it is true).

Boot: This is the initialization program that sets up the computer when it is turned on.

Bootstrap: An existing version, perhaps a primitive version, of a computer program that is used to establish another version of the program. It can be thought of as a program that loads itself.

Bps: Bits per second.

Branch: A program segment that tells the computer to skip certain line numbers and possibly return to them later.

Break: To interrupt execution of a program. The computer has a control key labeled "BREAK."

Bubble sort: A technique for sorting a list of items into sequence. Pairs of items are examined and exchanged if they are out of sequence. This process is repeated until the list is sorted.

Buffer: A temporary storage register used to hold data for further processing.

Bug: A problem that causes a program or the computer itself to malfunction.

Bus: A circuit or group of circuits which provides an electronic pathway between two or more microprocessors or input/output devices, such as a keyboard and a computer.

Byte: A group of eight bits (or a memory cell that can hold eight bits) usually treated as a single unit. It takes one byte to store each unit of information. For example, the word "TABLE" would require five bytes, one for each letter.

CAI: Computer Aided Instruction, the process of teaching by computer. This is a system of individualized instruction that uses a computer program as the teaching medium.

Call: To bring a computer program, a routine, or a subroutine into effect, usually by specifying the entry conditions and jumping to an entry point.

Cassette drive: A tape cassette machine designed for use with a computer. Cassettes are usually modified audio cassette tape recorders.

Cathode Ray Tube: The picture tube of a television set is a CRT. A television or a monitor is often used to display computer output.

Central Processing Unit: This is the heart of the computer. It contains the circuits that control the interpretation and execution of instructions.

Character: Any letter, number, or other basic unit of communication

produced by a computer. Normally, one byte represents one character in a personal computer's language.

Character printer: Any device that prints one character at a time in a series across a page, just like a standard typewriter.

Chip: The building block of a computer. Chips with different functions are delicately wired together and arranged, usually, on a small piece of silicon or other semiconductor material.

Clock: A device that generates periodic signals to synchronize the computer's operations. Each signal is called a *clock pulse* or *clock tick*.

COBOL: An acronym for COmmon Business Oriented Language, a high-level language generally used with medium-sized or large computers for business problems.

Code: (1) A system of symbols and rules for representing the transmittal and storage of information. (2) Program instructions.

Command: An instruction that tells the computer to perform an operation immediately.

Comment: A statement used to document a program. Comments include information that may be helpful in running the program or reviewing the output listing.

Compatibility: Any computer's ability to interconnect directly with another device without special equipment, programs, or codes.

Compiler: A computer program that translates high-level language statements into machine language.

Compression: Arranging data so that they take up a minimal amount of space.

Computer: Technically speaking, the computer comprises only the CPU and the board and the chips connected to it. But when people speak of a computer, they usually mean to include the other components, like disk drives, a monitor, and a keyboard. What was once a room-sized assembly of 19,000 vacuum tubes has, through technological development since the 1950s, shrunk to the size of a fingernail. The thousands of microscopic electronic circuits, crowded into a space less than 1/8-inch square, became known as a *microprocessor*. It is, in fact, an electronic device that can receive and follow instructions and then use these instructions to perform calculations or compile, select, or correlate data. The primary difference between a computer and a calculator is that a computer can manipulate text, display graphics and make decisions.

Computer Assisted Instruction (CAI): A teaching method in which a computer is used to help students learn. Various CAI methods include drill and skill programs, simulations, and computer literacy exercises.

Computer language: The specific syntax, vocabulary, and grammar through which a computer can be instructed to carry out various functions and operations.

- Computer literacy:** The knowledge of the fundamentals of computer operations and the rudiments of computer programming. A person with this knowledge is said to be computer literate.
- Concatenation:** The operation that joins two strings together in the order specified, forming a single string with a length equal to the sum of the lengths of the two strings.
- Console:** The equipment used for communication between the operator and the computer.
- Constant:** A fixed value or data item.
- Control character:** A character whose occurrence in a particular context initiates, modifies, or stops a control operation. A control operation is an action that affects the recording, processing, transmission, or interpretation of data; for example, carriage return, font change, or end of transmission.
- Control key:** A special key that is used in conjunction with another key causes the computer to perform special functions. For example, pressing the control key in conjunction with the up arrow key causes the cursor to move up.
- Controller:** A device that can be attached directly to the computer or to an external mechanical device so that objects on the screen can be moved around. Joysticks and game paddles are both controllers.
- CPU:** Central Processing Unit. This is the "brain" of the computer and is located on a chip which is called the microprocessor, or the CPU. It is an electronic "traffic cop," handling all the information that comes into the computer from the keyboard or the disk drives and then sending it back to the monitor or to the disk drives. It performs all calculations. The CPU of the IBM PC is an 8088 chip and is manufactured by the Intel Corp. Apple II computers run on an entirely different chip, the 6502, made by MOS Technology. The Commodore 64 uses the 6510 (similar to the 6502.) An often-used microprocessor is the Z80, made by Zilog. Almost all microprocessors have CPUs that work with information either 8 bits or 16 bits at a time. Accordingly, they are called 8-bit or 16-bit machines. Generally, 16-bit are faster than 8-bit computers. The Commodore 64 and Apple IIe are 8-bit machines, whereas the IBM PC microcomputer has a 16-bit CPU.
- Cursor:** The little flashing symbol (usually a square) on the screen that indicates where the next character will be displayed.
- Daisy wheel printer:** A printing machine whose moving head has a number (usually 96) of radial arms or petals with a type character at the end of each.
- Data:** Any kind of information composed of letters, numbers, symbols, and so on, which can be processed by a computer.
- Data base:** Any collection of information, such as a list of accounts, ad-

dress lists, or newspaper stories. A data base management system enables a computer to store large amounts of information and then sort it in almost any manner. For example, a company's data base could give a list of customers by zip code, by credit line, alphabetically by name or by telephone number. The data base program takes care of managing the storage and retrieval of data.

Debug: To find and eliminate all errors in a program or a computer.

Default: A value or option that is assumed when none is specified by the programmer.

Delimiter: A character that groups or separates words or values in a line of input.

Diagnostic: Pertaining to the detection and isolation of a malfunction or mistake.

Digital computer: A computer that uses to represent information a series of electronic "on"s and "off"s, which are converted to (or from) binary numbers. Microcomputers are all digital computers, as opposed to analog computers.

Directory: A list of the file names stored on a disk.

Disk: A flat, rotating, circular sheet coated with magnetic material that is used to store bits of information in a compact form.

Disk drive: The most effective device for storing information, the disk drive is like a small record player. Its motor turns a circular piece of plastic, called a *disk*, which is similar in looks to a 45 rpm phonograph record. A disk drive has a magnetic head that moves across the disk, reading and writing data.

Diskette: A floppy disk that is 5 ¼ or 8 inches in diameter. It is encased in a stiff, square paper jacket for protection. Disks store not only information that the computer produces but also the instructions, called *programs*, that the computer needs in order to function. Not all disk drives are created equal. Different drives record different amounts of information on each disk. Some drives put information on both sides of a disk. These are called double-sided drives. Others, single-sided drives, record on only one side. Beyond that, some drives pack more information onto a disk than others. Drives with double-density capacity store much more data than do single-density drives. The density of a drive refers to the amount of data per track encoded on a disk. The tracks, unlike the spiraling grooves on a phonograph record, are arranged in concentric circles, like rings within rings. The most widely used microcomputer floppy disk is 5-¼ inches in diameter. But some drives use disks that are 8 inches, 3 inches, or 3-½ inches in diameter. The Commodore 64 uses 5-¼-inch floppy disks. Some disk drives are built into the compartment that houses the computer. Others, like those used by the Apple IIe microcomputer,

are housed in separate boxes and are connected to the computer by cables. In addition to floppy disk drives, there are hard disk drives, which rapidly spin hard, metal disks.

Documentation: All the available information about a particular computer, computer program, or set of programs. It usually contains operating instructions, troubleshooting hints, and so on.

Dot-matrix printer: A printer that forms characters as patterns of dots. The dots lie within a grid of definite dimensions, such as 7-by-9 dots.

Edit: To make changes in a program or data.

Element: A member of a set; in particular, an item in an array.

Enabled: A state of the processing unit that allows certain types of interruptions.

End of file (EOF): A "marker" immediately following the last record of a file, signaling the end of that file.

Execute: To run a computer program or part of a program.

File: An organized collection of related records. A payroll file would have a complete payroll record for each employee.

Flag: Any of various types of indicators used for identification, for example, a character that signals the occurrence of some condition.

Floppy disk: A flexible plastic disk coated with magnetic recording material on which computer data may be stored.

Flowchart: A graphical representation of the sequence of operations within an information system.

Folding: A technique for converting data to a desired form when it doesn't start out in that form. For example, lowercase letters may be folded to uppercase.

Formatting: The process of organizing a diskette into tracks and sectors so that the computer can access it.

FORTRAN: An acronym for FORMula TRANslation. It is a high-level computer language used for scientific and mathematical applications.

Function: A procedure that returns a value depending on the value of one or more independent variables in a specified way. More generally, the specific purpose of a thing or its characteristic action.

Function key: A keyboard key that tells the computer to perform a specific action.

GIGO: An acronym for Garbage In, Garbage Out.

Glitch: An error or problem in computer components or physical apparatus.

Graphics: The ability of a computer to show pictures, line drawings, special characters, and so on, on the CRT or printer.

Hard copy: A copy of the computer's output printed on paper or on some other permanent medium.

Hardware: All the physical components of a computer system, including the computer itself, the printer, and the monitor.

Hierarchy: A structure having several levels, arranged in a treelike form. "Hierarchy of operations" refers to the relative priority assigned to arithmetic or logical operations that must be performed.

High-level language: A computer language that uses simple English words to represent computer commands. For example, the command RUN in BASIC tells the computer to run (execute) a program.

Home computer: A microcomputer or personal computer. Its definition keeps changing as the price of computers keeps falling and machines become more powerful. It is characteristically defined by price—usually less than \$800. Computers with additional internal memory and storage capacity (costing somewhat more, of course) are known as personal computers. They are more powerful than home computers and are big enough for word processing, some financial planning, other serious work—and for playing games. Although they are called "personal," they are found in the offices of many companies. More expensive are business microcomputers, which generally have still more memory and storage capacity. They can use sophisticated software for financial analysis, data base management, and communication with large mainframe computers, the biggest computers made.

Housecleaning: The process by which BASIC compresses string space, collecting all its useful data, and frees up unused areas of memory that were once used for strings.

IC: Abbreviation for "Integrated Circuit," which holds in a plastic or ceramic case a tiny chip of semiconductor material. Thousands of transistors, capacitors, and other electronic components are fixed on each IC.

Impact printer: A type of printer that produces information on paper by actually striking a ribbon onto a sheet of paper, in the manner of a typewriter.

Implicit declaration: The establishment of a dimension for an array without it having been explicitly declared in a DIM statement.

Increment: A value used to alter a counter.

Initialize: To set counters, switches, addresses, or contents of memory to zero or other starting values at the beginning of, or at prescribed points in, the operation of computer routine.

Input: The data which are transferred from the keyboard, a diskette, or a cassette to the internal RAM memory.

Input device: A device used to enter information into the computer.

Input/Output: The process of entering data into or receiving data out of a computer.

Instruction: Properly coded information that causes the computer to perform certain operations.

Integer: One of the numbers 0, 1, 2, 3, . . . and their negative values.

Integrated circuit: A group of components that form a complete miniaturized electronic circuit consisting of a number of transistors plus associated circuits. These components are fabricated together on a single piece of semiconductor material, usually silicon.

Interactive: A computer system that responds immediately to user input.

Interface: A device that allows two devices to communicate with each other.

Interpreter: A program that translates a high-level language such as BASIC into a machine language so that it can be used in the computer. It is slower and less efficient than a compiler but much easier for programmers to use.

Interrupt: To stop a process in such a way that it can be resumed.

Inverse video: A process that allows you to show dark text on a light background on the CRT. Normally, light text is shown on a dark background.

Invoke: To activate a procedure at one of its entry points.

I/O: An abbreviation for “input/output.”

Jack: A plug socket on a computer.

Joystick controller: A box with a movable stick attached. When connected to the computer, motion of the stick causes objects on the screen to move around.

K: An abbreviation for “kilo,” a prefix meaning 1,000. Thus 4K of memory is about 4,000 bytes. More exactly, 1K is 1,024 bytes so 4K represents 4,096 bytes, but 4K is a convenient way of keeping track of it.

Keyword: One of the predefined words of a programming language; a reserved word.

Language: Any code that allows humans to communicate with a computer. Computer languages vary greatly in their complexity, starting with the native idiom of the CPU, which is machine language—the 1s and 0s of binary code. Next comes assembly language represented in letters and numbers that can be more easily used by people. Above those are “high-level” languages, like BASIC, FORTRAN, and COBOL. These higher-level languages use easily understood letters, words, and numbers and turn them into machine language for the computer’s use.

Line number: A number that defines a line of programming in a high-level language such as BASIC. Each line of the program begins with a line number. The computer executes the program in line number order, starting with the lowest number.

- Line printer:** Any printer that prints one line at a time rather than one character at a time.
- Literal:** An explicit representation of a value, especially a string value; a constant.
- Location:** Any place in which data may be stored.
- Logic:** A systemized interconnection of devices in a computer circuit that causes it to perform certain functions.
- Logo:** A high-level language designed at MIT for use in educational settings.
- Loop:** A series of programming instructions that recycle. The last instruction in the loop tells the computer to return to the first instruction. Intentional loops have some means of escape built into them. Unintentional loops, caused by programmer error, can only be stopped by pressing the BREAK key or turning the computer off.
- M:** Mega; one million. When referring to memory, two to the twentieth power; 1,048,576 in decimal notation.
- Machine infinity:** The largest number that can be represented in a computer's internal format.
- Machine language:** The lowest-level language. It is a pattern of binary coding that tells the computer what to do.
- Mainframe:** Originally meant the CPU, now refers to large computers. However, in microcomputers, the cabinet that holds the CPU is also called a mainframe.
- Mantissa:** For a number expressed in floating point notation, the numeral that is not the exponent.
- Mass storage:** The files of computer data that are stored on media other than the computer's main memory (RAM), such as cassettes or diskettes.
- Matrix:** A set of numbers or terms arranged in rows and columns. Each element is accessed in terms of its subscript.
- Memory:** The internal hardware in the computer that stores information for further use.
- Microcomputer:** A fully operational small computer that uses a microprocessor as its CPU.
- Microprocessor:** A central processing unit contained on a single chip.
- Minicomputer:** A small to medium-sized computer offering a range of capabilities somewhere between those of a microcomputer and a mainframe.
- Minifloppy:** Diskette.
- Modem:** A modulating and demodulating device that enables computers to communicate over telephone lines. Electronic signals from the computer are converted into sound, which in turn, are reconverted into electronic signals at the other end. An acoustic coupler is a modem device into which a telephone can be placed.

Monitor: A television receiver or CRT device used to display computer output.

Nest: Used to incorporate structure of some kind into another structure of the same kind. For example, you can nest loops within other loops, or call subroutines from other subroutines.

Network: An interconnected group of microcomputers or terminals linked together for specific purposes.

Notation: A set of symbols, and the rules for their use, for the representation of data.

Null: Empty, having no meaning; in particular, a string with no characters in it.

On-line: Interactive

Operand: That which is operated on. For example, in the expression

$$A = B \text{ OR } C$$

OR is the operator and B and C are the operands.

Operation: A well-defined action that when applied to any permissible combination of known entities, produces a new entity.

Output: Information or data transferred from the internal memory of the computer to some external device, such as a CRT, a mass-storage device, or a printer.

Output device: A device designed to take information out of a computer.

Overflow: When the result of an operation exceeds the capacity of the intended unit of storage.

Parallel: The performance of two or more operations or functions at the same time. For instance, a parallel port accepts all eight bits of a byte at one time, in contrast to a serial port that accepts only one bit at a time.

Parameter: A name in a procedure that is used to refer to an argument passed to that procedure.

Pascal: A powerful, high-level computer language for business and general use. It is named for the French philosopher and mathematician Blaise Pascal (1623 - 1662).

PEEK: A BASIC command that tells the computer to look into a specific location in the computer's memory and see what is stored there.

Peripheral: Any I/O device, such as a printer.

Personal computer: Microcomputer; home computer.

POKE: A BASIC command that tells the computer to put a new number into a specific location in the computer's memory.

Ports: Points of access to a computer. These channels through which computers send and receive data are either serial or parallel. That is, they send or receive data one bit at a time (serially) or several bits at the same time (in parallel). Telephone communication, for example, is done serially.

- Position:** In a string, each location that may be occupied by a character and that may be identified by a number.
- Precision:** A measure of the ability to distinguish between nearly equal values.
- Printer:** A device for producing paper copies (hard copy) of the data output by the computer. There are two basic kinds of printers for microcomputers—dot matrix and letter quality. The first kind forms letters by striking the paper with small pins, forming each letter with a pattern of dots. As the technology has advanced, more and more pins are being used for each letter, resulting in a denser image. Some dot-matrix printers squeeze the dots so closely together that they look almost like letter-quality print, which is made by machines that form letters with a single impact, as do traditional typewriters. When dot-matrix printers produce letters that are close to the quality of traditional electric typewriters, the result is often called “correspondence-quality” print. Letter-quality printers run at speeds between 12 and 40 characters per second, and dot-matrix printers can run as fast as 150 characters per second. Dot-matrix printers generally cost less than do letter-quality printers, although as technology advances the cost of both kinds is falling. Many letter-quality printers are called “daisy wheel printers” because they use print elements that resemble daisies, rather than the ball-type print element on an IBM electric typewriter.
- Printout:** Any sheet of paper, or collection of paper (hard copy), a computer printer produces.
- Program:** An organized group of instructions written in a language the computer understands, directing it to a solution to a problem.
- Programming:** The process of writing a program in a language a computer can understand.
- Prompt:** A symbol, usually a question mark, that appears on the screen, indicating that it is awaiting information.
- Queue:** A line or list of items waiting for service; the first item that went into the queue is the first item to be serviced.
- Random access memory (RAM):** The read/write memory available for use in the computer. Through random access the computer can retrieve or deposit information instantly at any memory address.
- Random number generator:** A program statement or hardware device that provides a number that cannot be predicted. This is very useful in decision-making programs. For instance, using a random number generator, the computer can simulate dice rolls.
- Range:** The set of values that a quantity or function may take.
- Read:** The act of taking data from a storage device, such as a diskette, and placing it in the computer’s memory.
- Read only memory:** A random access memory device that has perma-

nently stored information. The contents of this memory are set during manufacture.

Recursive: Pertaining to a process in which each step makes use of the results of earlier steps, such as when a function calls itself.

Register: A small temporary storage device in the computer. It holds data that the computer is about to use.

Reserved word: A word that is defined in a programming language for a special purpose, and that you cannot use as a variable name.

Response time: The time it takes a computer to answer a question or accept a line of input.

RETURN key: Causes the next character to be printed in the left column of the screen.

ROM: Read Only Memory.

Routine: Part of a program, or a sequence of instructions called by a program, that may have some general or frequent use.

Row: A horizontal arrangement of characters or other expressions.

Scalar: A value or variable that is not an array.

Scan: To examine sequentially, part by part.

Scrolling: The ability to move lines displayed on a CRT terminal screen either up or down.

Sector: The smallest block of data that can be written to or read from a disk file.

Segment: A particular 64K-byte area of memory.

Semiconductor: A metal or other material (such as silicon) with properties between those of conductors and insulators. Its electrical resistance can be changed by electricity, light, or heat.

Software: The instructions by which a computer operates, also known as *programs*. The information produced by the computer is generally called *data*. Software is the electronic instructions that enable the user to tell the computer what to do.

Sort: A procedure that arranges a group of elements into some kind of sequence—for example, sorting them into alphabetical order.

Spread sheet: A program that sets up an electronic spread sheet in which the lines and columns are automatically calculated according to formulas chosen by the user. When one number is changed, the program automatically changes all the sums and multiples that are affected.

Statement: A meaningful expression that may describe or specify operations and is complete in the context of the particular programming language.

Storage device: A unit into which data can be entered, retained, and retrieved, for example, punched cards, magnetic tapes, disks, floppy disks.

String: A set of letters, numbers, and/or characters.

Structured programming: A method of programming in which clarity is emphasized through three basic control structures.

Subroutine: A part of a program that can be executed by a single statement. It is especially useful when a certain part of the program has to be executed many times or has to be accessed from different points in the program.

Subscript: A number that identifies the position of an element in an array.

Syntax: The rules governing the structure of a language.

System: All the various hardware components that make the computer usable, including the computer itself, printer, joystick controller, disk drive, and so on.

Tape drives: Some less expensive microcomputers use tape recorders instead of disk drives to store information and programs. Tape drives are very slow in comparison, however, because they store data sequentially. It is time-consuming for the tape recorder to wind and rewind a tape, looking for information. And it is impossible for the computer to insert data among other data on a tape.

Target: In an assignment statement, the variable whose value is being set.

Terminal: An input/output device used to communicate with a computer and receive information from it.

Text editor: A computer program that permits the contents of memory to be changed. It can modify either data or the program.

Trailing: Located at the end of a string or number.

Transistor: A semiconductor device that acts primarily as an amplifier or a current switch.

Trap: A set of conditions that describe an event to be intercepted and the action to be taken after the interception.

Truncate: To remove the ending elements from a string.

Variable: A quantity that can assume any of a given set of values.

Variable-length record: A record having a length independent of the length of other records in the file.

Word processor: A very special computer program that helps to manipulate text. It permits the writing of documents, allows inserts or changes of words or paragraphs, and prints the document letter perfect.

Write: To store data on external media such as diskette or cassette. The expression "write to diskette" means that the information stored in the computer's memory is sent to the diskette, where it is stored permanently.

Appendix A

Abbreviations for BASIC Keywords

One of BASIC's more attractive features is that it allows you to abbreviate commands such as PRINT, RETURN, and POKE. In almost every case the abbreviation consists of the first letter of the command word followed by the shifted second letter. The following list contains all of the keywords in Commodore BASIC as well as their abbreviations. The right-hand column shows you the graphic representation of the abbreviation as it appears on the screen.

Com- mand	Abbreviation	Appearance	Com- mand	Abbreviation	Appearance
ABS	A SHIFT-B	A	END	E SHIFT-N	E
AND	A SHIFT-N	A	EXP	E SHIFT-X	E
ASC	A SHIFT-S	A	FN	NONE	FN
ATN	A SHIFT-T	A	FOR	F SHIFT-O	F
CHR\$	C SHIFT-H	C	FRE	F SHIFT-R	F
CLOSE	CL SHIFT-O	CL	GET	G SHIFT-E	G
CLR	C SHIFT-L	C	GET#	NONE	GET#
CMD	C SHIFT-M	C	GOSUB	GO SHIFT-S	GO
CONT	C SHIFT-O	C	GOTO	G SHIFT-O	G
COS	NONE	COS	IF	NONE	IF
DATA	D SHIFT-A	D	INPUT	NONE	INPUT
DEF	D SHIFT-E	D	INPUT#	I SHIFT-N	I
DIM	D SHIFT-I	D	INT	NONE	INT


Com-mand	Abbreviation	Appearance	Com-mand	Abbreviation	Appearance
LEFT\$	LE SHIFT-F	LE	RIGHT\$	R SHIFT-I	R
LEN	NONE	LEN	RND	R SHIFT-N	R
LET	L SHIFT-E	L	RUN	R SHIFT-U	R
LIST	L SHIFT-I	L	SAVE	S SHIFT-A	S
LOAD	L SHIFT-O	L	SGN	S SHIFT-G	S
LOG	NONE	LOG	SIN	S SHIFT-I	S
MID\$	M SHIFT-I	M	SPC(S SHIFT-P	S
NEW	NONE	NEW	SQR	S SHIFT-Q	S
NEXT	N SHIFT-E	N	STATUS	ST	ST
NOT	N SHIFT-O	N	STEP	ST SHIFT-E	ST
ON	NONE	ON	STOP	S SHIFT-T	S
OPEN	O SHIFT-P	O	STR\$	ST SHIFT-R	ST
OR	NONE	OR	SYS	S SHIFT-Y	S
PEEK	P SHIFT-E	P	TAB(T SHIFT-A	T
POKE	P SHIFT-O	P	TAN	NONE	TAN
POS	NONE	POS	THEN	T SHIFT-H	T
PRINT	?	?	TIME	TI	TI
PRINT#	P SHIFT-R	P	TIMES\$	TI\$	TI\$
READ	R SHIFT-E	R	USR	U SHIFT-S	U
REM	NONE	REM	VAL	V SHIFT-A	V
RESTORE	RE SHIFT-S	RE	VERIFY	V SHIFT-E	V
RETURN	RE SHIFT-T	RE	WAIT	W SHIFT-A	W

Appendix B

































Screen Display Codes





































































The Commodore has two separate character sets, only one of which can be used (or displayed) at a time. Each set is available directly from the keyboard and can be alternately accessed by simultaneously pressing the SHIFT and Commodore keys. Either set can be accessed from BASIC with the POKE command: POKE 53272,21 for Set 1 and POKE 53272,23 for Set 2. You can then poke the numbers for individual characters into the screen memory (locations 1024–2023). For example,

POKE 1024,94

yields either π or , depending on the set you have selected, at location 1024. Also, while this chart lists only codes 0–127, poking 128–225 will create reverse images of the characters. Furthermore, you can control the color of each character by poking a special section of memory (55296–56295) along with a color code (1–16). For convenience many symbols and characters are available from both sets. (To print a character with the CHR\$ function, see Appendix C.)

Set 1	Set 2	POKE	Set 1	Set 2	POKE	Set 1	Set 2	POKE
@	@	0	C	c	3	F	f	6
A	a	1	D	d	4	G	g	7
B	b	2	E	e	5	H	h	8

Set 1	Set 2	POKE	Set 1	Set 2	POKE	Set 1	Set 2	POKE
I	i	9	%	%	37		A	65
J	j	10	&	&	38		B	66
K	k	11	'	'	39		C	67
L	l	12	((40		D	68
M	m	13))	41		E	69
N	n	14	*	*	42		F	70
O	o	15	+	+	43		G	71
P	p	16	,	,	44		H	72
Q	q	17	-	-	45		I	73
R	r	18	.	.	46		J	74
S	s	19	/	/	47		K	75
T	t	20	0	0	48		L	76
U	u	21	1	1	49		M	77
V	v	22	2	2	50		N	78
W	w	23	3	3	51		O	79
X	x	24	4	4	52		P	80
Y	y	25	5	5	53		Q	81
Z	z	26	6	6	54		R	82
[[27	7	7	55		S	83
£	£	28	8	8	56		T	84
]]	29	9	9	57		U	85
↑	↑	30	:	:	58		V	86
←	←	31	;	;	59		W	87
SPACE	SPACE	32	<	<	60		X	88
!	!	33	=	=	61		Y	89
“	“	34	>	>	62		Z	90
#	#	35	?	?	63			91
\$	\$	36			64			92

Set 1	Set 2	POKE	Set 1	Set 2	POKE	Set 1	Set 2	POKE
		93			105			117
		94			106			118
		95			107			119
SPACE	SPACE	96			108			120
		97			109			121
		98			110			122
		99			111			123
		100			112			124
		101			113			125
		102			114			126
		103			115			127
		104			116			



Appendix C

ASCII and CHR\$ Codes

You can enhance your ability to manipulate character strings in BASIC by using the ASCII code. The following table lists the decimal value of all the characters in Commodore 64 BASIC. Typing

































```
PRINT CHR$(X)
```
































where *X* is a value in the right hand column, will yield the corresponding character in the left-hand column. Typing

```
PRINT ASC("X")
```

where *X* is a character in the left-hand column, will yield the corresponding decimal value.

Character	Decimal Value	Character	Decimal Value	Character	Decimal Value	Character	Decimal Value
	0		12		24	\$	36
	1	RETURN	13		25	%	37
	2	Lower Case	14		26	&	38
	3		15		27	.	39
	4		16	Red	28	(40
White	5	CRS Down	17	CRS Right	29)	41
	6	RVS ON	18	Green	30	*	42
	7	HOME	19	Blue	31	+	43
DISABLES SHIFT-COM	8	DEL	20	SPACE	32	,	44
ENABLES SHIFT-COM	9		21	!	33	-	45
	10		22	"	34	.	46
	11		23	#	35	/	47

Character	Decimal Value	Character	Decimal Value	Character	Decimal Value	Character	Decimal Value
0	48	J	74		100		126
1	49	K	75		101		127
2	50	L	76		102		128
3	51	M	77		103	Orange	129
4	52	N	78		104		130
5	53	O	79		105		131
6	54	P	80		106		132
7	55	Q	81		107	f1	133
8	56	R	82		108	f3	134
9	57	S	83		109	f5	135
:	58	T	84		110	f7	136
;	59	U	85		111	f2	137
<	60	V	86		112	f4	138
=	61	W	87		113	f6	139
>	62	X	88		114	f8	140
?	63	Y	89		115	RETURN	141
@	64	Z	90		116	Upper Case	142
A	65	[91		117		143
B	66	£	92		118	Black	144
C	67]	93		119	CRS Up	145
D	68	↑	94		120	RVS OFF	146
E	69	←	95		121	CLR	147
F	70		96		122	INST	148
G	71		97		123	Brown	149
H	72		98		124	Lt. Red	150
I	73		99		125	Grey 1	151

Character	Decimal Value	Character	Decimal Value	Character	Decimal Value	Character	Decimal Value
Grey 2	152		162		172		182
Lt. Green	153		163		173		183
Lt. Blue	154		164		174		184
Grey 3	155		165		175		185
Purple	156		166		176		186
CRS Left	157		167		177		187
Yellow	158		168		178		188
Cyan	159		169		179		189
SPACE	160		170		180		190
	161		171		181		191

Values 192–223 are the same as 96–127.



Values 224–254 are the same as 160–190.

Value 255 is the same as 126.










ABOUT THE AUTHORS



HENRY MULLISH is Senior Research Scientist and Lecturer in Computer Science at the Courant Institute of Mathematical Sciences of New York University. He is the author of over a dozen books on computer programming.

DOV KRUGER went to high school in New York City and attended his first computer course, sponsored by NYU, at the age of 11. At the age of 16 he entered Stevens Institute of Technology where he is now a freshman in the electrical engineering/computer science department. He has previously co-authored *Applesoft BASIC: From the Ground Up* and *Zappers: 23 Games for the TI-99/4A*, both of them with Henry Mullish.







\$12.95

AT HOME WITH BASIC

At last there's an intelligible, practical, and entertaining way to learn good programming on your Commodore 64. This easy-to-understand, clearly organized book lets you work hands-on with your computer while you read. Whether you're a newcomer or a seasoned Commodore 64 owner, you will learn the fundamentals of programming, from the simplest commands to advanced programming tools like nested loops and numeric and string arrays, for doing serious, useful programs. With this book you will be able to create such programs as a text analyzer to improve your writing style and devise routines to take advantage of your machine's advanced features like color graphics and sound generation.

Bestselling computer authors Henry Mullish and Dov Kruger reveal insiders' tips on good programming practice and debugging techniques. With detailed instructions, fascinating problems to solve, self-checking quizzes, and review sections that let you proceed at your own pace, this confidence-building book will make you feel **AT HOME WITH BASIC** in no time.

Cover design by
Zimmerman/Foyster

Computer Book Division/Simon & Schuster, Inc.

0884-1270

0-671-49861-4

AT HOME WITH BASIC
MULLISH/KRUGER

COMPUTER
BOOKS
SIMON &
SCHUSTER