BY THE EDITORS OF CONSUMER GUIDE®

EVERYONE'S GUIDE TO BASIC

BY THE EDITORS OF CONSUMER GUIDE®

EVERYONE'S GUIDE TO

BASIC

BEEKMAN HOUSE
New York

Louis Weber, President
Publications International, Ltd.
3841 West Oakton Street
Skokie, Illinois 60076

Manufactured in the United States of America
10 9 8 7 6 5 4 3 2 1

Library of Congress Catalog Card Number: 83-63052
ISBN: 0-517-41594-1

This edition published by:
Beekman House
Distributed by Crown Publishers, Inc.
One Park Avenue
New York, New York 10016

Writer: Brian Starfire
Design: Jeff Hapner and Barbara Clemens
Cover photo: Donna Preis and George Siede

CONTENTS

WHO CAN USE THIS BOOK

The BASIC commands and functions discussed in this book are those used in Microsoft BASIC and Atari BASIC. Therefore, the exercises and programs included will work without modification in the standard BASIC languages on the following computers: Apple (II, II Plus, and IIe); Atari (all models); Commodore (PET, VIC 20, and 64); Epson QX-10; Franklin Ace; IBM (PC, PCjr, and PC XT); Mattel Aquarius II; NEC PC-6000; Osborne (all models); Radio Shack TRS-80 (Color Computer and Models 1, 2, 3, and 4); and Spectravideo (SV-318 and SV-328). You will also be able to use other computers with this book if you have Microsoft BASIC.

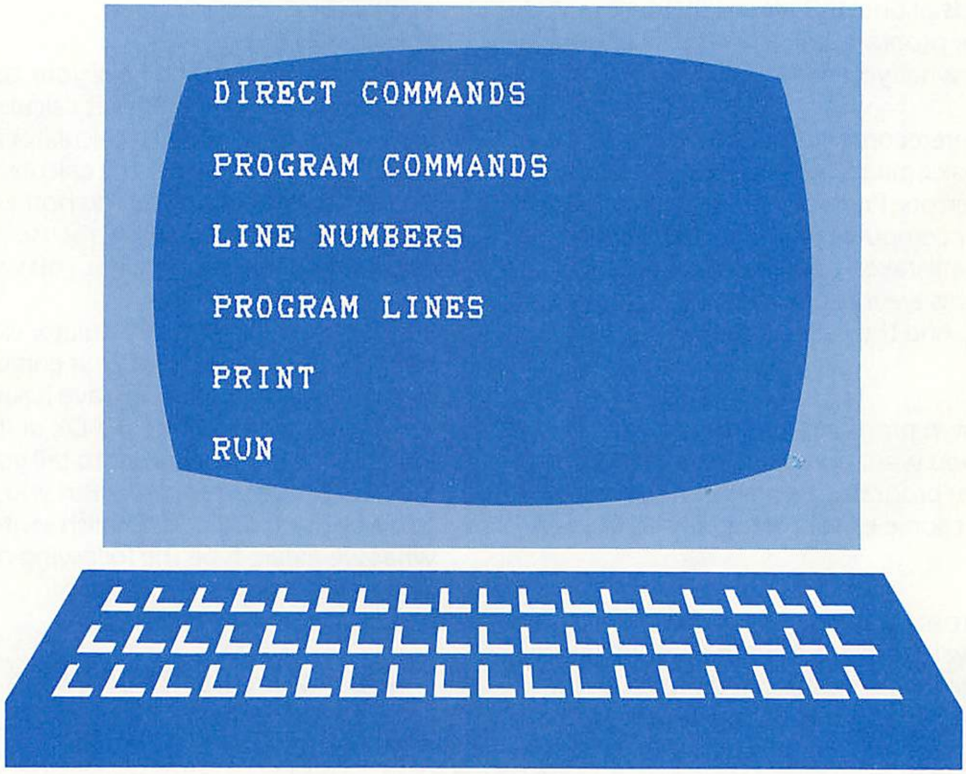
In order to use this book with a Texas Instruments TI-99/4A computer, you will need Extended TI BASIC, because the standard TI BASIC will not handle multiple-statement lines as used in some of the programs and exercises. (You can, however, do many of the exercises using standard TI BASIC.)

If your computer does not use Microsoft or Atari BASIC but uses some other version of BASIC, you will still be able to use most (and possibly all) of the exercises in this text.

You may learn a lot from this book without even having access to a computer. But, of course, the best way to learn BASIC is to use it. So get that computer up and running BASIC, and let's go!

Chapter 1

THE BASICS OF BASIC



So you bought a personal computer. You read the magazines, talked to friends who are into computers, comparison-shopped from dealer to dealer, and tried very hard to convince the rest of the family that your new purchase is more than an expensive toy. The big day came when you finally set the box down in the middle of the living room floor and unpacked your new computer, along with the cables, tapes, and pages of instructions that accompanied it. If all went well, and it usually does, the family got tired of playing video backgammon after a few hours. And then you began to wonder, "How do I do all these things I've read about?" This book is part of your answer. It teaches you the basics of BASIC, a computer language in which many software programs are written.

You have learned by now that any computer, no matter how sophisticated, requires **software**—instructions that tell it exactly what to do. Your computer system uses software programs which are stored on cartridges, cassette tapes, or floppy disks. You probably have already purchased prepackaged software for several purposes— video games, a home budget of some sort, and perhaps an educational program or a word processing package. As an alternative to buying software, you may have discovered books and magazines with programs printed in them. You may even have joined a users' group where you can get free or low-cost software (either in printed form or on cassette or disk). But now you want to understand what those printed programs mean and how they work, and you want to learn to write your own programs.

Why would you want to write your own programs? You can buy prerecorded software for nearly any purpose you can imagine. When you buy software, it is nearly always free from errors. Prerecorded software tends to be clear and understandable, seldom requiring detailed knowledge of computer technology. There are, however, two significant disadvantages of prerecorded software. First, it is not free (and may be quite expensive, depending on precisely what you need). Second, it is written to meet the general needs of many users, rather than the individual needs of one; this means that before you buy a particular program, you have to make sure that it does exactly what you need it to do.

In contrast to prerecorded programs, printed programs from books, magazines, or users' groups sometimes contain errors. Printed programs also must be typed into your computer, which means that you may create errors—mistakes you'll have to find and fix. Printed programs are inexpensive, but they can be a real headache. And they still weren't written especially for you.

Writing your own programs assures you that you get exactly what you want and need. And being able to write your own programs also means that you have the skill to alter some existing programs to fit your specific needs.

Of course, there are some types of software programs that you will want to buy rather than write yourself—like the longer, more complicated programs for word processing, spreadsheets, or extensive accounting. But other programs you may want to write for yourself—a checkbook balancer, an auto mileage log, a calorie-counting chart, an address book. With a little imagination you can make your computer work for you in all sorts of ways!

LEARNING A COMPUTER LANGUAGE

Writing your own programs requires the ability to communicate with the computer. This must be done in the computer's language, which is **not** English. That keyboard may be familiar, and the words on the screen may resemble English, but the computer is actually using what is known as a **high-level programming language**—a computer language that uses English words and translates them into computer instructions (in the form of numbers).

Most spoken or written human languages are far too complex for computers to understand directly, unless a lot of money is spent on the computer's memory and programs. To get around this limitation, we use high-level programming languages, and BASIC is one of these. (Other frequently used computer languages include FORTRAN, Logo, Pascal, COBOL, and FORTH.) Originally developed at Dartmouth College in 1955, BASIC (an acronym for **B**eginner's **A**ll-Purpose **S**ymbolic **I**nstruction **C**ode) can be used for many different types of programs, and it is relatively easy to use.

To help you understand how your computer works, let's compare it with a pocket calculator. What happens when you turn on a calculator? Absolutely nothing. You have to tell the calculator what to do. By itself, the calculator can do nothing at all. But by pressing its keys, you are in a sense programming the calculator to make it do what you want it to do.

Punching numbers on a calculator is simple enough, but how do you program your computer? When you turn on the computer and have it running BASIC, the screen should say OK or READY, or it may display a flashing box called a **cursor** to tell you that it's ready for you to tell it what to do. But you can't tell it what to do by using ordinary English sentences. To see what we mean, type the following on your computer keyboard:

```
ADD TWO AND TWO AND TELL ME
THE ANSWER.
```

Press the RETURN key (or the ENTER key, whichever your computer has). Your technological marvel immediately displays an answer something like this:

```
SYNTAX ERROR
```

or

```
ERROR--ADD TWO AND TWO AND
TELL ME THE ANSWER.
```

In order to get an answer from the computer, you have to ask the question in a language the computer understands. You must use the BASIC commands that the computer is programmed to understand and follow.

COMMAND FORMS: DIRECT AND PROGRAM

When running BASIC, your computer accepts two types of commands: **direct** commands and **program** commands.

Direct Commands

A **direct command** is a command that the computer executes immediately when you press the RETURN key (or the ENTER key, whichever your computer has).

Try some direct commands on your computer now. Type in the following, pressing the RETURN or ENTER key after each line:

```
PRINT 22+30

PRINT 5-3

PRINT 5*5

PRINT 10/2
```

Each time you press the RETURN key, your system responds with the answer—first 52, then 2, then 25, and finally 5.

In each case, the computer provided you with an immediate answer because you gave it a direct command. The PRINT command tells the computer to display on the screen the value of whatever follows the command. (The asterisk symbol tells the computer to multiply, and the slash tells it to divide.)

Some BASIC commands (like PRINT and LET) can be used either as direct commands or as program commands. Other commands (like SAVE) are nearly always used as direct commands.

Program Commands

As the name implies, **program commands** are those that are used in a sequence within a program to perform a specific task. To give the computer a program command, you precede the command with a **line number.** Whenever the computer reads a new line starting with a number instead of a letter, it automatically includes that line in any program it may already have in its memory. It assumes that the first number in the line is the line number. When you press the RETURN key, the computer inserts the command in its proper numeric sequence with any other commands already stored in memory.

Line numbers enable the computer to keep track of where it is and where it is going. Together, the line numbers in a program make up a type of road map that the computer follows. The computer will not do anything with the commands in program lines until it is told to do so—when you give it the RUN command to start the program.

Every program line must contain a line number, a BASIC command, and whatever information the computer needs to act upon the BASIC command.

Some rules of BASIC always apply when writing programs:

1. Each program line must begin with a line number.

2. Each program line must have a larger line number than the preceding program line.

3. Although the lines do not have to be typed in numeric order, they are processed by the computer in numeric order. This means that if you type in line 10 and then line 30 and then line 20, the computer places the lines in their correct numeric sequence in its memory.

4. No two different lines can have the same number. If you type two lines beginning with the number 10, the computer replaces the first line 10 with the second one. This is convenient when you want to change or correct a program line.

5. Every program line must contain a BASIC command to tell the computer what to do (with the exception of the LET command, which may be omitted in most versions of BASIC). Commands include words like PRINT, which you used as a direct command, and many others—LET, GOTO, READ, DATA, INPUT, FOR, NEXT, GOSUB, CLEAR, CONT, STOP, END, and more.

These rules will be second nature to you by the time you finish this book.

As you work through the exercises and programs in this book, you will be introduced to many BASIC commands. These commands and brief descriptions of what they do are also included in a list in the back of the book entitled "BASIC Commands Used in This Book." You can refer to that list whenever you need to refresh your memory about the use of a particular command.

INSTRUCTIONS FOR TYPING ALL PROGRAM LISTINGS

For all program listings (printed in this book or anywhere else), there are two rules that you must follow:

1. Type all program lines (or direct commands) exactly as they are shown, including numerals, quotation marks, all other characters and punctuation marks, and spaces.

2. Press the RETURN key (or the ENTER key, whichever your computer has) after you finish typing each line.

In this book, programs or commands for you to type in are printed in color type. Computer responses to what you have typed are shown in color boxes.

SPECIAL COMMANDS AND FUNCTION KEYS

There are some special commands and function keys on all computers. While some of these vary from system to system, others are common to most. For example, all computers have a RETURN key or an ENTER key which you press to send commands or information to the computer's working memory. For the sake of simplicity, we will refer to this as the RETURN key.

There are a few special commands and keys that you should avoid using unless you are sure what they do on your particular computer. On some computers, holding the CONTROL key while pressing the C key will stop a program; on other computers, the BREAK key performs this function. On some computers, the CONTROL key alone will reset the computer (which has the same effect as turning the computer off and then back on). This could be disastrous to the execution of a program. You could erase a program or other information stored in memory. Or you might have to start running a program again from the beginning. Be sure to check your operating manual for information on special commands and special key functions on **your** computer.

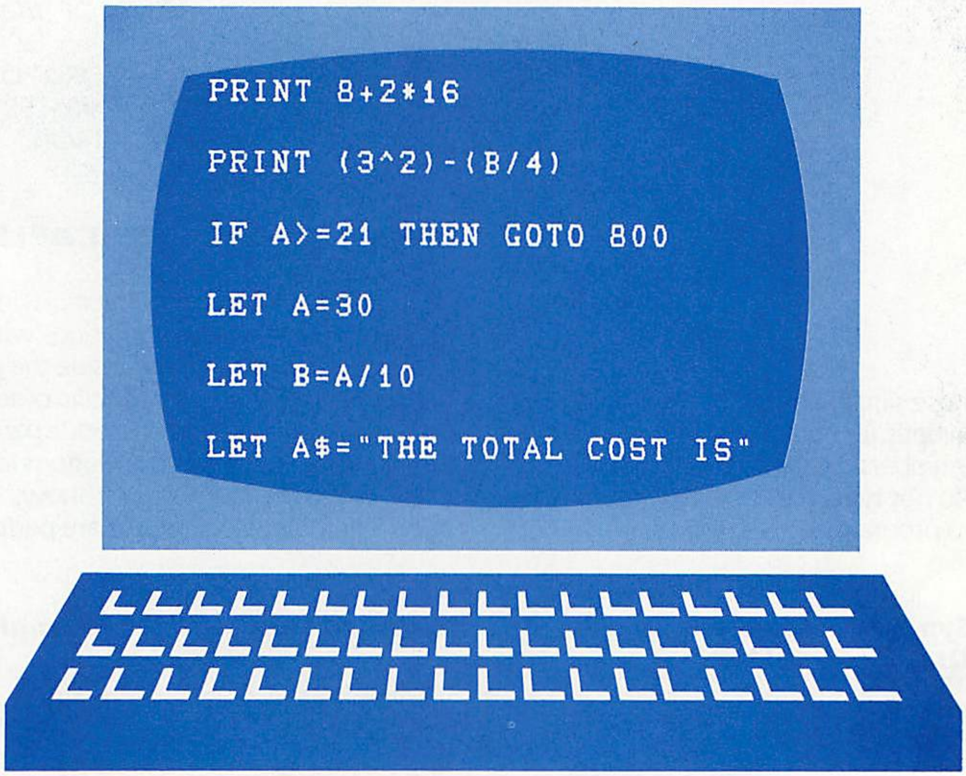
REVIEW

In this chapter, you learned that...

- Your computer can be operated using either direct commands or program commands.
- A direct command causes the computer to act on the command as soon as you press the RETURN key.
- Program commands require line numbers and are executed by the computer in sequential order after you've given the RUN command.
- When typing a program listing, you must type it **exactly** as printed.

Chapter 2

ARITHMETIC OPERATIONS



An **operation** is simply any action taken by the computer in response to a command. The BASIC language can be divided into four types of operations:

arithmetic operations
input and output operations
control operations
library operations

Regardless of the size of the program you are writing, you always begin by analyzing what you want to accomplish and then choosing operations to fit your needs.

In this chapter, we will review arithmetic operations (a few of which you used with direct commands in Chapter 1). We will review the other three types of BASIC operations in other chapters.

Arithmetic operations are the easiest group of BASIC operations to understand. Arithmetic operations include the standard mathematical operations of addition, subtraction, multiplication, division, and exponentiation (raising to powers). Personal computers use the following symbols to perform these operations:

Operation	Symbol Used
addition	+
subtraction	−
multiplication	*
division	/
exponentiation	∧ (varies among computers)

In addition to these simple arithmetic operations, some other arithmetic functions deal with relationships between numbers. These are shown in the following table. (Do not type in the examples now; they must be within a program to operate.)

Function	Symbol Used	Example of Use
equal to	=	IF A=0 THEN PRINT "WE'RE OUT OF STOCK"
less than	<	IF A<18 THEN PRINT "WE DON'T SERVE DRINKS TO MINORS!"
greater than	>	IF A>15 THEN PRINT "ELIGIBLE FOR DRIVER'S LICENSE"
less than or equal to	<= **or** =<	IF X<=25000 THEN PRINT "EAST COAST ZIP CODE"
greater than or equal to	>= **or** =>	IF A>=21 THEN GOTO 200

There are also functions called **logical operators** that are used to tell the computer under what conditions it should take a specific action. For example, the word AND is used to mean "if both expressions are true." The word OR is used to mean "if either expression is true."

Function	Example of Use
AND	IF A>=16 **AND** A<=70 THEN PRINT "OF WORKING AGE"
OR	IF A$="FORD" **OR** "G.M." **OR** "CHRYSLER" THEN PRINT "BY DETROIT"

ORDER OF ARITHMETIC OPERATIONS

The computer executes calculations in a specific order. It always performs the operations within a set of parentheses first. Working inside the parentheses, it performs operations in a specific order. Once it has performed any calculations inside parentheses, it goes on to perform all other calculations following that same order. The following list shows the order in which arithmetic operations are performed.

Order of Operations	Example
parentheses	(6*3)
exponentiation	6∧3
negation	−3
multiplication/division	6*3 **and** 6/3
addition/subtraction	6+3 **and** 6−3
greater than/less than	A>3 **and** B<6
greater than or equal to/ less than or equal to	A>=6 **and** B<=3
AND	A=6 AND B=3
OR	A=6 OR B=3

What this order of operations means is that if you have a mathematical calculation like

(6+2*7) + (2∧2/2−1)*3

the computer will first compute the segments in the parentheses. In the first set of parentheses, it multiplies 2*7 to get 14 and then adds 6 to get 20. In the second set, it squares the first 2 (raising a number to the power of 2 is called **squaring**) to get 4, divides that by 2 to get 2, and subtracts 1 to get 1. Then, having reduced the calculation to 20+1*3, the computer first multiplies 1*3 to get 3, and then adds 20 to get 23. Although it took some time for us to describe all of this, the computer does all these calculations in just a split second and shows you only the final answer.

In cases of multiplication and division, addition and subtraction, and greater than and less than operations, one operation is precisely the opposite of the other, so it doesn't matter which the computer does first. For example, 5*2 and 5÷½ both equal 10, and 5−2 and 5+(−2) both equal 3. So if you need the answer to 15*4/2, the computer can first multiply 15 times 4 to get 60 and then divide by 2 to get 30. Or it can first divide 4 by 2 to get 2 and then multiply 15 times 2 to get 30. The answer is the same in such a case, regardless of the order of operations.

It might be important to know the correct order of operations if you decide to write a computer program to calculate your taxes or to do your math homework, depending on how you decide to set up the program and exactly what you have to do. It will also help you to keep the order of operations in mind when you are looking at printed programs and trying to figure out how they work.

WORKING WITH LARGE NUMBERS

There are limits to the size of the calculations that your computer can perform. Your computer will be able to handle your financial transactions (unless you are very, very rich). But if you use your computer to perform complex mathematics, you may occasionally ask your computer to perform a calculation that is too much for it to handle. If you do, it will let you know about it in a hurry by displaying an error message of some sort. You can demonstrate this by running the short program shown here. This program squares numbers—until the numbers get too large for your particular computer to handle.

Type the following (remembering to press the RETURN key after you type each line):

```
NEW

10 LET X=2

20 PRINT X

30 LET X=X*X

40 GOTO 20
```

The NEW command erases any program lines currently in the computer's memory, so that you can type in a totally **new** program.

The LET command assigns a value to a variable, in this case X. (We'll learn more about variables shortly.)

The PRINT command tells the computer to display on the screen the value indicated after it.

The GOTO command tells the computer to vary from its normal sequential execution of program lines, in this case to return to line 20 instead of looking for the next larger line number.

Now type the word RUN and press RETURN to run the program. Here are the results of running the program on our computer:

```
2

4

16

256

65536

4294967296

1.84467E +19

OVERFLOW ERROR IN LINE 30
```

Your results may differ from ours in how they look on the screen, the exact words, and the final answer. If your computer repeats the overflow error and will not stop, try pressing the CONTROL key and C key (or just the BREAK key). If that doesn't stop it, turn the computer off and then back on; you don't have anything in it that you need to keep right now.

If you're not familiar with scientific notation, that last number may look strange. The computer is letting you know that the number is too large to display all of it. So the computer added a decimal point and gave you an additional figure following the letter E. That figure indicates where the decimal point actually belongs (in this case, 19 places to the right of where the decimal point is displayed on the screen).

The error resulted when the number became so large that the computer could not handle the calculation. The limits in performing calculations vary among computers.

NUMERIC VARIABLES

You can put numbers into your computer's memory, using either direct or program commands, by creating a **variable** and assigning it a value (like B=10). You can create variables of single letters (T=43), two letters combined (AX=76), or a letter followed by a number (E4=82). Once you have assigned a numeric value to a variable, that variable retains the value assigned until you assign it a different value.

The LET command is used to assign values to variables (such as LET A=5). In most versions of BASIC, however, the word LET can be omitted; you can simply type in A=5 and the computer understands that you are assigning the value of 5 to the variable A.

Try this exercise. First type in the following to assign values to variables A, B, and C. Press RETURN after each line.

```
A=5

B=10

C=20
```

Notice that we have not used the word LET to assign these variables their values. If this exercise doesn't work for you, try typing in the word LET when assigning variables; your version of BASIC may require that you include LET.

Now type in these direct commands (again, pressing RETURN after each line):

```
PRINT A

PRINT B*A

PRINT C
```

The computer uses the values you assigned to the variables and responds like this:

```
PRINT A
```

```
5
```

```
PRINT B*A
```

```
50
```

```
PRINT C
```

```
20

READY
```

The numbers were stored by the computer in its program memory, and the computer used them when you requested the calculations. Variables are useful in programs where you need to make calculations using amounts that may vary over time or under certain conditions (for example, if you wanted to determine what your mortgage payments would be at different interest rates).

USING NUMERIC VARIABLES IN A PROGRAM

Now let's see how we can use variables in a program. Type in the following program, pressing RETURN at the end of each line.

```
NEW

10 A=10

20 B=2

30 C=5

40 PRINT A;B;C

50 PRINT A+B+C

60 PRINT A-B
```

When you finish typing the program, type the word RUN and press RETURN to start the program. The answers will be displayed on your screen, like this:

```
10 2 5

17

8
```

Now type LIST and press the RETURN key. The LIST command tells the computer to display all of the program lines that are stored in memory. This is useful when you want to check your program or to make changes or corrections in a program.

You can also list a single line, or a group of lines, by typing in the LIST command along with the line numbers of the lines you want to see. For example, type the following:

```
LIST 10
```

Now press the RETURN key. The computer displays line 10 of the program.

```
10 A=10
```

When you entered the RUN command to execute the program, the first thing the computer did was to retrieve line 10 from its memory. Since this line told the computer to LET A=10, the computer matched the value of 10 to the variable A.

Now type the following:

```
LIST 20-60
```

Now press the RETURN key. The computer lists the rest of the program.

When the computer finished running line 10, it went on to process line 20 and then line 30. It matched the value of 2 to the variable B, and the value of 5 to the variable C.

When the computer got to line 40, it followed the command to PRINT A;B;C by displaying 10 2 5 on your screen. The semicolons between the variables told the computer to display the values of those variables on the same line.

The computer then proceeded to line 50, where it was told to print the sum of A+B+C (PRINT A+B+C); so the computer added the values and printed the result (17).

Finally, line 60 had the computer subtract B from A and print the result (8). Since there were no more lines for the computer to process, it stopped and displayed the READY or OK message. You have just performed data processing using a computer program!

STRING VARIABLES

In addition to storing numbers in your computer's memory, you can store words, phrases, and sentences in the form of character strings. A **character string** is any group of letters, numerals, symbols, and spaces enclosed by quotation marks. A character string is really a value, just as a number is a value, and the computer handles numeric values and character strings similarly. A variable name followed by a dollar sign (T$, AX$, B1$) is called a **string variable.** The dollar sign tells the computer that you are assigning a character string (not a number) to that variable. If you omit the dollar sign and still assign the variable a character string rather than a numeric value, the computer will give you a TYPE MISMATCH error message.

Let's try some character strings. Type the following direct commands:

```
LET A$="HI, "

LET B$="THERE!"

PRINT A$+B$
```

The computer responds like this:

```
HI, THERE!
```

If you got up and walked away from your computer and returned a week later, the computer would still remember that A$="HI, " and B$="THERE!" (as long as you didn't turn the computer off, of course). The ability to store and manipulate character strings is what actually sets your computer apart from a calculator.

In using a character string, what you are essentially doing is creating an abbreviation for whatever you put inside the quotation marks. For example, as a programmer for an automobile manufacturer, you might want to write a program to select any automobile part from the computer's memory and print a description of that part. You can take the full name of a part, such as LEFT REAR TAILLIGHT FOR 1982 LEMON, and represent it with a much simpler name like A$. To do so, type this direct command:

```
LET A$="LEFT REAR TAILLIGHT
FOR 1982 LEMON"
```

Once you have programmed that value into the computer, you can use the abbreviation A$ anywhere within your program in place of the full name of the part.

You can use entire words as string variables (such as PART$), but most computers only examine the first two characters of the variable name. So you shouldn't use PART$ and PANT$ within the same program, since most computers would interpret both names as PA$ and would not be able to distinguish between the two names.

You can also use character strings with commands other than the LET command. You will use them often with the PRINT command.

Here are a few rules to remember when using character strings:

1. All string variables must end with a dollar sign.

2. Quotation marks indicate the beginning and the end of the character string, so you must not use quotation marks inside the string. For example, A$="ROBERT'S NICKNAME, "BOB" IS ACCEPTABLE" would be read by the computer as A$="ROBERT'S NICKNAME, " and the remaining part of the line would trigger a syntax error message. You could get a character string that would be close to what you want by using apostrophes instead of quotation marks inside the character string: A$="ROBERT'S NICKNAME, 'BOB,' IS ACCEPTABLE".

3. Any numerals in the character string will not be treated as numeric data by the computer. This means that if you put 3+2 inside quotation marks, the computer will not give you 5; it will give you 3+2.

CARPET COST PROGRAM—A REAL-LIFE APPLICATION

We can write an infinite variety of programs using just arithmetic operations, numeric variables, and string variables. Suppose you want to know how much some new carpeting for your home would cost. Let's say that the carpet you want costs $13.00 per square yard. You want to put carpet in three rooms: the first room measures 10 by 12 feet; the second measures 14½ by 13 feet; and the third measures 16 by 20 feet. You want to know what the total bill will be after you add 5% sales tax. Considering these facts, we can write a program to calculate the total bill.

If we consider what we have to do and proceed to write a sequence of steps to do it, we can create a **flowchart** that looks something like this:

tell computer price per square yard of carpet → line 10

calculate square feet of first room
calculate square feet of second room → line 20
calculate square feet of third room

add footage of all three rooms for total square footage → line 30

divide by 9 to figure total square yards → line 40

multiply total square yards by price per yard of carpet → line 50

calculate sales tax → line 60

add sales tax to total carpet cost → line 70

print total amount → lines 80 & 90

We can use this flowchart to write a program that will run on your system. Type this program in now:

```
NEW

10 LET PY=13

20 R1=10*12:R2=14.5*13:R3=16*20

30 LET F=R1+R2+R3

40 LET Y=F/9

50 LET P=Y*PY

60 LET TX=P*.05

70 LET T=TX+P

80 PRINT "TOTAL PRICE IS"

90 PRINT "$";T
```

Type RUN and press RETURN to start the program. Your answer will look like this:

Answer 953.225

```
TOTAL PRICE IS
$1057.12
```

To see what we've just done, let's go over each line of the carpet cost program in detail:

Line 10 defines the price per yard of the carpet.

Line 20 calculates the square feet of carpet required for each room. The use of the colon (:) between the calculations for the rooms enables us to calculate for all three rooms using only one line of program space. If your system didn't respond correctly to this program, try making line 20 into three lines (20, 21, and 22) with one room's square feet calculated in each line. You might also try adding the LET command to each segment of line 20 (although in nearly all forms of BASIC, the LET command can be omitted).

Line 30 adds together the square footage of the three rooms to determine the total number of square feet of carpet you need.

Line 40 divides the number of square feet by nine to determine the number of square yards.

Line 50 multiplies the total number of square yards of carpet needed by the price per square yard of the carpet (as defined in line 10).

Line 60 calculates the amount of the 5% tax.

Line 70 adds the tax to the total cost of the carpet.

Line 80 causes the computer to display on the screen the words enclosed in the quotation marks. This works any time you use the PRINT command: whatever you put inside quotation marks following the PRINT command will be displayed on the screen exactly as you typed it (only without the quotation marks).

Line 90 causes the computer to display on the screen the dollar sign ($) by enclosing the dollar sign in quotation marks after the PRINT command. The semicolon (;) indicates that the computer is to print something else on the same line with the dollar sign, in this case the total cost of the carpet to you (including tax).

The lines in the carpet cost program actually perform what we set out to do in the flowchart.

You can use these same steps anytime you write programs for your system:

1. First, analyze the task that needs to be done.

2. Second, create a solution (a flowchart) for that task.

3. Third, write and debug the program. (**Debug** means to get rid of any **bugs,** or errors.)

REVIEW

In this chapter, you learned that...

- There are four types of BASIC operations: arithmetic, input/output, control, and library.

- Arithmetic operations use the symbols + for addition, − for subtraction, * for multiplication, / for division, and ∧ for exponentiation. Also used are the symbols > for greater than, < for less than, and = for equal to. The words AND and OR are logical operators.

- The computer always performs arithmetic operations in a specific order.

- There are limits to the size of numbers your computer can handle in calculations. These limits vary from computer to computer.

- You can store a numeric value in the computer and represent it with a variable of letters (such as A or TX) or with a letter and a number (like B1).

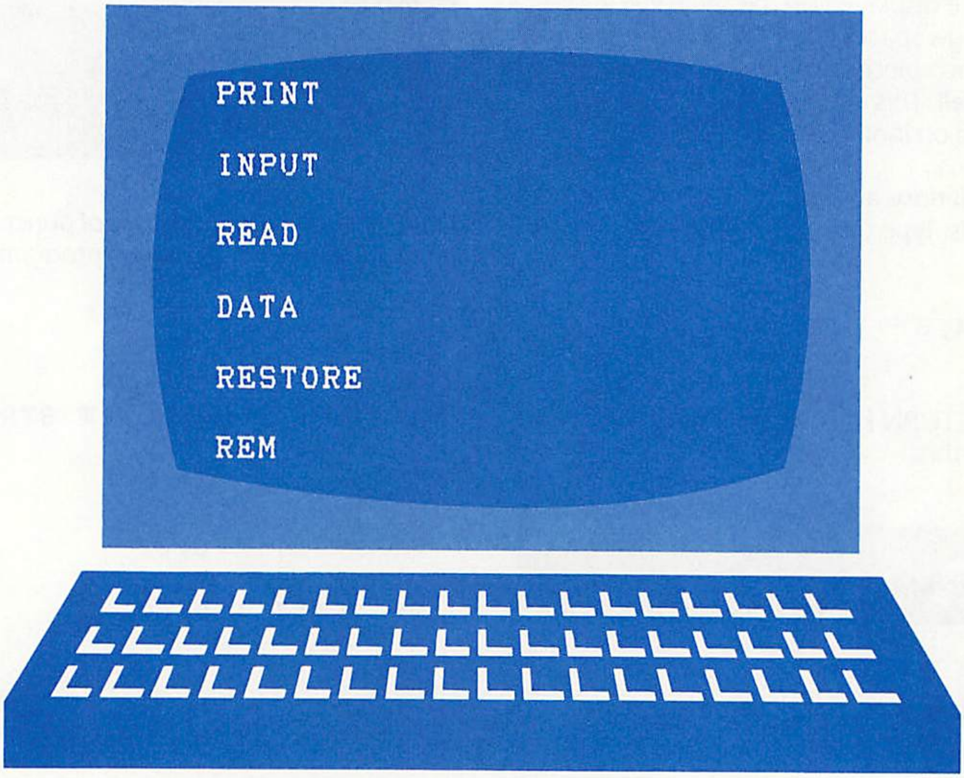
- You can store groups of letters, numbers, symbols, and spaces in any combination, enclosed in quotation marks, as a character string. Character strings are represented by string variables, which are similar to numeric variables except that they end with dollar signs: B$, TX$, A1$.

- You can use the colon to place more than one command on the same program line.

- You can use the semicolon to tell the computer to display more than one value on the same line on the screen.

Chapter 3

INPUT/OUTPUT OPERATIONS



You use input/output (I/O) operations to exchange information with your computer. In an input/output operation, you use BASIC commands to put information into, or get information out of, your computer.

Input statements put information into the computer's memory. The information may be typed at the keyboard, loaded from a cassette or disk, or transmitted over a telephone line with the use of a modem.

Output statements take information that is in the computer's memory and display that information on a screen, save it on a cassette or disk, or transmit it to a printer or another computer.

Let's look at some of the commands and procedures that are used to perform input/output operations.

PRINT STATEMENTS

PRINT statements cause the computer to send information to an output device; usually the computer displays the requested information on the TV or monitor screen to which the computer is connected. A PRINT statement consists of the command PRINT followed by the data to be printed. The data to be printed may take the form of variables, constants, character strings, or a combination of all three.

In Chapter 2, we used PRINT statements containing numeric variables and character strings. We also used punctuation to control the manner in which the data would be displayed. If there are several items of data to be printed within one PRINT statement, the data may be separated by commas, semicolons, or colons.

If commas are used in a PRINT statement, they separate the data into **fields** (blocks of space) when displayed. The number of fields in a line of print varies from system to system. If semicolons are used, the pieces of data are displayed on the same line and separated by single spaces. If colons are used, the data is treated as if each piece of data had been in a PRINT statement by itself. This means that the information will be displayed on individual lines.

To see these variations at work, try the following direct commands. Type this:

```
PRINT 2,4,6
```

and press the RETURN key. The result on the screen should look like this:

```
2        4        6
```

Now let's try using semicolons; type this:

```
PRINT 2;4;6
```

and press the RETURN key. The result on the screen should look like this:

```
2 4 6
```

To try using colons, type this:

```
PRINT 2:4:6
```

and press the RETURN key. The result on the screen should look like this:

```
2
4
6
```

Now let's combine these uses of punctuation into one program. Enter the following program:

```
NEW
10 A=22:B=44:C$="W STREET"
20 PRINT A;B;C$
30 PRINT A,B,C$
40 PRINT A:B:C$
```

When you type RUN and press RETURN, the result of this program is displayed on your screen like this:

```
22 44 W STREET
22      44      W STREET
22
44
W STREET
```

In our direct command examples, we used constants as data; in the program, we used variables, assigning values to the variables in line 10.

Line 10 is actually a LET statement, but we have omitted the optional command LET. To combine several LET statements on one line, separate the variable assignments by colons (just as you would to combine several PRINT statements on one line).

If you use a PRINT statement without any data in it, a blank line will be inserted on the screen when that line number is executed. This can make your programs more readable (at the expense of memory space).

With most systems, you can substitute a question mark for the word PRINT as a BASIC command. Try this by typing the following:

```
NEW

100 ? "HELLO"
```

When you run this program, the computer displays the word HELLO on the screen.

Now enter the LIST command. The computer should display the program line as follows:

```
100 PRINT "HELLO"
```

This demonstrates that the computer has read the question mark as the word PRINT.

INPUT STATEMENTS

The INPUT statement (called GET in some versions of BASIC) causes the computer to halt the program until you enter (or **input**) specific values requested by the program. The INPUT statement consists of the command INPUT followed by a variable. The type of variable used tells the computer what type of input to accept—numeric data or a character string.

Why would you ever need to use an INPUT statement? So far, we've used program statements and variables to assign values (such as in our carpet cost program). Suppose, though, you needed to assign a value that would vary every time you ran the program. An example is a home budget program that totals your monthly expenses. Since your electric bill varies from month to month, you couldn't use a program line like LET A=56.50 unless you knew in advance what the exact amount would be. Instead of assigning a definite value in the program, you could use an INPUT statement.

Try the following example:

```
NEW

50 PRINT "TELL ME A NUMBER"

60 INPUT A

70 PRINT "THE NUMBER IS"

80 PRINT A
```

Type RUN and press RETURN to start the program. When the computer executes line 60, it stops and prints a question mark on the screen. It is waiting for you to provide a number. When you type in a number, the computer assigns the value of that number to the variable A.

Just for fun, RUN the program again, and when the question mark appears, enter the word **three** instead of the numeral 3. The computer will give you an error message something like this:

```
REDO FROM START?
```

or

```
ERROR: TYPE MISMATCH IN 60
```

To make the program operate with words instead of numerals, you would have to change the numeric variables in lines 60 and 80 to string variables, like this:

```
60 INPUT A$

80 INPUT A$
```

The computer would then expect you to input a character string instead of a numeric value.

You can INPUT more than one value in the same line by using commas. Here is an example, but don't type it in right now because it needs a whole program in order to do anything:

```
10 INPUT A,B,A$
```

If the computer were to execute this program line, it would display a question mark on the screen. It would expect what you told it you would supply: a numeral, another numeral, and a character string. If you were to enter less data than it needed, the computer would print two question marks to tell you to enter more data.

When all of the necessary data had been given, the computer would assign the data to the variables used by the INPUT statement. It would then go on to execute the next line number in the program.

Type in the following program to demonstrate the use of an INPUT statement with multiple values:

```
NEW

10 PRINT "ENTER NAME AND AGE
IN THE FOLLOWING MANNER-
NAME, AGE"

20 INPUT A$,A

30 PRINT "YOUR NAME IS ";A$

40 PRINT "YOU ARE ";A;" YEARS
OLD."
```

Type RUN and press RETURN to start the program. When the computer displays a question mark on the screen, try entering just your name (which means typing your name and then pressing the RETURN key). The computer will display two more question marks to tell you that it needs more information, so enter your age to let it continue with the program.

Now RUN the program again. This time type in both your name and your age separated by a comma (such as SUSIE SMITH, 32). The computer will quickly complete the program.

If you press the RETURN key without entering any data when the computer asks for it, one of two things happens. On some computers, the program terminates. On others, the computer inserts a **null** string (a meaningless character set, consisting of zeros) or a value of zero.

As a shortcut and to save space, you can write an INPUT statement that acts as both an INPUT and a PRINT statement. Try this example:

```
NEW

10 INPUT "WHAT'S YOUR AGE?";A
```

Then run the program. Note that a semicolon must be included in a combined statement like line 20, to show the computer where the PRINT statement ends and the INPUT statement begins.

READ AND DATA STATEMENTS

READ and DATA statements are used together in programs. DATA statements allow you to store data in a program. READ statements allow you to call that data up at a later time.

A READ statement acts something like an INPUT statement; however, instead of putting a question mark on the screen and waiting for you to supply a value, a READ statement looks for a DATA statement in the program to supply the information. The values in DATA statements are read in sequential order.

The information in a DATA statement can be numeric values or character strings, but it must match the types of variables in the READ statements. The individual data items must be separated by commas (DATA 105,112,56,83). Character strings in DATA statements must be enclosed in quotation marks, just as they must anywhere else. DATA statements can appear anywhere in a program; the computer will find them when it needs them.

When the computer sees the command READ, it jumps out of the line number routine and goes in search of a DATA statement; it skips any line that does not contain the word DATA. It doesn't care what the line number of the DATA statement is. All it cares about is finding a DATA statement that it hasn't read yet. Once it finds a DATA statement, it reads the first piece of information it finds. If it needs more pieces of information, it keeps reading until it has enough information to assign values to all of its variables. When the computer has assigned values to all of the variables in the READ statement, it continues executing the program.

If the computer encounters another READ statement later in the program, it reads the next DATA statement that had not been read previously. The computer does not read the same DATA statement twice in the same program (unless you force it to by using the RESTORE command, which we'll discuss later in this chapter).

For now, remember that READ and DATA statements provide a powerful method of storing values in your computer's memory. READ and DATA statements are very useful in programs that make extensive use of files, such as recipes, home budgets, and mailing lists.

Try the following example to demonstrate the use of READ and DATA statements:

```
NEW

10 READ A,B,C:PRINT A;B;C

20 DATA 10,20

30 DATA 30,40
```

When you run the program, the system responds like this:

```
10 20 30
```

Line 10 told the computer to read and display on the screen the first three values in the DATA statements. So the computer looked through the program for DATA statements to supply values for the variables A, B, and C. When the computer had read all of the values in the first line of data (line 20), the next READ statement (READ C) called up the first value in the next DATA statement (line 30). Since the READ statement asked the computer to read only three values, the computer did not read or display the remaining value in line 30.

Now type in 30 and press the RETURN key. This erases line 30. LIST the program to see the following:

```
10 READ A,B,C:PRINT A;B;C

20 DATA 10,20
```

Now RUN the program again. You will get an OUT OF DATA error message. This is because there were three READ requests in the program, and there were only two DATA values left.

READ and DATA statements are used when you have large amounts of information to enter into the computer. For example, you might want to store words in the computer's memory for use in a word display program for young children. It is possible to display a word list without using READ and DATA statements, as in the following program:

```
NEW

100 LET A$="APPLE"

110 LET B$="DOG"

120 LET C$="CAT"

130 LET D$="BOY"

140 LET E$="CHAIR"

150 LET F$="GIRL"

160 LET G$="DESK"

170 LET H$="TOY"

180 PRINT A$,B$,C$,D$,E$,F$,G$,H$

190 END
```

Now try running the program. While this program works, you have to write an individual line for each word to be stored. If the program contained many more words, you would have a very large program (and a lot of typing).

You can perform the same task with a savings in memory space, and a lot less typing, by using READ and DATA statements, as in the following program:

```
NEW

100 FOR W=1 TO 8

110 READ X$

120 PRINT X$

130 NEXT W

140 END

1000 DATA "APPLE","DOG",
"CAT","BOY","CHAIR",
"GIRL","DESK","TOY"
```

When you run the program, it displays the list of words in the DATA statement in line 1000.

The FOR and NEXT commands in lines 100 and 130 set up a kind of **loop,** something we will discuss in detail in Chapter 4. In this case, the FOR and NEXT commands force the computer to go through lines 110 and 120 eight times, once for each word on the word list (in the DATA statement, line 1000).

Each time the computer gets to line 110, it ignores the other lines in the program and searches for the DATA statement (line 1000). When it finds the data it needs, it continues to execute the program, printing the value it has just assigned. When the eight DATA values have been read, the computer continues to line 140. There it follows the command to END the program.

If you wanted to have 25 words in the word list in this program, all you would have to do is add the extra words to the list and change line 100 to read as follows: 100 FOR W=1 TO 25.

RESTORE STATEMENTS

The RESTORE statement is used in combination with READ and DATA statements. It actually restores the information in the DATA statements so that the information can be used more than once. After a RESTORE statement is executed, the next READ statement goes back to the first DATA statement in the program.

Type in the following program to see how this works:

```
NEW

10 READ A$

20 RESTORE

30 READ B$

40 PRINT A$,B$

50 DATA "FIRST","SECOND"
```

When you run the program, your computer displays the following:

```
FIRST   FIRST
```

Because the program included a RESTORE statement, the computer read the first piece of information in the DATA statement twice, rather than reading the first piece of information the first time and the second piece of information the second time.

To see what the computer would display **without** the RESTORE command in the program, type 20 and press the RETURN key to erase line 20. Then RUN the program.

This time the system responds like this:

```
FIRST   SECOND
```

You might want to use a RESTORE statement in a program in which you need to use the same information several times (as we will in the telephone directory program in Chapter 4).

REM STATEMENTS: NOTES IN PASSING

The REM (for **remark**) statement is difficult to classify as a type of operation because the REM statement accomplishes absolutely nothing, as far as the computer is concerned. REM statements are used to make notes that aid the programmer in keeping track of what the program is doing. A typical REM statement is shown in line 90 of the program excerpt that follows:

```
90 REM THIS PROGRAM CALCULATES
INTEREST

100 LET A=X/100

110 . . . . (program continues)
```

When the computer reads the BASIC keyword REM, it ignores the remaining information in the line and proceeds to the next line number.

It is good practice to use REM statements often to keep track of what is happening in your program. REM statements also make it easier for others to understand your program if you leave notes throughout the program.

The only drawback with REM statements is that they take up memory space in your computer. If you are having difficulty storing a large program in your computer's memory, you can delete REM statements to save memory space.

PROGRAM EXERCISE: A SMALL-BUSINESS PAYROLL

Setting up a payroll program for a small business is an example of a popular application for small computers. The following is a small part of a payroll program. It demonstrates the use of string variables, numeric variables, and INPUT statements.

```
NEW

50 LET A$="J. SMITH"

60 LET B$="J. DOE"

70 LET C$="C. JONES"

80 A=9.75:REM PAY RATE

100 PRINT "HOW MANY HOURS
DID ";A$;" WORK?"

110 INPUT HA

120 PRINT "HOW MANY HOURS
DID ";B$;" WORK?"

130 INPUT HB

140 PRINT "HOW MANY HOURS
DID ";C$;" WORK?"

150 INPUT HC

200 PA=HA*A:PB=HB*A:PC=HC*A

300 PRINT "SALARY FOR ";A$:
PRINT "$";PA

310 PRINT "SALARY FOR ";B$:
PRINT "$";PB

320 PRINT "SALARY FOR ";C$:
PRINT "$";PC

350 PRINT:PRINT "TOTAL SALARY
FOR ABC COMPANY THIS WEEK IS
$";PA+PB+PC
```

When you run the program, pick a number of hours to insert each time the computer asks you for input, and the program will do the rest. If you get a SYNTAX ERROR message on the screen, LIST the program and look at it again—perhaps you put a semicolon where a colon ought to be or made some other typing mistake.

Here's how the program works:

Lines 50, 60, and 70 assign the names of the employees to string variables (A$, B$, C$).

Line 80 assigns the pay rate ($9.75 per hour) to the numeric variable A.

Lines 100 through 150 assign numeric variables (HA, HB, HC) to the number of hours worked by each employee.

Line 200 performs the salary calculations by multiplying the pay rate by the variables that represent the hours worked by the employees. Line 200 also assigns new variables (PA, PB, PC) that represent the results of these calculations (each employee's salary).

Lines 300, 310, and 320 print each employee's total salary for the week. Line 350 prints a blank line and then prints the total payroll for all employees combined.

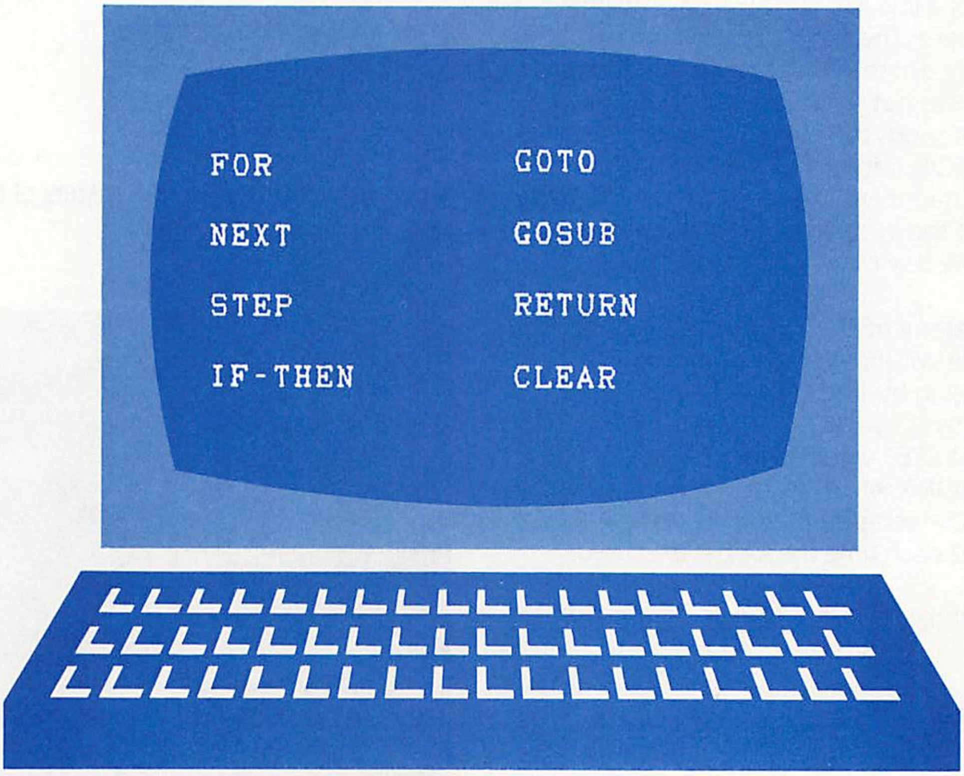
REVIEW

In this chapter, you learned that...

- PRINT statements tell the computer to print information, usually on the TV or monitor screen.
- INPUT statements tell the computer to request information from the person using the program.
- READ and DATA statements work together. READ statements tell the computer to read information that is stored in the corresponding DATA statements.
- RESTORE statements allow DATA statements to be read more than once in the same program.
- REM statements allow the programmer (you!) to make notes for personal reference, to help remember exactly what is happening at any point in the program.

Chapter 4

LOOPS AND OTHER VARIATIONS



You've already learned that your computer follows a path, according to the program line numbers. In this chapter, we will talk about **loops** that cause your computer to repeat certain jobs and **detours** that send your system off on different paths.

Suppose you want the computer to follow different paths when you run the program on different occasions. The idea is similar to reaching a fork in a road. One day you might want to turn left at the fork. The next day you might want to turn right. A computer program can be designed to branch in much the same manner. You can create **detours** to cause the computer to follow different paths—by using GOTO, GOSUB, and IF–THEN commands. And you can create **loops** that cause the computer to repeat a particular section of a program over and over again—by using FOR–NEXT commands.

The ability to create loops and detours gives you tremendous flexibility. At one time, you might want to use a part of the program, and at another time, you might want to skip that part of the program and go to a different part of the same program. Using loops and detours, you can make one program follow a variety of paths, according to changes you make each time you run it.

To get a better idea of exactly what this means, let's first learn about FOR–NEXT loops.

GOING AROUND IN CIRCLES: FOR-NEXT STATEMENTS

FOR and NEXT statements create a repetitive loop. They cause a specific segment of a program to be repeated over and over again. How many times the segment loops is controlled by the values that you use in the FOR and NEXT statements.

It works like this. The FOR statement takes the form of a line like 10 FOR X=1 TO 10. The NEXT statement takes the form of a line like 100 NEXT X. The NEXT statement compares the value of variable X with the final value assigned in the FOR statement (in this case 10). If the value has not yet reached its final limit, the NEXT statement sends the computer back to the line containing the FOR statement. The value of the variable in the FOR statement increases by 1 each time the loop segment of the program is executed, unless otherwise specified with a STEP statement.

Using a STEP statement changes the value by which the FOR variable will increase. Instead of automatically increasing by 1 each time, the value will increase (increment) by the amount specified. A FOR statement with a STEP value assigned takes the form of a line like 10 FOR X=1 TO 10 STEP 2. Adding STEP 2 to this line tells the computer to increase the value of the variable by 2 each time the loop is executed.

Here's how a simple FOR-NEXT loop would look:

```
10 FOR X=1 TO 5
20
        section of program to be repeated
90
100 NEXT X
```

What happens with a program like this? The FOR statement begins by letting the variable X equal 1. Then the computer executes lines 20–90. When it reaches line 100, the computer increases the value of X by 1 and returns to line 10. Lines 20–90 are thus executed over and over—in this case, five times. When X reaches the value of 5, the loop ends. And if there are more lines in the program (after the NEXT statement), the computer goes on to execute them.

Here is an example of a small program containing a FOR-NEXT loop. Try it!

```
NEW
10 FOR X=1 TO 10
20 PRINT "SAY IT TEN TIMES!"
30 NEXT X
40 PRINT "DONE!"
50 END
```

Now run the program. The results of the program look like this on the screen:

```
SAY IT TEN TIMES!
SAY IT TEN TIMES!
SAY IT TEN TIMES!
SAY IT TEN TIMES!
SAY IT TEN TIMES!
SAY IT TEN TIMES!
SAY IT TEN TIMES!
SAY IT TEN TIMES!
SAY IT TEN TIMES!
SAY IT TEN TIMES!
DONE!
```

In line 10, when the FOR statement was executed the first time, the variable X was assigned a starting value of 1. In line 20, the computer printed the character string on the screen. In line 30, the NEXT statement was executed and the computer read the value of X. Since the value did not yet equal 10, the computer increased the value of X by 1 and went back to line 10. This process was repeated until the value of X was found to equal 10 when the computer reached line 30. Then the computer was allowed to continue to line 40 to print the word DONE! Line 50 told the computer to end the program.

Now try the program again, but add a STEP statement to it. To do this simply change line 10 by typing in this line:

```
10 FOR X=1 TO 10 STEP 2
```

Now run the program. The result? Because the STEP statement specified an increase of two each time, the computer printed the character string "SAY IT TEN TIMES!" only five times.

If the value specified after the STEP statement is negative, then the variable will decrease (decrement) by the specified amount. To see how this works, try the following program:

```
NEW

10 FOR A=5 TO 1 STEP -1

20 PRINT A

30 NEXT A

40 PRINT "FINISHED COUNTING
BACKWARDS!"

50 END
```

When you run the program, your computer responds this way:

```
5

4

3

2

1

FINISHED COUNTING BACKWARDS!
```

Now try a STEP statement using a different increment:

```
NEW

10 FOR B=1 TO 10 STEP 4

20 PRINT B

30 NEXT B

40 END
```

When you run the program, your computer responds this way:

```
1

5

9
```

The STEP counter increased by the value of 4 on each pass. By the fourth pass, B had reached a value of 13. When the NEXT statement found the value of B to be greater than the specified ending value of 10, it ended the loop without printing another value of B.

A Mistake To Avoid

Once you are inside the loop, do not change the value of the variable that you used in the FOR statement. For example, if the variable used in your FOR statement is C, do not use C as a variable again until after the line containing the NEXT statement.

Here is an example of a loop that causes a problem. If you run this program, you will find that it creates a continuous loop, one that you can break only with your CONTROL-C key combination or your BREAK key (depending on your computer). Before you try this program, make sure you know how to stop it. If all else fails, turn the computer off and start again. (Remember, however, that when you turn the computer off, you lose anything in its working memory that you have not stored onto cassette tape or disk.) If you want to try this program, go ahead.

```
10 FOR C=1 TO 50
20 PRINT "I'M COUNTING!"
30 LET C=44
40 LET D=12
50 LET E=18
60 NEXT C
```

This program creates a continuous loop because line 30 locks the value of C. At the start of the program, line 10 uses C as a variable in the FOR statement. C is assigned a starting value of 1 and a final value of 50. Then in line 30, C is assigned a new value of 44, and that happens every time the loop is executed. The NEXT statement will never find that C has reached or exceeded its final value of 50, because C will always equal 44 when the computer gets to the NEXT statement in line 60.

This sort of thing will confuse your computer every time. The best way to prevent it is to avoid using the same letter as a variable more than once in the same program.

Using For-Next Statements To Create Time Delays

FOR-NEXT statements can be useful in building time into your programs to let you read information displayed on your screen. When using FOR and NEXT for this purpose, you can easily put them together in the same line, as in the following example. (Don't type this on your computer now; it needs a program with it to make sense.)

```
10 FOR X=1 TO 1000:NEXT X
```

Since there are no program instructions between the FOR and NEXT statements, all the computer can do is perform 1000 FOR-NEXT loops and then proceed to the next line number. Since there is no change in the screen display while the computer is performing such loops, this creates the effect of a pause or delay in the program—at least as far as the user is concerned. Try the following program to see how this works.

```
NEW
10 PRINT "TIME TO READ SCREEN"
20 FOR A=1 TO 3000:NEXT A
30 PRINT "TIME'S UP!"
```

Now run the program. The final value of A in line 20 controls the amount of time that the system appears to pause. Changing the final value of the variable in the FOR statement changes the amount of time. Try changing the final value of A in line 20 to different values so that you get an idea of how long or short a pause you get from different values.

Nested Loops

A **nested loop** is a FOR–NEXT loop inside a larger FOR–NEXT loop. The following program makes use of a nested loop.

```
NEW
10 FOR A=1 TO 20
20 FOR X=1 TO 5
30 PRINT X
40 NEXT X
50 PRINT "HERE WE GO AGAIN!"
60 NEXT A
```

Now run the program. The results whisk by on the screen before you, and you may wonder if you did something wrong. You may even be tempted to use your CONTROL–C key combination to stop it. But don't stop it. Let the program run its course. Your screen will show something like this when the program is finished.

```
1
2
3
4
5
HERE WE GO AGAIN!
1
2
3
4
5
HERE WE GO AGAIN!
1
2
3
4
5
HERE WE GO AGAIN!
1
2
3
4
5
OK
```

What the program does is list the numerals 1, 2, 3, 4, and 5 and print the character string "HERE WE GO AGAIN!" twenty times. It does the whole thing twenty times because of the FOR–NEXT loop started in line 10; this is the **outer** loop (in this case, the A loop). The program printed the numbers because of the loop started in line 20; this is the inner or **nested** loop (in this case, the X loop). If you had a program consisting of only lines 20, 30, and 40 (the X loop), the computer would print the numerals once and then stop. But in this program, the outer loop causes the inner loop to repeat.

Notice that the X loop is entirely within the A loop—hence the term **nested loop.** Keeping the nested loop entirely within the outer loop is very important. Make these changes in the program by typing these two lines:

```
40 NEXT A
60 NEXT X
```

Now run the new program. You'll see that you don't get very far. List the new program (by typing LIST and pressing RETURN) so you can see what's wrong here.

```
10 FOR A=1 TO 20
20 FOR X=1 TO 5
30 PRINT X
40 NEXT A
50 PRINT "HERE WE GO AGAIN!"
60 NEXT X
```

CROSSED LOOPS

By switching the order of lines 40 and 60, you have created a crossing between the two loops, and neither loop can complete itself. Although you have both of the matching NEXT statements to go with your FOR statements, the NEXT statements are out of sequence. The X loop is not yet complete at line 40, yet you try to complete the A loop with a NEXT A statement.

For a nested loop to work correctly, no matter how many loops you use, there must be no crossing between them. In writing programs, you may find it useful to make a chart to show yourself that the loops are properly nested. Here's an example:

```
10 FOR X=1 TO 10
20 FOR Y=1 TO 10
30 FOR Z=1 TO 10
40 NEXT Z
50 NEXT Y
60 NEXT X
```

PROPERLY NESTED LOOPS

IF-THEN STATEMENTS

An IF–THEN statement causes the computer to evaluate an expression and then take a particular action if the expression is true. The expression being evaluated may involve either a numeric variable or a string variable. If the expression is **true**, then the computer will perform whatever action is specified in the THEN statement. If the expression is **false**, then the computer takes no further action at that line number. It simply continues to the next line in the sequence.

Here are a couple of examples of IF–THEN statements. (Do not type them in now; they need to be within a program to operate.)

```
IF X>=21 THEN PRINT "YOU'VE
WON THE GAME."

IF A$="YES" THEN GOTO 400
```

We can see how IF–THEN statements work by setting up a flight destination schedule for Treetop Airlines. We will use Dallas as a departure city, with Houston, Austin, and San Antonio as destination cities. We could use PRINT statements to display the entire flight schedule to all the cities, but then we would have to hunt through the list for the city we wanted to know about (a real problem if you had fifty or a hundred cities!). A more efficient way to do this is to use IF–THEN statements, like this:

```
NEW

5 PRINT "TREETOP AIRLINE
SCHEDULES"

10 PRINT "ENTER CITY
DESIRED."

20 INPUT A$

30 IF A$="HOUSTON" THEN GOTO
100

40 IF A$="AUSTIN" THEN GOTO
200

50 IF A$="SAN ANTONIO" THEN
GOTO 300

60 PRINT "WE DON'T FLY
THERE!":GOTO 10

100 PRINT "HOUSTON FLIGHTS ARE
1PM, 3PM, 5PM":END

200 PRINT "AUSTIN FLIGHTS ARE
2PM, 4PM, 7PM":END

300 PRINT "SAN ANTONIO FLIGHTS
ARE 2:30 PM, 6 PM, 8 PM":END
```

Now run the program. The computer tells you to "ENTER CITY DESIRED." Type in the name of the city you want to know about. The computer then compares the name you typed in to the cities in the IF–THEN statements until it finds a match. As long as no match is found, the computer ignores the rest of the line and continues to the next line number. When the computer finds a match, it carries out the THEN statement in that line. Each THEN statement tells the computer to go to the program line containing the flight times for that city—line 100, 200, or 300. If the computer does not find a match in lines 30, 40, or 50, it goes on to line 60. Line 60 lets you know that you have entered the name of a city that is not in the program. The GOTO command in line 60 sends the computer back to line 10 to give you a chance to enter another city name. Now let's take a closer look at GOTO statements.

DETOURS IN THE ROAD: GOTO AND GOSUB

There are ways to force the computer to stray from its normal path of following the line numbers in sequence. GOTO and GOSUB statements are two ways to do so. The names of the statements are fitting: they tell the computer to **go to** a different location or to **go** to a **sub**routine. A **subroutine** is a small part of a program; usually it is a part that is used often within the program. Using a subroutine allows the programmer to call on the same section of a program at different times (without writing it over and over).

GOTO STATEMENTS

A GOTO statement sends the computer to a specified line. It tells the computer to branch off to another part of the program, rather than going on to the next line number in sequence.

Branches in computer programs are like branches on trees. When you are following the main part and get to a branch, you have to make a choice. You can stay with the main part or go off on the branch. And if you follow the branch, you go away from the main part. Computer program branches often have commands built into them to get you back to the main part eventually.

You can use a GOTO statement in one of two ways: in an **unconditional branch** or in a **conditional branch.**

In an **unconditional branch,** the computer is simply instructed to go to another line. An unconditional branch is created with a simple GOTO statement like the following:

```
430 GOTO 30
```

The other type of GOTO statement is a **conditional branch.** For a conditional branch, a GOTO statement is used within an IF–THEN statement or in the same program line following an IF–THEN statement. The IF part of the statement tells the condition under which the computer should execute the rest of the line. Here are examples of conditional branch statements:

```
50 IF X1=10 THEN GOTO 400

60 IF X1=20 THEN PRINT "NO":
GOTO 500
```

Whether the computer goes to the line indicated depends on whether the condition in the IF statement is true. For example, when line 50 is executed, the computer evaluates the value of X1. If X1 equals 10, then the computer proceeds (or **branches,** in computer terminology) to line 400 and continues program execution from that point. We have set a specific condition for the branch—hence the term **conditional branch.** If X1 does not equal 10, the computer ignores the rest of this line and proceeds to the next line number in sequence.

To see both conditional and unconditional branches at work, try the following program:

```
NEW

10 PRINT "WHAT'S YOUR AGE?"

20 INPUT A

30 IF A>17 THEN PRINT "NORMAL
DRIVER'S LICENSE":GOTO 100

40 IF A>15 THEN PRINT "JUNIOR
LICENSE":GOTO 100

50 GOTO 200

100 PRINT "ISSUE APPLICABLE
LICENSE":END

200 PRINT "NOT ELIGIBLE FOR
LICENSE":END
```

When you run the program, the computer first responds this way:

```
WHAT'S YOUR AGE?
 ?
```

What it does next depends on your response to the INPUT statement in line 20. Lines 30 and 40 both contain GOTO statements, but these statements will be processed only if the conditions set by the IF–THEN statements in the lines are met. Because of this condition, these GOTO statements are **conditional** statements.

By comparison, line 50 is an **unconditional** GOTO statement. If the computer reads line 50, it will go to line 200; no conditions are included in that program line.

The program works like this. If the number you type in is greater than 17, then line 30 will be executed; the computer will print "NORMAL DRIVER'S LICENSE" and then go directly to line 100, which tells it to print "ISSUE APPLICABLE LICENSE" and end the program.

If the number is not greater than 17, the computer goes to line 40. The condition in line 40 is that the number must be greater than 15. But if the number were greater than 17, the computer would have gone on to line 100 and not read line 40 at all. Therefore, line 40 will be executed only if the number is 16 or 17, since these are the only numbers greater than 15, but not greater than 17.

If the number is not greater than 15, the computer goes on to read line 50, which sends it to line 200. Line 200 prints "NOT ELIGIBLE FOR LICENSE" and ends the program.

Here's one additional note about GOTO commands. When you are using a GOTO command within the THEN part of an IF–THEN statement, you can usually eliminate the word GOTO in most versions of BASIC. For example, the computer would read the following two program lines in the same way:

```
10 IF R=14 THEN GOTO 300

10 IF R=14 THEN 300
```

For all other uses of GOTO, however, you must include the word GOTO in the program line.

You will use GOTO statements often in programming, simply because you will often find it necessary to send the computer to another part of the program.

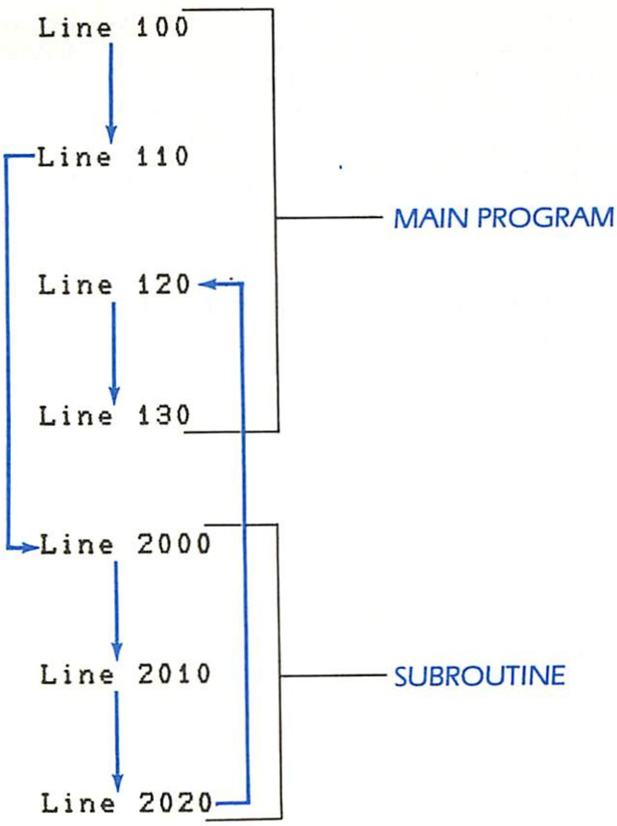
GOSUB AND RETURN STATEMENTS

The idea behind GOSUB is similar to that behind GOTO. However, the GOSUB statement causes the computer to remember what line number it left when the GOSUB command sent it to another line. The computer then continues to execute the program from that point until it sees a RETURN statement. When it does, it returns to the line immediately following the line containing the GOSUB command and continues executing program lines in sequence.

The entire operation is called a **subroutine.** The following demonstrates how a subroutine is used within a program:

```
NEW

100 PRINT "I'M LEAVING!"

110 GOSUB 2000

120 PRINT "I'M BACK!"

130 END

2000 FOR T=1 TO 2000:NEXT T

2010 PRINT "FAR AWAY IN
SUBROUTINE!"

2020 FOR T=1 TO 2000:NEXT
T:RETURN
```

The program took the route shown in this illustration:



Here are a few rules to follow in using subroutines:

1. The RETURN statement is a BASIC command and should not be confused with pressing the RETURN key to enter a direct command or program line. The RETURN command must **always** be typed in using the letter keys (except on a computer like the Timex Sinclair where you have specific keys to press for each command).

2. Be sure that you include a matching RETURN statement for every GOSUB that you use. If you send your computer to a subroutine and never give it a RETURN command, errors may develop later in the program that will be difficult to find.

3. When you put the subroutine after the main program, use an END statement at the end of the main program (the part which you leave to go to the subroutine). The END statement keeps the main program and the subroutine separate.

Here's an example of why you need to use an END statement at the end of the main program. Try it.

```
NEW
10 PRINT "ONE"
20 PRINT "TWO"
30 GOSUB 100
40 PRINT "FOUR"
50 PRINT "FIVE"
60 PRINT "DONE!"
100 PRINT "THREE"
105 RETURN
```

Now run the program. Oops! What happened? An extra THREE prints, out of sequence, then an error results, because the computer **crashed** into the subroutine. Stop the program (using the CONTROL and C keys or the BREAK key). Now add this line to the program:

```
90 END
```

Then run the program again. With the END statement, the program does what you probably thought it should do.

One additional note: If there are no subroutines after the main program, you usually do not have to use an END statement. END is optional in nearly all versions of the BASIC language.

TELEPHONE DIRECTORY PROGRAM— A REAL-LIFE APPLICATION

The following program makes use of FOR–NEXT loops, IF–THEN statements, and GOTO commands. It also contains one statement which we have not used yet: the CLEAR statement. CLEAR is a control operation. (Control operations are discussed in Chapter 5). But since we need the CLEAR statement for this program, let's briefly discuss it now.

The CLEAR statement resets all of a program's numeric variables to zero and all of the string variables to **null** (or zero) characters. The CLEAR statement is useful if you want to reset the program's variables without erasing the entire program (as the NEW command would).

Here's the program:

```
NEW
10 CLEAR
20 PRINT "PLEASE ENTER THE NAME DESIRED."
30 PRINT "TO SEE ALL NAMES IN THE DIRECTORY, ENTER THE WORD NAMES."
40 INPUT D$
50 IF D$="NAMES" THEN 700
100 FOR P=1 TO 10
110 READ A$,B$
120 IF A$=D$ THEN 200
130 GOTO 300
200 LET C$=A$
210 LET E$=B$
300 NEXT P
310 RESTORE
400 IF C$="" THEN 420
410 GOTO 500
420 PRINT "SORRY, THAT NAME IS NOT IN THE DIRECTORY."
```

```
430 GOTO 30
500 PRINT:PRINT "THE PHONE NUMBER YOU REQUESTED"
510 PRINT "IN THE NAME OF ";C$;" IS"
520 PRINT E$
530 PRINT "WOULD YOU LIKE ANOTHER NUMBER?"
540 INPUT K$:IF K$="YES" THEN 10
600 END
700 FOR Z=1 TO 10
710 READ A$,B$:PRINT A$
720 NEXT Z
730 PRINT:PRINT "PLEASE ENTER THE NAME DESIRED."
740 INPUT D$:RESTORE:GOTO 100
800 DATA "JOHNSON","555-1213"
810 DATA "TIME OF DAY","234-1232"
820 DATA "LEONARD","555-3479"
830 DATA "SMITH","211-3377"
840 DATA "EWING","311-9880"
850 DATA "FORD","411-6674"
860 DATA "ALLEN","611-3434"
870 DATA "OFFICE","911-3302"
880 DATA "WEATHER","314-5545"
890 DATA "DOCTOR","512-3323"
```

Now run the program by typing the word RUN and pressing RETURN. The first thing the computer does is display this message:

```
PLEASE ENTER THE NAME DESIRED.

TO SEE ALL NAMES IN THE
DIRECTORY, ENTER THE WORD
NAMES.

?
```

The computer wants you to tell it what to do next. It has executed lines 10, 20, and 30, and line 40 tells it to ask you for information.

So that we can explain what the program does as you go along, follow our directions this first time. Type in the word NAMES (and press RETURN).

The computer now displays a list of the names, something like this:

```
JOHNSON     TIME OF DAY
LEONARD     SMITH     EWING
FORD     ALLEN     OFFICE
WEATHER      DOCTOR

PLEASE ENTER THE NAME DESIRED.

?
```

When you gave the word NAMES as the value for D$, line 50 sent the computer to line 700 where it executed a loop. The loop told the computer to display the names in the DATA statements in lines 800 to 890.

When it finished the loop, the computer went on to line 730, which gave it two commands. The first command was to PRINT a blank line; the second was to PRINT a character string.

Line 740 tells the computer to ask you to input a new value for D$, to RESTORE the DATA values (so that they can be read again), and to go back to line 100. As soon as you respond to the input question mark and give the computer a new value for D$, the computer will complete the rest of the line 740 commands. This time, type the name ALLEN (and press RETURN). The computer then RESTORES the DATA values and returns to line 100.

The computer executes the loop between lines 100 and 300 until it finds the name and telephone number you requested. It reads each DATA statement until it finds the value that matches the value you assigned to D$—ALLEN. When the values match, the computer can break out of the loop; line 120 sends it to line 200 if the values match. (Then it does not go on to line 130, which would have sent it back to line 300 and through the loop again.)

In lines 200 and 210, the computer changes the names of A$ and B$ to C$ and E$, respectively. Line 310 again restores the DATA values (although that doesn't matter much right now). Since C$ has a value, line 400 has no effect. The computer gets to line 410 which sends it to line 500.

Lines 500 through 540 print a blank line first and then the message that follows:

```
THE PHONE NUMBER YOU REQUESTED
IN THE NAME OF ALLEN IS
611-3434

WOULD YOU LIKE ANOTHER NUMBER?

?
```

Line 540 causes the computer to ask you for input. If you type YES (and press RETURN), the computer will return to line 10 and execute the program again. (In line 10, the CLEAR command resets all the program's variables to zero or null characters so that you can use the program again, assigning new values to the variables.) If you type anything other than YES in response to line 540, the computer will continue to line 600 and end the program.

If you would like to replace the sample names and numbers in the DATA statements with some names and numbers of your own, here's how (it's easy!):

1. Change the final values for the variables in both FOR–NEXT loops (the one starting in line 100 and the one starting in line 700) to match the number of names in **your** list. If you have 14 names instead of 10, lines 100 and 700 should read like this:

```
100 FOR P=1 TO 14
700 FOR Z=1 TO 14
```

2. Use a new line number (after line 890) for each extra name and phone number.

If you want to store this program on cassette or disk for future use, turn to the "SAVE" section in Chapter 5, "Control Operations."

REVIEW

In this chapter, you learned that...

• The FOR–NEXT statement creates a repetitive loop. How many times the loop repeats depends on the values specified within the FOR statement.

• The IF–THEN statement causes the computer to take a particular action under certain conditions. If the expression in the IF part of the statement is true, the computer executes the THEN part of the statement. If the expression is false, the computer ignores the rest of the line and proceeds directly to the next program line.

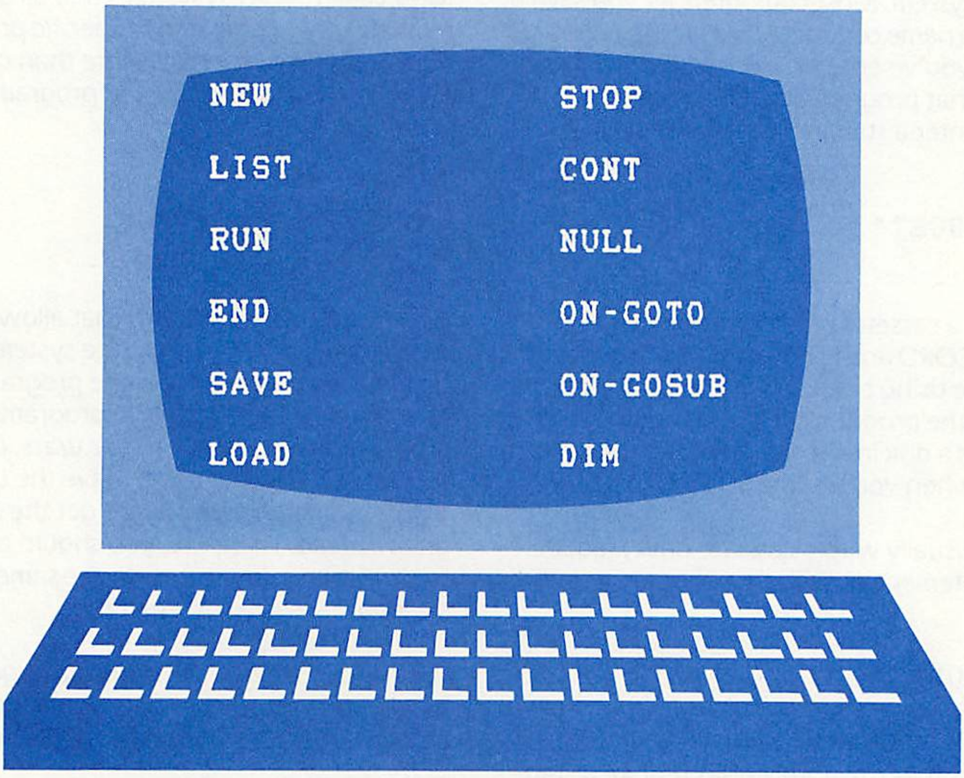
• The GOTO statement sends the computer to a specific line number. The GOTO statement can be used in one of two ways: in **conditional** statements or **unconditional** statements.

• The GOSUB statement also sends the computer to a specific line number. In the case of the GOSUB statement, the line number that the computer goes to is the start of a **subroutine.** When the subroutine is completed, a RETURN statement is used. The RETURN statement sends the computer back to the program line immediately following the line containing the GOSUB command that sent it to the subroutine.

• The CLEAR statement is a control operation which resets all program variables at zero or null.

Chapter 5

CONTROL OPERATIONS



Control operations can be considered **controllers** of the computer. They give the BASIC language its power. The names of the control operations always provide clues to what they do.

If you examine any extensive BASIC program, you'll find that the bulk of the program is made up of control operations and arithmetic operations. Input/output operations are only necessary when humans need to interact with the computer, and many library operations are used only for advanced mathematical or scientific functions.

The extensive programs you may have seen elsewhere (or even later in this book) may appear far more complex than any program you feel you could ever write, but keep this in mind: any large program is actually a collection of several much smaller programs. By taking a large task a step at a time, and writing and debugging small pieces of programs, you'll find that longer programs just take a little more time than short programs.

We have already discussed a few control operations: NEW, LIST, RUN, END, FOR and NEXT, GOTO, GOSUB and RETURN, IF–THEN, CLEAR, and RESTORE. In this chapter, we will discuss some other control operations: SAVE, LOAD, STOP, CONT, NULL, ON–GOTO, ON–GOSUB, and DIM. We will also discuss RUN and END in more detail than we did earlier. Of all of these operations, DIM is the most complicated and will require the most explanation.

SAVE

The SAVE command stores information from the computer's memory to a cassette tape or floppy disk. You will use the SAVE command to avoid having to type the same program into the computer each time that you want to use it.

The exact way the SAVE command is used varies from computer to computer, so you should check your owner's manual to be sure you use the correct command for your system. Most computers let you save a program using a name of your choice for the program. For instance, if you've entered a program, and you want to name that program BUDGET, most computers let you enter a statement similar to this:

```
SAVE "BUDGET"
```

If you are using a cassette recorder, you must also depress the RECORD and PLAY keys to save programs or data. If you're using a disk drive, the system automatically saves the program on disk (assuming that you have placed a disk in the disk drive and turned on the disk drive) when you use the SAVE command.

Some systems (usually with a cassette drive) require you to use a statement like this:

```
CSAVE "BUDGET"
```

Some computers that use floppy disks require a statement like this:

```
SAVE DSK1 "BUDGET"
```

Because of these minor command variations, check your owner's manual to see how to save a program on your model of computer.

LOAD

The LOAD statement (CLOAD for systems using cassette tapes for storage) commands the computer to load a program from the cassette or disk drive into the active memory of the computer. The LOAD statement turns on the computer's **load flag,** a sort of electronic switch that transfers the computer's capability to receive data from the keyboard to the cassette input cable or floppy disk operating system.

Most cassette-based systems and all disk-based systems allow you to load one specific program from a tape or disk that contains more than one program. This is done by specifying the program desired with a statement like this:

```
LOAD "BACKGAMMON"
```

On cassette-based systems that allow the retrieval of specific programs by name, the system usually reads only the first two letters of the program name. This means that if you saved two programs, one named **Star Trek** and one named **Star Wars**, on the same side of a cassette tape and then gave the LOAD "STAR WARS" command, you might get the **Star Trek** program instead. Therefore, you should avoid putting two programs that start with the same two letters on the same side of a cassette tape.

All disk-based systems examine a larger number of letters, so if you are working with floppy disks you will always get exactly the program you request.

The actual methods of loading programs from a tape or disk (like the actual methods of saving programs) vary from system to system, so check your operator's manual for details on your specific model of computer.

RUN

The RUN command causes the computer to execute the program currently in its active memory. The RUN command also automatically resets all variable values to zero.

If you type the RUN command without specifying a line number, the computer begins executing from the first line of the program. If you specify a line number, the execution begins with the specified line number. For example, if you type the command RUN 160, the computer begins at line 160 of the program currently stored in memory (instead of at the first program line number).

You can also use the RUN command within a program. With disk-based systems, you can use the command in a menu to select other programs on the same floppy disk. A **menu** is simply a list of choices; you select the item you want when the program runs. Here is an example of how this might work in a program. It takes more than what we have given you here to run the program effectively, but this will give you the idea.

```
100 PRINT "CHOOSE PAYROLL (1),
LEDGER (2), OR INVENTORY (3)"

110 INPUT A

120 IF A=1 THEN GOTO 200

130 IF A=2 THEN GOTO 300

140 IF A=3 THEN GOTO 400

200 RUN "PAYROLL"

300 RUN "LEDGER"

400 RUN "INVENTORY"
```

Line 100 contains a menu or list of choices. Line 110 asks the user to type in a choice number. Depending on the choice number, line 120, 130, or 140 sends the computer to the appropriate RUN command in line 200, 300, or 400.

STOP

A STOP statement interrupts the execution of a program. When a STOP statement is encountered, the computer halts and displays a BREAK message. If the program stopped at line 100, for example, the message would look like this:

```
BREAK IN 100
```

CONT (Continue)

The CONT command continues the execution of the program that was running when halted with the STOP command. The following program demonstrates the use of the STOP and CONT commands:

```
NEW

10 PRINT "ONE,TWO"

20 PRINT "THREE,FOUR":STOP

30 PRINT "FIVE,SIX"

40 PRINT "SEVEN,EIGHT"

50 END
```

When you run the program, the computer halts after it prints the word FOUR on the screen and displays this message:

```
BREAK IN 20
```

Now type

```
CONT
```

Press the RETURN key, and the computer continues executing the rest of the program.

The combination of STOP and CONT commands is useful if you want to halt a program for some reason—for example, to give you time to read the screen at your leisure. Using FOR–NEXT statements lets you stop the screen for set periods of time, but the STOP and CONT commands let you take as long as you need to read the screen each time.

CONT and STOP commands are also useful when you need to **debug** (find errors in) programs. If you are not sure which line in a program is causing the problem, you can halt the program at places where you suspect the problem might be.

WARNING: If you use STOP to go into the program and make changes, you cannot use the CONT command to restart the program once you have made changes in the program. If you try to do so, you will see a CONTINUE ERROR message on the screen. Once you have used STOP and made changes, you have to run the program again from the beginning to see whether your changes have fixed the problem.

END

An END statement completes the execution of a program in a normal manner. It is usually the last statement in a program. If you are using a subroutine, however, you should put the END statement between the main program and the subroutine. In such cases, END is used to prevent the computer from going past the end of the main program into the subroutine.

The END command differs from a STOP command in that once the END command is executed, it cannot be reversed. (You can, of course, run the program again from the beginning.)

NULL

A NULL command inserts a specific number of **nulls** (dead spaces) in a program line. In executing a program line like this

```
180 PRINT NULL 4; "BOB
SMITH,222 CHESTNUT LANE"
```

the computer would execute four nulls before executing the PRINT statement.

Nulls are useful when a mechanical peripheral (usually a slow printer) is attached to your system and there is a chance that the computer might outrun the peripheral. This possibility exists with some printers, such as Selectric and Teletype converted printer/typewriters. If the computer is sending data to the printer faster than the printer can print it, nulls can create delays that give the printer time to catch up. If your printer was designed to be used with a personal computer (or if you don't have a printer), this should not be a problem and you won't have to use nulls.

ON-GOTO

An ON-GOTO statement is a multiple GOTO statement. The value of the variable used in the ON part of the statement determines where the program will branch. Here's a sample of how ON-GOTO statements work.

```
ON X GOTO 400, 500, 600, 700
```

This is what happens when the system executes this line:

If X=1, the computer branches to the **first** line number listed in the GOTO part of the statement, in this case line 400.

If X=2, the computer branches to the **second** line number listed, line 500.

If X=3, the computer branches to the **third** line number, line 600.

If X=4, the computer branches to the **fourth** line number, line 700.

If X does not equal 1, 2, 3, or 4, the computer simply goes on to the next line number (since there are only four choices listed in the GOTO part of the statement).

The ON-GOTO statement can save space within a program by reading a numeric value and sending the computer to another line number, depending on what the numeric value is.

Let's see how this savings might work. In Chapter 4, we used a sample schedule from Treetop Airlines. We will use part of that program again and then make some changes to shorten it while getting the same outcome. Try this (and compare it with the airline schedule program in Chapter 4):

```
NEW

5 PRINT "TREETOP AIRLINE
SCHEDULES"

10 PRINT "FOR HOUSTON (1),
AUSTIN (2), OR SAN ANTONIO
(3), ENTER CHOICE NUMBER."

20 INPUT A

30 ON A GOTO 100, 200, 300

60 PRINT "INVALID CHOICE
NUMBER!":GOTO 10

100 PRINT "HOUSTON FLIGHTS ARE
1PM, 3PM, 5PM":END

200 PRINT "AUSTIN FLIGHTS ARE
2PM, 4PM, 7PM":END

300 PRINT "SAN ANTONIO FLIGHTS
ARE 2:30PM, 6PM, 8PM":END
```

In the program in Chapter 4, we used IF–THEN statements to deal with the choice of cities the user might type in. If we had many cities on the list, it would take a large number of IF–THEN statements to accommodate the choices. By using the menu technique (in line 10) and the ON–GOTO statement (in line 30), you can use fewer program lines and less of your computer's memory to store the program.

ON–GOSUB

The ON–GOSUB statement is similar to the ON–GOTO statement. The difference is that the GOSUB part of the statement sends the computer to the appropriate subroutine, rather than to the appropriate line number, as the GOTO part of the ON–GOTO statement would.

DIM

The DIM statement is used to set up a **dimensional array.** An **array** is simply an amount of space the computer sets aside to handle variables. Most personal computers normally allot ten spaces for each variable. If, in special cases, you want to enlarge that space (to use a larger array), you can specify the size of the array you want by using a DIM statement. It is good BASIC grammar to put all of your DIM statements at the beginning of the program, although it is not necessary to do so.

The DIM statement is made up of the basic keyword DIM followed by the name of the array whose size you want to specify and then a set of parentheses enclosing the amount of space you want to set aside for the array. What the DIM statement does is create a dimension in the computer's memory.

Normal variables occupy only a one-dimensional array: they are given one row with ten columns in it in the computer's memory. Using the DIM command, you can set up a one-dimensional array with as many columns as you like. The command to set up a one-dimensional array of twenty columns, with the variable name A, would be DIM A(20).

A two-dimensional array occupies two dimensions in the computer's memory—it has more than one row, and as many columns as you like. The command to set up a two-dimensional array of five rows and ten columns, with the variable name B, would be DIM B (5,10). Once you have established the size of the expanded space that you need, you can store various values inside those dimensions. We will explain what all this means in a little more detail in the examples that follow.

So far in this book we've stayed away from programs that would require us to store a large amount of data in memory. We've kept things simple. We have been working with a limited number of variable names in our programs. If we had many values to represent within a program, we could let AA equal the first variable, AB equal the second, AC equal the third, and so on. But if we were using a large number of variables, keeping track of which variable represents which value could be very confusing. The DIM command makes it a little easier to keep track of multiple variables. To see how this works, we will develop a Retirement Investment Analysis program.

RETIREMENT INVESTMENT ANALYSIS PROGRAM

Let's say that you have decided to save a specific amount each month in a retirement account. You could use 12 different variables to represent the amount you saved each month over a period of one year, like this:

JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC
A1	A2	A3	A4	A5	A6	A7	A8	A9	B1	B2	B3

At first, this method of assigning variables may seem perfectly workable. But what would happen if you wanted to track the results of more than one year of savings? For two years, you would need twice as many variables. For thirty years, you would have to name so many individual variables that the program probably wouldn't fit in the computer's memory.

Instead, you can let the same variable represent the entire group of values! You can do this by using dimensioned arrays. In many ways, the use of dimensioned arrays makes the program much easier to work with. You can consider one year as a one-dimensional set of values, like this:

← DIMENSION: 12 columns →

JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC

Then you can consider five years as a second dimension, like this:

1984

1985

1986 DIMENSION: 5 rows

1987

1988

The amazing fact about the DIM command is that you can use only one variable (perhaps A) to represent either the one-dimensional set of values or the two-dimensional set of values. To do this, you use parentheses to define the individual values as **subsets** (individual elements) of A. To see what this means, look at the charts that follow.

Let's take the one-dimensional sets of values first. We can let each month be represented like this:

JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC
A (1)	A (2)	A (3)	A (4)	A (5)	A (6)	A (7)	A (8)	A (9)	A (10)	A (11)	A (12)

We can represent the two-dimensional set of values (for the five-year span) like this:

	JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC
1984	A (1, 1)	A (1, 2)	A (1, 3)	A (1, 4)	A (1, 5)	A (1, 6)	A (1, 7)	A (1, 8)	A (1, 9)	A (1, 10)	A (1, 11)	A (1, 12)
1985	A (2, 1)	A (2, 2)	A (2, 3)	A (2, 4)	A (2, 5)	A (2, 6)	A (2, 7)	A (2, 8)	A (2, 9)	A (2, 10)	A (2, 11)	A (2, 12)
1986	A (3, 1)	A (3, 2)	A (3, 3)	A (3, 4)	A (3, 5)	A (3, 6)	A (3, 7)	A (3, 8)	A (3, 9)	A (3, 10)	A (3, 11)	A (3, 12)
1987	A (4, 1)	A (4, 2)	A (4, 3)	A (4, 4)	A (4, 5)	A (4, 6)	A (4, 7)	A (4, 8)	A (4, 9)	A (4, 10)	A (4, 11)	A (4, 12)
1988	A (5, 1)	A (5, 2)	A (5, 3)	A (5, 4)	A (5, 5)	A (5, 6)	A (5, 7)	A (5, 8)	A (5, 9)	A (5, 10)	A (5, 11)	A (5, 12)

The numbers in the parentheses tell you which year and month you are defining. Since January of 1984 is the first year and the first month, it is represented by A (1, 1). Since December of 1988 is the fifth year and the twelfth month, it is represented by A (5, 12).

Throughout all of this, it's important to realize that there is a significant difference between this statement

```
A1=475.80
```

and this statement

```
A(1)=475.80
```

The first statement merely tells the computer that the variable A1 is assigned the value 475.80. The second statement tells the computer that a one-dimensional array of variables called A exists; the 1 inside the parentheses tells the computer that the first subset inside that array is assigned the value 475.80.

Now you may think at this point that we are not saving anything at all by using this method of naming values; in fact the whole method seems rather complex. But the savings in time and in effort come when you consider that variables can be used **inside** the array parentheses. In other words, if M=1 and Y=1, then A(M, Y) is the same as A(1, 1). Using this fact, we can combine it with FOR–NEXT statements to set up a repetitive loop that will rapidly change the values within the parentheses.

Let's try this in a Retirement Investment Analysis program:

```
5 Y=1:M=1
10 INPUT "ENTER AMOUNT SAVED PER MONTH";X
20 DIM A(5,12): REM CREATE SPACE FOR 5 BY 12 ARRAY
30 FOR Y=1 TO 5: REM YEARS 1 TO 5
35 PRINT "YEAR NUMBER ";Y
40 FOR M=1 TO 12: REM MONTHS 1 TO 12
50 LET A(Y,M)=X
60 LET T=T+A(Y,M): REM TOTAL SAVED SO FAR
70 PRINT "MONTH NUMBER ";M;" TOTAL IS ";T
80 NEXT M: REM ADVANCE MONTH BY ONE
90 FOR Z=1 TO 2000: NEXT Z: REM TIME TO READ SCREEN
100 M=1: REM SET MONTH BACK TO JANUARY
110 NEXT Y: REM ADVANCE YEAR BY ONE
120 PRINT "FIVE YEAR ANALYSIS COMPLETE."
```

Now run the program. When the computer displays this message

```
ENTER AMOUNT SAVED PER MONTH
?
```

select the amount per month you want to put into this retirement account. Enter that number (don't use a dollar sign with the number).

The program will flash by on the screen as it is being executed. The computer is printing the amount of money accumulating by the year and the month, assuming that you invest the same amount per month. At the end, the screen will say

```
FIVE YEAR ANALYSIS COMPLETE.
```

and the final amount given (under the fifth year, the twelfth month) will be how much money you would have in the account at the end of the five years.

This program demonstrates the power of dimensioned arrays by setting up a two-dimensional array and storing each month's savings in an individual subset of the array. Line 20 is the crucial part of this setup. Line 20 contains our DIM statement, which warns the computer that we are about to take a section of its memory and devote it to a two-dimensional array.

Line 30 is the FOR statement that creates the Y loop, which is executed five times, once for each year. Line 40 is the FOR statement that creates the M loop, which is nested inside the Y loop. The M loop is executed 12 times, once for each month. The use of the two repeating loops together results in the core of the program, lines 50 through 70, being executed 60 times over the projected five-year period.

The values of M and Y (which change constantly after having been given starting values in line 5) result in the values provided for by the two-dimensional array created by line 50. Lines 60 and 70 calculate and print the running total.

Those are the basics of how this program runs. But now let's add the compounding of interest, which will make our Retirement Investment Analysis program more useful. Add the following lines to the program already in memory by simply typing them in (press RETURN after each program line):

```
12 PRINT "ENTER INTEREST RATE AS FOLLOWS--6.75%, ENTER .0675, ETC."
14 INPUT "YEARLY INTEREST RATE";B
62 LET I=(B/12)*T: REM CALCULATE MONTH'S INTEREST
64 LET T=T+I: REM ADD INTEREST TO TOTAL
```

Now run the program again. This time you have to enter not only the amount you plan to invest per month but also the amount of interest you will be earning on the money. Once you have done so, the computer executes a procedure similar to what it did before. If your computer has enough memory space, it will complete the program and display a final answer (under the twelfth month of the fifth year). If your computer doesn't have a large enough memory, the computer will not be able to finish the program, and you will see some sort of error message on the screen. Either way, you will see the effect that the increasingly large arrays have on memory size.

If you want to change the number of years being analyzed in the program, you must change lines 20 and 30. In line 20, the value of 5 in the DIM statement must be changed to the value of the final year desired. In line 30, the final value of the FOR statement must also match the new value.

This program gives you the opportunity to find out how your total savings would vary, depending on the amount you save per month, the interest rate, and the number of years you save. You might want to try changing these amounts a few times to see the results of various investments.

If you would like to store this program for future use, use the SAVE command to store it on cassette or disk.

Once you have the program safely stored (or if you don't mind losing the program), you might try an experiment that shows how important the DIM statement is in this program. Try deleting line 20 by typing 20 and pressing the RETURN key. Then RUN the program without the DIM statement. You will see that it works, up to a point. But you wind up with something like this on your screen:

```
MONTH #10 TOTAL IS 2051.01

?SUBSCRIPT OUT OF RANGE IN 50
```

This highlights the importance of the DIM statement. The use of dimensioned arrays sets aside all the memory space needed to fit the entire array. This occurs whether you use all of the array or not. The larger the array, the more memory required.

REVIEW

In this chapter, you learned that...

- Control operations are a group of complex BASIC operations that control the computer during the execution of programs.

- The SAVE command (CSAVE for cassette drives) lets you store programs on cassette tape or floppy disk.

- The LOAD command (CLOAD for cassette drives) lets you load a single program from a tape or disk.

- The RUN command can be used by itself to run a whole program or with a line number to specify that the program should run only from the specified line on.

- The STOP and CONT commands work together to temporarily halt the program and then make it continue running.

- The END command completes the execution of a program in a normal manner.

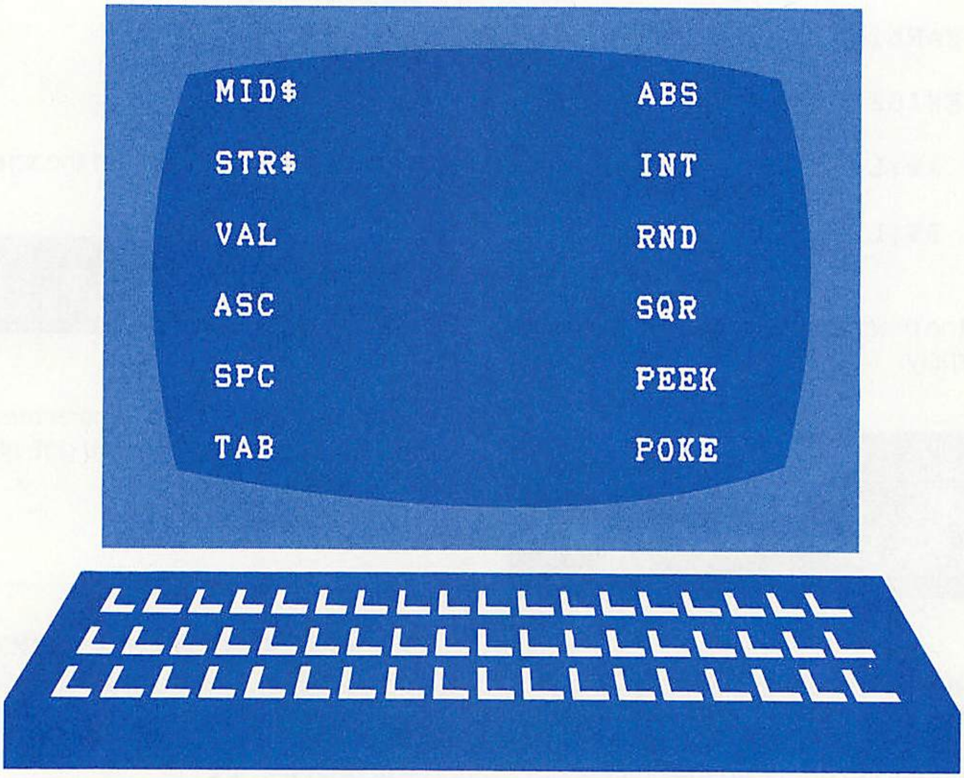
- The NULL command inserts nulls (dead spaces) in a program.

- The ON-GOTO and ON-GOSUB commands send the computer to particular program locations, depending on the value of the variable in the ON part of the statement.

- The DIM command tells the computer to set aside a specific amount of its memory space to hold data to be used in the program.

Chapter 6

LIBRARY OPERATIONS



Library operations have two general types of functions: they work with character strings, and they perform mathematical operations. Their names, like other BASIC command names, generally indicate their functions. They are called library operations because they make up a library of routines (like small programs which respond to single commands) that are permanently stored in your computer. These routines make it easier to perform certain complex functions, but every one of them could be duplicated by a BASIC program. Like the ON–GOTO and ON–GOSUB commands, they make writing programs easier once you know how to use them, and they save memory space.

There are some library operations that you may use often, even if you're not involved with advanced mathematics. Those are the library operations that we'll discuss, starting with the ones involving character strings.

INSIDE STRINGS

You already know that a group of characters can be stored in your computer as a character string. One group of powerful library operations allows you to reach inside a character string and pull out specific characters (or values based on those characters). These operations have the following names: LEN, LEFT$, MID$, RIGHT$, STR$, and VAL. Let's look at each of these operations.

LEN

The LEN operation causes the computer to count the number of characters (including spaces) in a specified character string. LEN is an abbreviation for **LENgth** of a character string. The actual command to the computer to get it to count the number of characters in the character string A$ would be LEN(A$). Try this short program to see how the LEN operation works:

```
NEW
10 A$="CHARLIE"
20 B$="DENISE"
30 PRINT A$;LEN(A$)
40 PRINT B$;LEN(B$)
```

When you run the program, the computer's response is this screen display:

```
CHARLIE 7
DENISE 6
```

The computer printed the contents of the character strings in response to the PRINT commands in lines 30 and 40 (although you wouldn't always have it do that). It also counted and displayed the number of characters in each string in response to the LEN commands in lines 30 and 40. In some cases, you might not want to have it display the number of characters in the string. You might instead use LEN to assign a value to a variable with a command like X=LEN(A$).

LEFT$

The LEFT$ operation identifies the specified number of characters at the **left** end (the beginning) of the character string. The number of characters wanted may be specified by using either a constant or a variable. The form for a command to identify the first five characters in string A$ would be LEFT$ (A$,5).

To see how this works, try these direct commands:

```
A$="WHYNOT"
PRINT LEFT$(A$,3)
```

The computer displays this on the screen:

```
WHY
```

You asked for the first three characters of the character string A$, and that's what you got. Now try a program example:

```
NEW
10 A$="EASY BASIC COMMANDS"
20 B$=LEFT$(A$,4)
50 PRINT B$
```

Now run the program. The computer responds by printing the first four letters of the character string A$, because that is the value assigned to B$ in line 20.

```
EASY
```

MID$

The MID$ operation identifies the specified number of characters in the **middle** of the character string. The form for a command to identify a section of four characters, starting with the fifth character in the character string A$, would be MID$ (A$,4,5). You can use either a constant or a variable to specify the number of characters to be identified and the number of the character where the section begins.

Add these lines to the program you just used:

```
30 C$=MID$(A$,5,6)
60 PRINT C$
```

Then use the LIST command to display the program on the screen. The program now looks like this:

```
10 A$="EASY BASIC COMMANDS"
20 B$=LEFT$(A$,4)
30 C$=MID$(A$,5,6)
50 PRINT B$
60 PRINT C$
```

Now run the program, and the computer gives you this response:

```
EASY
BASIC
```

As before, you asked the computer to identify and print the first four characters in A$. The new lines asked it to identify and then print a section of five characters, starting with the sixth character in A$.

RIGHT$

The RIGHT$ operation identifies the specified number of characters at the **right** end (the end) of the character string. The number of characters to be identified may be specified by using either a constant or a variable. The form for a command to identify the last five characters in string A$ would be RIGHT$ (A$,5).

Let's add two more lines to the program we've been working with to see the RIGHT$ operation in action:

```
40 D$=RIGHT$(A$,8)
70 PRINT D$
```

Now LIST the program on the screen again. It looks like this:

```
10 A$="EASY BASIC COMMANDS"
20 B$=LEFT$(A$,4)
30 C$=MID$(A$,5,6)
40 D$=RIGHT$(A$,8)
50 PRINT B$
60 PRINT C$
70 PRINT D$
```

Now run the program. The computer gives you this response:

```
EASY
BASIC
COMMANDS
```

You asked the computer to do the same things that it did before. In the new lines, you asked it to identify and print the last eight characters in A$.

STR$

The STR$ operation converts a numeric value into a character **string.** Why might you want to do that?

Remember that these two statements are **not** the same:

```
10 A=29.95

20 A$="29.95"
```

The value in line 10 can be used in any numeric calculation you might want to perform, but the value in line 20 cannot. The computer considers the numeral in line 20 as a string of characters—not as a numeric value.

There will be times when you have a numeric value that you want to display as a character string. The form for a command to change a value of the numeric variable A to a string value would be STR$ (A). Here's a case in which you need to do just that:

```
NEW

10 PRINT "ENTER THE COST OF
THE ITEM"

20 INPUT A

30 LET A$=STR$(A)

40 LET B$="$":REM FOR DOLLAR
SIGN

50 PRINT "THE COST OF THE ITEM
IS"

60 PRINT B$+A$
```

When you run the program, the INPUT statement in line 20 asks you to supply a numeric value. The STR$ operation in line 30 converts that numeric value into a character string, which can then be used in line 60.

VAL

The VAL operation is the opposite of the STR$ operation. The VAL operation converts a character string into a numeric **value.** The VAL operation can be very useful for pulling numeric data out of a character string. But it is much more complex for the computer than the STR$ operation, because the character string which the computer is asked to convert may contain letters, numerals, or combinations of both. For this reason, the computer follows some very specific rules when performing VAL operations.

If there are no numeric characters in the character string, or if the character string begins with letters rather than numerals, the VAL operation will assign a value of zero. If the string contains both numerals and letters, the VAL operation will read only the first numeral (or first combination of numerals).

The form for a command to change the string value of character string A$ to a numeric value would be VAL (A$). Here is a simple example of a VAL operation in a program:

```
NEW

10 A$="1000 TIMES, 2000 TIMES."

20 PRINT VAL(A$)
```

When you run the program, the computer displays this response:

```
1000
```

The VAL operation gave you the first combination of numerals in the character string A$.

HOW YOUR COMPUTER HANDLES CHARACTERS

Your computer performs all of its calculations by translating whatever you type on the keyboard (or whatever it receives through a telephone modem) into numbers. Each key on your keyboard can be represented by a specific number. The group of numbers that represents all the characters on the keyboard is known as the **ASCII Table.** The word ASCII (pronounced **ask-ee**) is an acronym for **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange. The entire scheme of ASCII codes allows one brand of computer to talk to another brand of computer over telephone lines. They communicate by using the ASCII codes.

We could show you a table displaying the ASCII value for every character on your keyboard. But it would be more fun to let your computer do that for you. Try this program for an interesting display:

```
NEW

10 FOR X=0 TO 255

20 PRINT "CHARACTER IS ";CHR$(X)

30 PRINT "ASCII VALUE IS ";X:
PRINT

40 FOR T=1 TO 500: NEXT T: REM
TIME DELAY TO READ SCREEN

50 NEXT X
```

The program causes your computer to print every value assigned within the normal ASCII range of 0 to 255. This program will not do exactly what you might expect it to do. You may get blank spaces often, and the screen display may be scrambled. These things are all fine, but we can't tell you exactly what they will be because they are totally different from computer to computer.

The uppercase and lowercase letters, the numbers, punctuation marks, and other keyboard symbols are the same or very similar from computer to computer, but those values occupy only part of the range displayed by this program. The other values displayed are assigned by the manufacturer of your particular computer. On some computers, various graphics patterns correspond to various ASCII values. On other computers, you will get special control codes, or blank spaces. The exact ASCII table you get, therefore, will depend on your specific model of computer.

CHR$

The CHR$ operation (performed in line 20 of the program we just ran) provides the character that is equivalent to the ASCII value specified. The operation name CHR$ is an abbreviated form of **character.** The form for a command to identify the character equivalent to ASCII value 78 would be CHR$ (78). The value to be identified can be any number between 0 and 255. The program we just ran went through variables with each of those values (using a FOR–NEXT loop).

You can also see the character equivalents of individual ASCII values. Try it, using this direct command:

```
PRINT CHR$(78)
```

Your computer displays the character N on the screen.

ASC

The ASC operation is just the opposite of the CHR$ operation. ASC causes the computer to identify a numeric value representing the ASCII equivalent of the first character in the character string. The operation name ASC is an abbreviated form of **ASCII.**

To find out what the ASCII value of the character M is, for example, you can use this direct command:

```
PRINT ASC("M")
```

The computer tells you that the ASCII value of M is 77. (Note that the computer gives you the ASCII number in decimal form, even though the computer actually operates with numbers in binary form.)

POSITIONING CHARACTERS ON THE SCREEN

The SPC and TAB functions are used to move the cursor to different positions on the screen.

SPC or PRINT AT

The SPC operation (called PRINT AT on some computers) moves the cursor to another position on the screen. It creates **space** (hence its name) between the left side of the screen, where your computer normally starts to display text, and where the text actually shows on the screen. You specify the number of spaces you want by placing that number in parentheses after the SPC command. The command takes the form SPC (X).

You can also use SPC to create controlled spacing between words and numbers in your PRINT statements.

Try this direct command to see how SPC works.

```
PRINT SPC(12) "HI!"
```

The computer displays this response:

```
HI!
```

If your computer gives you an error message instead, try

```
PRINT AT (12)"HI!"
```

TAB

The TAB operation also moves your cursor a specified number of spaces from the left side of the screen. It works like the TAB key on a typewriter. TAB is useful for preparing charts with columns of numbers, or for anything else that requires standard but indented spacing.

Try this example to see how it works:

```
PRINT TAB(5)2;TAB(10)4;TAB(15)6
```

The computer should give you this evenly spaced response:

```
    2    4    6
```

The number in parentheses after the TAB command tells the computer the exact column in which to begin displaying the word or numeral. The first TAB tells the computer to print the numeral 2 in the fifth column from the left side of the screen. The second TAB tells it to begin in the tenth column. The third TAB tells it to begin in the fifteenth column.

On some computer keyboards, there is a TAB key, which means that you may not need the TAB command. If you have a TAB key on your computer, see your operator's manual for instructions on setting tabs.

AND NOW FOR THOSE MATHEMATICAL OPERATIONS!

The remaining library operations are used primarily in advanced mathematics; for that reason, we will explain only what they do (not how or why or what that might mean). Some of these functions will be familiar to you if you have studied algebra or trigonometry.

ABS

The ABS operation identifies the **absolute value** of the specified number (or variable value). To find the absolute value of −5, enter the following:

```
PRINT ABS(-5)
```

The computer responds like this:

```
5
```

ATN

The ATN operation identifies the **arctangent** of the value of the angle when the specified number (or variable value) is the tangent of the angle. The ATN value is expressed in radians; for conversion to degrees, multiply it by 57.29578.

To find the arctangent of the value of the angle when the tangent of the angle is 29, enter the following:

```
PRINT ATN(29)
```

The computer responds like this:

```
1.53633
```

COS

The COS operation identifies the **cosine** of the number (or variable value) when the number is expressed in radians. To find the cosine of 45, enter the following:

```
PRINT COS(45)
```

The computer responds like this:

```
.525324
```

EXP

The EXP operation identifies the value of e raised to the power of the number (or variable value) specified, as represented by the mathematical expression e^x. The symbol e represents the base of the natural system of logarithms. The name EXP is an abbreviation for **exponent.** The EXP operation is the opposite of the LOG operation. To find the value of e^1, enter the following:

```
PRINT EXP(1)
```

The computer responds like this:

```
2.71828
```

INT

The INT operation rounds a specified number (or variable value) to the nearest integer (whole number) that is not larger than the number itself. In other words, INT tells the computer to round **down** (truncate) a decimal number to a whole number—which means that it simply cuts off anything to the right of a decimal point.

Try this short program to get a few examples:

```
NEW

10 PRINT INT(55.8)

20 PRINT INT(8.9)

30 PRINT INT(3.01)
```

When you run this program, the computer gives you these responses:

```
55

8

3
```

LOG

The LOG operation identifies the natural (base) **logarithm** of the specified number (or variable value). LOG is exactly the opposite of the EXP operation. To find the natural logarithm of 15, enter the following:

```
PRINT LOG(15)
```

The computer responds like this:

```
2.70805
```

RND

The RND operation generates an artificial **random** number. The numbers produced are not actually random numbers, since the computer uses a specific procedure to create them. However, due to the rate at which they vary, they appear to be true random numbers.

Some computers require the use of the command RANDOM at the beginning of a program that uses a RND statement. The exact method in which RND is used varies from computer to computer. On most computers the format used here is acceptable. But if your computer gives you an error message, refer to your operator's manual to find the precise form required for your computer.

Using the RND statement in an expression creates random numbers within whatever restrictions you set. Try these examples:

```
PRINT (25*RND(1))

PRINT INT(25*RND(1))
```

The first example produces a number between 0 and 25; the second produces a whole number between 0 and 25.

The RND and INT operations are useful in many game programs. Various random values add an element of surprise to some games and are necessary in many others. For example, in a program that plays blackjack, the computer usually creates the effect of drawing different cards by using a RND statement. And since the RND statement generates a number that is not necessarily a whole number, an INT statement is used to round the value down to make it a whole number.

SGN

The SGN operation identifies the **sign** of a value. The computer responds with a −1 for any value that is negative, 0 if the value is zero, and +1 for any value that is positive.

Try this example:

```
PRINT SGN (-312)
```

The computer responds like this:

```
-1
```

SIN

The SIN operation identifies the **sine** of the specified number (or variable value) when the value is expressed in radians. To find the sine of 45, enter the following:

```
PRINT SIN(45)
```

The computer responds like this:

```
.850902
```

SQR

The SQR operation identifies the **square root** of the specified number or variable. To find the square root of 25, enter the following:

```
PRINT SQR(25)
```

The computer responds like this:

```
5
```

TAN

The TAN operation identifies the **tangent** of the specified number (or variable value) when the value is expressed in radians. To find the tangent of 129, enter the following:

```
PRINT TAN(129)
```

The computer responds like this:

```
.197194
```

MACHINE LANGUAGE

Now we'll consider some operations that are available on some but not all personal computers. These operations have direct control over the language in which your computer works. That language is called **machine language.** Since it deals entirely with numbers and never with words, it is a great deal more difficult to work with than BASIC. There are three common statements that provide direct access to the machine language of personal computers. Those statements are PEEK, POKE, and USR.

CAUTION

The PEEK, POKE, and USR statements allow direct control of the system. When using these statements, be very careful. You need to know where you are when you POKE or what you are accessing with USR. And you need to double-check your work before running programs. If you POKE the wrong locations (such as the registers or stack area) you can cause your program to crash, which will force you to start over from the beginning. In rare cases, if you POKE the memory circuitry, you can damage the circuitry and may need to replace the memory chips. Because of their tremendous power, PEEK, POKE, and USR are not even valid commands on some computers. Check your operator's manual before you try to use these commands.

We do not want to scare you away from using PEEK, POKE, and USR, but we do want you to be careful. Know where you are when you are using them.

PEEK

The PEEK statement provides the value that is stored at a specific memory location. The value is given in decimal form (though your computer actually works with numbers in binary form). Before you use PEEK, you should consult the memory map in the operations manual for your computer. PEEK (X) provides the contents of the memory address represented by the value given.

POKE

The POKE statement loads a specific value into a specified memory location. The complete statement takes the form POKE (X, Y). Here X is the address where we want to store data, and Y is the data that we want to store. The address and the data must be separated by a comma.

If you are writing the program, you should consult the memory map in the technical overview section of the operations manual to check the location to poke. The value (Y) can range from 0 to 255. The address (X) can range from 1 to 65,535 (the limit of addressable memory in an 8-bit microcomputer). Exceptions to the limit of 65,535 exist in the case of 16-bit computers (like the TI-99/4A and the IBM Personal Computer).

Both PEEK and POKE are useful in using machine language instructions. By using them, you can access machine language from a BASIC program for more efficient use of your system.

USR or CALL

The USR operation (named CALL on some computers) enables you to link your BASIC program with machine language to run more efficient programs. You can **call** machine language from a BASIC program with the USR operation and return to BASIC when you need to. Machine language, while far more efficient to use, is also considerably more difficult to use than BASIC. It requires you to be familiar with assembly language programming, as well as with the instruction set of the microprocessor used by your computer.

REVIEW

In this chapter, you learned that...

- Library operations generally either work with character strings or perform advanced mathematical operations.

- The LEN operation counts the number of characters (including spaces) in the specified character string.

- The LEFT $, MID $, and RIGHT $ operations identify the specified number of beginning, middle, and end characters (respectively) in a character string.

- The STR $ operation converts a numeric value into a character string. The VAL operation does just the opposite, converting a character string into a numeric value.

- The ASC operation gives the ASCII value that corresponds to the character specified. The CHR $ operation does just the opposite, giving the character equivalent to a specified ASCII value.

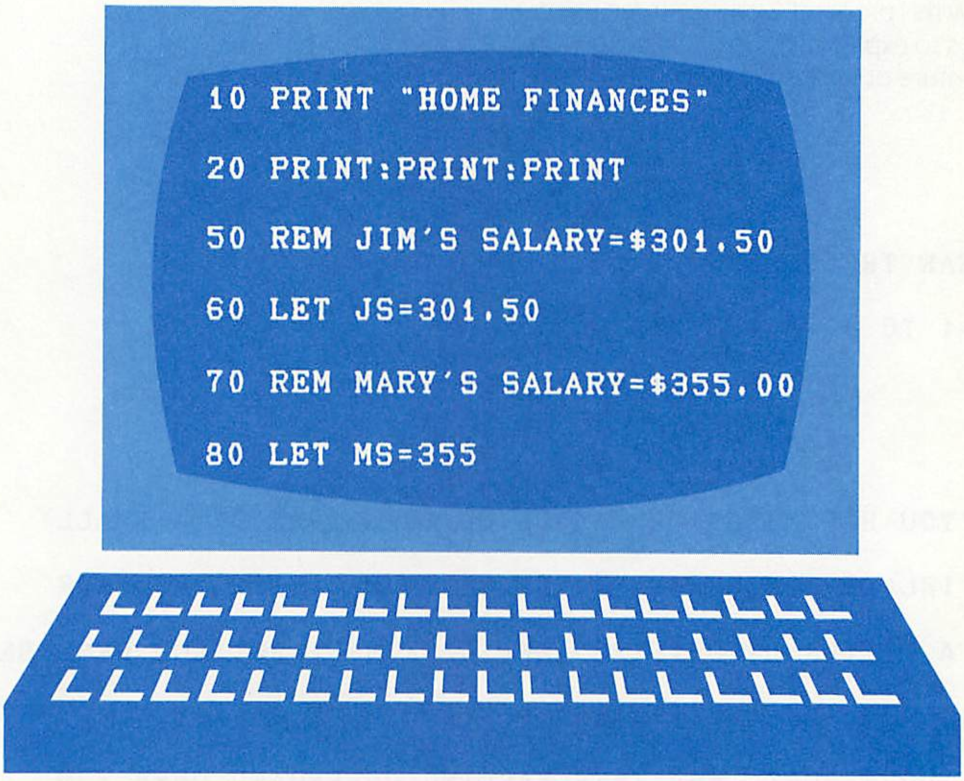
- The SPC (or PRINT AT) and TAB operations move the cursor to specified places on the screen.

- The ABS, ATN, COS, EXP, INT, LOG, RND, SGN, SIN, SQR, and TAN operations perform advanced mathematical functions.

- The PEEK, POKE, and USR (or CALL) operations allow direct control of the machine language of most computer systems.

Chapter 7

SOME SAMPLE PROGRAMS



This book is not intended to turn you into a computer programmer overnight, but by now you should have a good idea of what BASIC is and how it works. If you would like to learn more about programming, the sample programs in this chapter will help you practice and learn.

At this point, one of the best things you can do to learn more about programming is to work with and experiment with programs. Enter one of the following sample programs into your computer's memory. Try to figure out what the program does and how it works. Then try to modify it in some way, using what you have learned about BASIC programs.

When you get tired of these programs, buy a book of programs at your local computer store or bookstore, and try modifying some of the programs in it. Try writing a couple of short, simple programs of your own. The only way to learn to program is to sit down in front of the computer and try it—experiment!

ADVENTURE GAME PROGRAM

Here is a program for an adventure game called "Reef Rescue." The program uses IF–THEN statements to send you on different adventures every time you play the game. Note how lines 580 to 600 keep track of how long the player has waited on the reef. Once you understand how this program works, you can use the same techniques to expand on this game or to write your own adventure or mystery games.

```
10 IW=0
20 REM CLEAR THE SCREEN
30 FOR CL=1 TO 40:PRINT:NEXT CL
40 PRINT "            WELCOME TO REEF RESCUE!"
50 PRINT
60 PRINT "YOU HAVE JUST WASHED UP ON THE SHORE OF A SMALL"
70 PRINT "ISLAND SOMEWHERE IN THE SOUTH PACIFIC.  YOU ARE"
80 PRINT "A LITTLE DISORIENTED AND CONFUSED, BUT YOU REMEMBER"
90 PRINT "SOMETHING ABOUT GOING FOR A CRUISE ON A REAL-LIFE"
100 PRINT "LOVE BOAT, AND THEN FALLING OVERBOARD WHEN YOU"
110 PRINT "FAINTED WHILE READING YOUR LINES."
120 PRINT
130 PRINT "THERE IS A CLIFF TO THE EAST, WITH A TREACHEROUS"
140 PRINT "STONE PATH LEADING TO THE TOP.  THE BEACH IS TOO"
150 PRINT "NARROW TO FOLLOW SOUTH, BUT IT LOOKS LIKE YOU COULD"
160 PRINT "WALK NORTH ALONG THE BEACH."
170 PRINT
180 INPUT "DO YOU WANT TO GO NORTH OR EAST";DI$
```

```
190 PRINT
200 IF DI$="NORTH" THEN GOTO 260
210 IF DI$="EAST" THEN GOTO 230
220 PRINT DI$;" IS NOT ONE OF YOUR CHOICES!":PRINT:GOTO 180
230 PRINT "YOU CLIMB SLOWLY TO THE TOP OF THE CLIFF, WHERE YOU"
240 PRINT "ARE EATEN BY A MAN-EATING TIGER.  BUMMER."
250 GOTO 760
260 PRINT "WALKING ALONG THE BEACH, YOU SOON FIND A SMALL WOODEN"
270 PRINT "BOAT STUCK IN THE SAND.  THE TIDE IS RISING, AND THE"
280 PRINT "BOAT WILL SOON BE FLOATING.  A DARK AND NARROW PATH"
290 PRINT "LEADS INTO THE JUNGLE BEHIND THE BOAT."
300 PRINT
310 INPUT "DO YOU WANT TO GO BOATING OR WALKING";DI$
320 PRINT
330 IF DI$="BOATING" THEN GOTO 470
340 IF DI$="WALKING" THEN GOTO 360
350 PRINT DI$;" IS NOT ONE OF YOUR CHOICES!":PRINT:GOTO 310
360 PRINT "THE PATH LEADS TO A SMALL HOUSE.  A SMALL ELDERLY"
```

```
370 PRINT "GENTLEMAN COMES OUT OF THE HOUSE TO GREET YOU."
380 PRINT "YOU TELL HIM OF YOUR JOURNEY, AND HE"
390 PRINT "YELLS BACK TO SOMEONE INSIDE THE HOUSE, 'JOSEPH,"
400 PRINT "BRING THE LEAR JET AROUND AND TAKE THIS"
410 PRINT "POOR SOUL BACK TO THE MAINLAND.'"
420 PRINT
430 PRINT "SOON, A LEAR JET LANDS NEARBY.  YOU CLIMB ABOARD, AND"
440 PRINT "THE PILOT ASKS YOU WHERE YOU'RE GOING.  MOMENTS LATER,"
450 PRINT "YOU ARE AIRBORNE AND HEADED FOR HOME."
460 GOTO 760
470 PRINT "THE TIDE SLOWLY RISES, AND THE BOAT BREAKS FREE.  YOU"
480 PRINT "HOP ABOARD AT THE LAST SECOND, AND THE BOAT DRIFTS"
490 PRINT "SLOWLY OUT TO THE REEF, WHERE IT RUNS AGROUND.  YOU"
500 PRINT "HAVE TWO CHOICES NOW -- YOU CAN WAIT ON THE REEF, OR SWIM"
510 PRINT "BACK TO SHORE."
520 PRINT
530 INPUT "WHAT DO YOU WANT TO DO";DI$
540 PRINT
550 IF DI$="SWIM" THEN GOTO 630
560 IF DI$="WAIT" THEN GOTO 580
570 PRINT DI$;" IS NOT ONE OF YOUR CHOICES!":PRINT:GOTO 530
580 IF IW=20 THEN GOTO 740
590 IF IW>10 THEN GOSUB 720
```

```
600 IW=IW+1
610 PRINT "IT IS MUCH LATER NOW.  YOU HAVE TWO CHOICES -- YOU"
620 PRINT "CAN WAIT LONGER, OR SWIM BACK TO SHORE.":PRINT:GOTO 530
630 PRINT "YOU SLIP INTO THE WARM WATER AND BEGIN SWIMMING TOWARD"
640 PRINT "SHORE.  THE TIDE IS GOING BACK OUT NOW, SO IT TAKES A"
650 PRINT "LONG TIME TO REACH THE BEACH.  FINALLY, YOUR TOE HITS"
660 PRINT "BOTTOM AND YOU STAND UP.  YOUR FEET SINK IN A LITTLE"
670 PRINT "AND YOU FEEL SOMETHING SLIMY AGAINST YOUR LEGS.  TOO"
680 PRINT "LATE, YOU REALIZE THAT YOU'RE STANDING IN A GIANT CLAM"
690 PRINT "AND HE IS CLOSING HIS JAWS.  THE TIDE IS RISING AGAIN"
700 PRINT "NOW, AND...   (WE'LL LEAVE OUT THE GRUESOME DETAILS)"
710 GOTO 760
720 PRINT "A HELICOPTER CAN BE HEARD IN THE DISTANCE."
730 PRINT:RETURN
740 PRINT "CONGRATULATIONS!  YOUR INFINITE PATIENCE HAS FINALLY"
750 PRINT "PAID OFF.  THE HELICOPTER LANDS, AND YOU CLIMB ABOARD."
760 END
```

EDUCATIONAL PROGRAM

The following program can form the basis of a computer-assisted education program for virtually any subject. It makes use of READ and DATA statements, and you can change the information in the statements to match the questions you would like the computer to ask. Just make sure that you change the final value of 5 in the FOR–NEXT loop in line 50 to match the number of DATA statements used from line 200 on.

```
10 PRINT "THIS PROGRAM WILL TEST YOUR KNOWLEDGE OF"

20 PRINT "IMPORTANT YEARS IN AMERICAN HISTORY."

30 PRINT "ENTER YOUR ANSWER FOLLOWING EACH QUESTION."

35 PRINT "IF YOU CANNOT ANSWER, ENTER NEXT FOR THE NEXT
QUESTION."

40 PRINT:FOR T=1 TO 1000:NEXT T

50 FOR A=1 TO 5

60 READ Q$,Y$

70 PRINT "WHAT IS THE YEAR OF THE"

80 PRINT Q$

90 INPUT B$

95 IF B$="NEXT" THEN PRINT "THE CORRECT ANSWER IS ";Y$:GOTO 150

100 IF B$=Y$ THEN 130

110 PRINT:PRINT "NOT QUITE CORRECT. TRY AGAIN."

120 GOTO 70

130 PRINT:PRINT "THAT'S RIGHT!"

140 FOR T=1 TO 1000:NEXT T

150 NEXT A

200 DATA "DISCOVERY OF AMERICA BY COLUMBUS","1492"
```

```
210 DATA "START OF REVOLUTIONARY WAR","1776"
220 DATA "START OF CIVIL WAR","1861"
230 DATA "GREAT SAN FRANCISCO EARTHQUAKE","1906"
240 DATA "BOMBING OF PEARL HARBOR","1941"
```

FORM LETTER PROGRAM

Here is a program for a computerized form letter. The program uses string variables and INPUT statements to personalize the letter. Of course, the computerized form letters you receive in the mail are sometimes more sophisticated. But this simple program shows you the basic principles behind form letters. Using those principles, you can write your own form letter, using whatever variables you wish within the letter.

You might want to adjust the number of characters used in the character strings in the PRINT statements to match the number of columns that can be displayed on your computer. If you have a printer and want printed copies of the letter, use the LPRINT command (or whatever command your computer uses to provide output to the printer) instead of PRINT in those program lines containing the actual letter.

```
10 PRINT "SWEEPSTAKES WINNER FORM LETTER"

20 PRINT "ENTER RECIPIENT'S FIRST NAME."

30 INPUT TN$

40 PRINT "ENTER RECIPIENT'S LAST NAME."

50 INPUT LN$

60 PRINT "ENTER RECIPIENT'S STREET ADDRESS."

70 INPUT AD$

80 PRINT "ENTER RECIPIENT'S CITY."

90 INPUT CT$

100 PRINT "ENTER RECIPIENT'S STATE."

110 INPUT ST$

120 PRINT "ENTER RECIPIENT'S ZIP CODE."

130 INPUT ZC$
```

```
140 PRINT "ENTER PRIZE WON."
150 INPUT PW$
160 FOR CL=1 TO 40:PRINT:NEXT CL
200 PRINT TN$;" ";LN$
210 PRINT AD$
220 PRINT CT$;" ";ST$;" ";ZC$
230 PRINT
240 PRINT "CONGRATULATIONS, ";TN$;" ";LN$;"!!"
250 PRINT "YOU ARE ONE OF THE LUCKY WINNERS IN"
260 PRINT "THE STUPENDOUS SUPER SWEEPSTAKES."
270 PRINT
280 PRINT "THE WHOLE ";LN$;" FAMILY WILL BE"
290 PRINT "DELIGHTED WHEN YOUR SPECIAL PRIZE"
```

```
300 PRINT
310 PRINT TAB(10)PW$
320 PRINT
330 PRINT "IS DELIVERED TO YOUR DOOR AT"
340 PRINT AD$;", YOU'LL BE"
350 PRINT "THE ENVY OF ALL YOUR FRIENDS AND"
360 PRINT "NEIGHBORS IN ";CT$;"."
370 PRINT
380 PRINT "IMAGINE!! ";PW$
390 PRINT "WILL BE ALL YOURS!! YOUR PRIZE WILL BE"
400 PRINT "DELIVERED WITHIN THREE WEEKS."
410 PRINT
420 PRINT "CONGRATULATIONS AGAIN, ";TN$;" ";LN$;","
430 PRINT "ON YOUR GOOD FORTUNE!"
440 PRINT
450 PRINT "SINCERELY,"
460 FOR CL=1 TO 4:PRINT:NEXT CL
470 PRINT "ROGER THOMPSON"
480 PRINT "EXECUTIVE VICE-PRESIDENT"
490 PRINT "STUPENDOUS SUPER SWEEPSTAKES"
```

HOME FINANCE ANALYSIS

Here is a program for home finance management. Home finance software programs are extremely popular, and many such programs are available on the market. However, there is a problem with "standard" home finance programs. They are written to match any and all family finances, which does not necessarily mean **your** finances. You can solve this problem (and get a little practice in modifying a BASIC program) by developing a program tailored to your own home finances. Here is a basic personal finance program, along with details regarding each of the subroutines. By applying the suggestions following the program, you can customize the program to match your individual financial picture.

The program consists of four major subroutines. The first is an income subroutine that will take into account various pay periods and store in memory the total income for a given month. It also provides an input for additional income of varying amounts. The second subroutine considers fixed expenses—those costs that do not normally vary on a month-to-month basis. The third subroutine considers expenses that do vary, such as utility bills. The fourth subroutine is a display or print subroutine that provides a "balance sheet" of the financial information calculated by the program. The program is flexible in that it affords you the opportunity to shift expenses and re-display the new results if you don't like the looks of the balance sheet. For the months when expenses are greater than income, the program gives you the opportunity to borrow from savings or sources of credit; then you can re-display the balance sheet with the updated results.

```
10 PRINT "----HOME FINANCE ANALYSIS----"

20 PRINT:PRINT:PRINT

50 REM JIM'S SALARY=$301.50

60 LET JS=301.50

70 REM MARY'S SALARY=$355.00

80 LET MS=355

90 PRINT "HOW MANY PAY PERIODS THIS MONTH FOR JIM?"

100 INPUT JP

110 PRINT "HOW MANY PAY PERIODS THIS MONTH FOR MARY?"

120 INPUT MP

130 LET TS=(JS*JP)+(MS*MP)

140 PRINT "ANY ADDITIONAL INCOME I SHOULD KNOW ABOUT?"
```

```
150 INPUT A$:IF A$="YES" THEN 200
160 IF A$="Y" THEN 200
170 IF A$="NO" THEN 300
180 IF A$="N" THEN 300
190 PRINT "A SIMPLE YES OR NO WILL DO!":GOTO 140
200 PRINT "ENTER THE ADDITIONAL AMOUNT OF INCOME."
210 INPUT AS
220 LET TS=TS+AS
300 REM FIXED EXPENSE CALCULATIONS
310 REM MORTGAGE=$682.83
320 LET A=682.83
330 REM CAR PAYMENT=$188.85
340 LET B=188.85
350 REM SECOND CAR PAYMENT=$124.80
360 LET C=124.80
370 REM CAR INSURANCE=$32.30
380 LET D=32.30
390 REM BEDROOM FURNITURE=$57.50
400 LET E=57.50
500 LET FX=A+B+C+D+E+F+G+H+J+K
600 FOR CL=1 TO 30:PRINT:NEXT CL
610 PRINT "I WILL NEED INFORMATION REGARDING"
620 PRINT "THIS MONTH'S VARIABLE EXPENSES."
```

```
630 PRINT:PRINT "GROCERIES COST?"
640 INPUT M
650 PRINT "ELECTRIC BILL?"
660 INPUT N
670 PRINT "GAS BILL ?"
680 INPUT P
690 PRINT "WATER BILL?"
700 INPUT Q
710 PRINT "HEALTH COSTS?"
720 INPUT R
730 PRINT "CLOTHING COSTS?"
740 INPUT S
750 PRINT "CHARGE CARD BILL?"
760 INPUT T
770 PRINT "ANY ADDITIONAL EXPENSES? YES OR NO."
780 INPUT B$:IF B$="YES" THEN 830
790 IF B$="Y" THEN 830
800 IF B$="NO" THEN 1000
810 IF B$="N" THEN 1000
820 PRINT "A SIMPLE YES OR NO WILL DO!":GOTO 770
830 PRINT "PLEASE DESCRIBE THE ADDITIONAL EXPENSE"
840 PRINT "IN TWENTY CHARACTERS OR LESS."
850 INPUT AX$
```

```
860 PRINT "PLEASE ENTER THE AMOUNT"
870 PRINT "OF THE ADDITIONAL EXPENSE."
880 INPUT AX
1000 LET VX=M+N+P+Q+R+S+T+U+V+AX
1110 FOR TT=1 TO 6000:NEXT TT:FOR CL=1 TO 30:PRINT:NEXT CL
1120 PRINT "YOUR VARIABLE EXPENSES TOTAL $";VX
1130 PRINT "THIS MONTH'S EXPENSES TOTAL $";(VX+FX)
1140 PRINT "WHEN YOUR EXPENSES ARE SUBTRACTED FROM YOUR INCOME,"
1150 PRINT "THE AMOUNT REMAINING IS: $";(TS-(VX+FX))
1160 IF (TS-(VX+FX))<0 THEN 1400
1170 PRINT "ANY FUNDS TO SAVINGS? YES OR NO."
1180 INPUT C$:IF C$="YES" THEN 1230
1190 IF C$="Y" THEN 1230
1200 IF C$="NO" THEN 2000
1210 IF C$="N" THEN 2000
1220 PRINT "A SIMPLE YES OR NO WILL DO!":GOTO 1170
1230 PRINT "ENTER AMOUNT DESIRED."
1240 INPUT SV:IF (TS-(VX+FX+SV))<0 THEN 1600
1250 FOR TT=1 TO 6000:NEXT TT:FOR CL=1 TO 30:PRINT:NEXT CL
1260 PRINT "WHEN SAVINGS ARE DEDUCTED FROM YOUR READY CASH,"
1270 PRINT "YOUR NEW READY CASH REMAINING IS: $";(TS-(VX+FX+SV))
1280 LET RC=(TS-(VX+FX+SV)):GOTO 2000
1400 FOR TT=1 TO 6000:NEXT TT:FOR CL=1 TO 30:PRINT:NEXT CL
```

```
1410 PRINT "WARNING! CALCULATIONS SHOW A DEFICIT"
1420 PRINT "IN THIS MONTH'S FINANCES."
1430 PRINT "CHOOSE AN OPTION FROM THE MENU BELOW."
1440 PRINT:PRINT "(1) WITHDRAW FROM SAVINGS"
1450 PRINT "(2) BORROW FROM SOURCE OF CREDIT"
1460 PRINT "(3) REDUCE FLEXIBLE EXPENSES"
1470 PRINT:PRINT "ENTER CHOICE NUMBER."
1480 INPUT CN
1490 ON CN GOTO 1700,1800,1900
1600 PRINT "SAVINGS AMOUNT HAS CREATED DEFICIT."
1610 PRINT "PLEASE RE-FIGURE SAVINGS."
1620 LET SV=0:GOTO 1170
1700 PRINT "ENTER AMOUNT FROM SAVINGS"
1710 INPUT SN
1720 PRINT "THIS AMOUNT WILL NOW BE ADDED TO"
1730 PRINT "YOUR INCOME, AND A NEW READY-CASH BALANCE SHOWN."
1740 LET TS=TS+SN
1750 FOR TT=1 TO 6000:NEXT TT:FOR CL=1 TO 30:PRINT:NEXT CL
1760 GOTO 1140
```

```
1800 PRINT "ENTER AMOUNT BORROWED."
1810 PRINT "ALL LOANS WILL IMPACT FUTURE EXPENSES."
1820 INPUT BN
1830 PRINT "THIS AMOUNT WILL NOW BE ADDED TO YOUR"
1840 PRINT "INCOME, AND A NEW READY-CASH BALANCE SHOWN."
1850 LET TS=TS+BN
1860 FOR TT=1 TO 6000:NEXT TT:FOR CL=1 TO 30:PRINT:NEXT CL
1870 GOTO 1140
1900 PRINT "THE FIGURES THAT YOU RE-ENTER WILL"
1910 PRINT "REPLACE THE VARIABLE EXPENSES ENTERED EARLIER."
1920 FOR TT=1 TO 6000:NEXT TT:GOTO 610
2000 REM BALANCE SHEET
2010 PRINT:PRINT "PRESS D AND RETURN TO"
2020 PRINT "DISPLAY BALANCE SHEET."
2030 INPUT F$:IF F$="D" THEN 2070
2040 PRINT "WHAT???"
2050 GOTO 2010
2070 PRINT "---INCOME-----------FIXED EXPENSES---"
2080 PRINT "-----------------------------------"
2090 PRINT "JIM'S=$";(JS*JP)
2100 PRINT "MARY'S=$";(MS*MP)
2110 PRINT "ADDITIONAL=$";AS
2120 PRINT "TOTAL=$";TS
2130 PRINT "                   MORTGAGE=$";A
```

```
2140 PRINT "                         CAR PMT.#1=$";B
2150 PRINT "                         CAR PMT.#2=$";C
2160 PRINT "                         INSURANCE=$";D
2170 PRINT "                         FURNITURE=$";E
2250 PRINT:PRINT "PRESS N AND RETURN TO DISPLAY"
2260 PRINT "THE NEXT BALANCE SHEET."
2270 INPUT D$:IF D$="N" THEN 2300
2280 PRINT "WHAT???":PRINT:GOTO 2250
2300 FOR CL=1 TO 30:PRINT:NEXT CL
2310 PRINT"---VARIABLE EXPENSES-----------------------"
2320 PRINT"--------------------------------"
2330 PRINT "GROCERIES=$";M
2340 PRINT "ELECTRIC BILL=$";N
2350 PRINT "GAS BILL=$";P
2360 PRINT "WATER BILL=$";Q
2370 PRINT "HEALTH COSTS=$";R
2380 PRINT "CLOTHING COSTS=$";S
2390 PRINT "CHARGE CARD=$";T
2395 PRINT "-ADDITIONAL EXPENSES-"
2400 PRINT AX$;"=$";AX
2410 PRINT "FIXED EXPENSES TOTAL: $";FX
2420 PRINT "VARIABLE EXPENSES TOTAL: $";VX
2430 IF SV>0 THEN PRINT "AMOUNT TO SAVINGS IS: $";SV
2440 IF SN>0 THEN PRINT "AMOUNT FROM SAVINGS IS: $";SN
```

```
2450 IF BN>0 THEN PRINT "AMOUNT BORROWED IS: $";BN
2460 PRINT "REMAINING READY CASH IS: $";(TS-(VX+FX+SV))
2470 PRINT "ENTER Y AND PRESS RETURN FOR MENU."
2480 INPUT E$:IF E$="Y" THEN 2500
2490 PRINT "WHAT???":PRINT:GOTO 2470
2500 FOR CL=1 TO 30:PRINT:NEXT CL
2510 PRINT "WOULD YOU LIKE TO:"
2520 PRINT "(1) RE-DISPLAY THE ENTIRE BALANCE SHEET"
2530 PRINT "(2) RE-RUN THE PROGRAM FROM START"
2540 PRINT "(3) RE-RUN THE VARIABLE EXPENSE PORTION"
2550 PRINT "(4) QUIT"
2560 PRINT "ENTER NUMBER OF CHOICE."
2570 INPUT Z
2580 ON Z GOTO 2000,10,1900,2590
2590 FOR CL=1 TO 30:PRINT:NEXT CL
2600 PRINT "-----HAVE A NICE DAY-----":END
```

Program Analysis

The first subroutine, lines 10 to 220, computes total monthly income. Since pay periods vary from person to person, the program makes its calculations based on the number of pay periods in a given month. Lines 50 through 130 assume two wage earners in a family. Lines similar to lines 70, 80, 110, and 120 can be added for each additional wage earner in the family, or those lines can be deleted entirely if there is only one wage earner. Line 130 performs the actual calculation of total monthly income, which is represented by the variable TS. Lines 140 to 220 comprise an additional income routine for moonlighting , garage sales, and the like. You will need to substitute your salaries for the values given as JS and MS in lines 50 to 80.

Lines 300 to 500 comprise the fixed expense subroutine. You will need to substitute the names of each of your fixed expenses for those expenses used in lines 310, 330, 350, 370, and 390. In a similar fashion, you will need to substitute the values of those fixed expenses for the values of the variables in lines 320, 340, 360, 380, and 400. If you have more than five fixed expenses each month, you can add lines similar to lines 310 to 400, numbering them anywhere in the 401 to 499 range, using the unused variables of F, G, H, J, and K to represent the values of the expenses.

Lines 610 to 760 comprise the variable expense subroutine. Again, the names of the bills listed in the PRINT statements can be changed, if desired, to reflect your specific needs. The INPUT statements will cause the amounts entered (at the time the program is run) to be assigned to the variables M through V. Lines 770 to 880 allow additional unplanned expenses to be entered.

Lines 1000 to 2000 should not require any modification by you. These lines will total all expenses, subtract them from the total income, and display the remaining ready cash. A subroutine will be entered if a deficit occurs. This subroutine (lines 1400 to 1920) will present a menu that allows you to resolve deficits by withdrawing from savings, borrowing, or reducing flexible expenses.

Lines 2000 to 2590 display a balance sheet showing the results of all calculations. You will need to change the names of the expenses in the PRINT statements (lines 2090 to 2170 and 2330 to 2390). Also, if you added lines within the range of 401 to 499 to represent additional fixed expenses, you will need to add additional lines to match these between line numbers 2171 and 2249. Each of the lines that you add must be a PRINT statement, showing the name of the expense and the variable chosen (from F to K) that matches it. Lines 2250 to 2280 exist only for the purpose of splitting the balance sheet onto two full screens. This is necessary with most personal computers, as the balance sheet will not fit on one screen. If your personal computer displays 32 (or more) lines per screen, you can delete lines 2250 to 2280 entirely and fit the balance sheet on a single screen. If you desire printed output, and your computer uses the LPRINT statement to provide output to your printer, you can substitute LPRINT for PRINT in the lines between lines 2000 and 2460.

Also note lines 600, 1110, 1250, 1400, 2300, 2500, and 2590. These lines are used to clear the screen by repeating PRINT statements within a FOR–NEXT loop. This isn't the best method of clearing a screen, but it works with all personal computers using Microsoft, PET, or Atari BASIC. If your system has a "clear screen" command (such as CLS on a TRS-80 or HOME on an Apple II), you should use it instead. Lines 1400 and 1920 are used to create time delays before the screen displays further information. If the delay is too short or too long on your system, you can increase or decrease the final value of 6000 in these lines to increase or decrease the time delay.

The information provided here should help you develop a similar program, matched to your finances. It's also possible to modify the program to make it run more efficiently. For instance, you could include disk commands to save the financial values, if you have a disk drive. This program and these suggestions should provide a basis for an efficient cash management program for your home.

BASIC COMMANDS USED IN THIS BOOK

ABS	Identifies the absolute value of the specified number
ASC	Gives the ASCII value that corresponds to the character specified
ATN	Identifies the arctangent
CALL	See **USR.**
CHR$	Gives the character that corresponds to the ASCII value specified
CLEAR	Resets all program variables to zero or null
CLOAD	See **LOAD.**
CONT	Restarts execution of a program that was halted with a STOP command
COS	Identifies the cosine
CSAVE	See **SAVE.**
DATA	Allows storage of data in a program to be called up at a later time through the use of a READ statement
DIM	Sets up a dimensional array (a specific amount of memory space for handling variables)
END	Completes the execution of a program in a normal manner
EXP	Gives the value of *e* raised to the power of the specified number
FOR	Works in combination with the command NEXT; assigns beginning and ending values to the variable which controls the number of repetitions of a loop
GET	See **INPUT.**
GOSUB	Sends the computer to a specific line that is the start of subroutine
GOTO	Sends the computer to the line specified, which varies the sequential execution of program lines
IF–THEN	Causes the computer to take a particular action only under certain conditions. If the expression in the IF part of the statement is true, the computer executes the THEN part of the statement; if the expression is false, the computer ignores the rest of the line and proceeds directly to the next program line.
INPUT (or GET)	Halts the execution of a program until the operator enters values for the variables in the command
INT	Rounds down (truncates) the specified number to the nearest integer (whole number) not larger than the number itself

LEFT$	Identifies the specified number of beginning characters (those at the left side) in the specified character string
LEN	Counts the number of characters (including spaces) in the specified character string
LET	Assigns a value to a variable
LIST	Displays all program lines that are currently in the computer's memory; displays the line(s) specified after the LIST command
LOAD (or CLOAD)	Loads programs into working memory from tape or disk
LOG	Identifies the natural logarithm of the specified number
MID$	Identifies the specified number of middle characters in the specified character string
NEW	Erases any program lines currently in the computer's memory
NEXT	Works in combination with the command FOR; reads the value of the variable to determine whether to send the computer back through a loop
NULL	Inserts the specified number of nulls (dead spaces) in a program line
ON–GOSUB	Sends the computer to one of the specified subroutines, depending on the value of the variable in the ON part of the statement
ON–GOTO	Sends the computer to one of the specified line numbers, depending on the value of the variable in the ON part of the statement
PEEK	Provides the value stored at the specified memory location (in decimal form)
POKE	Loads a specific value into a specified memory location
PRINT	Displays the value of whatever follows the PRINT command
PRINT AT	See **SPC.**
READ	Looks for a DATA statement from which to assign values to variables
REM	Allows the programmer to make notes in the program to keep track of what the program is doing
RESTORE	Restores the information in already used DATA statements so that the information can be used more than once

RETURN	Returns the computer from a subroutine to the line immediately following the GOSUB line in the main program
RIGHT$	Identifies the specified number of end characters (those at the right) of the specified character string
RND	Generates artificial random numbers
RUN	Begins the execution of program lines in sequence
SAVE (or CSAVE)	Stores data and programs on tape or disk
SGN	Identifies the sign of a value (positive, negative, or zero)
SIN	Identifies the sine
SPC (or PRINT AT)	Moves the cursor to the specified location on the screen
SQR	Identifies the square root of the specified number
STEP	Specifies the value by which the variable in a FOR statement will increase
STOP	Halts or interrupts the execution of program lines (but allows for restarting the program with a CONT command)
STR$	Converts a numeric value into a character string
TAB	Moves the cursor to a specified location on the screen
TAN	Identifies the tangent of the specified number
THEN	See **IF–THEN.**
USR (or CALL)	Links your BASIC program with machine language
VAL	Converts a character string into a numeric value

THE ROAD TO COMPUTER LITERACY

There's no better way to learn a computer language than with **Everyone's Guide to BASIC** – the BASIC book designed to meet your needs.

This easy-to-understand book explains the uses of the essential commands in the BASIC language. Easy exercises give you practice using commands. And you'll see how BASIC commands work together in simple, practical computer programs. The exercises and programs in this book were written to work with versions of BASIC used by most personal computers on the market today.

If you're ready to explore the powers of your personal computer – to make your computer really work for you – you need **Everyone's Guide to BASIC**.

BEEKMAN HOUSE
Distributed by Crown Publishers, Inc.
One Park Avenue
New York, New York 10016

ISBN 0-517-41594-1