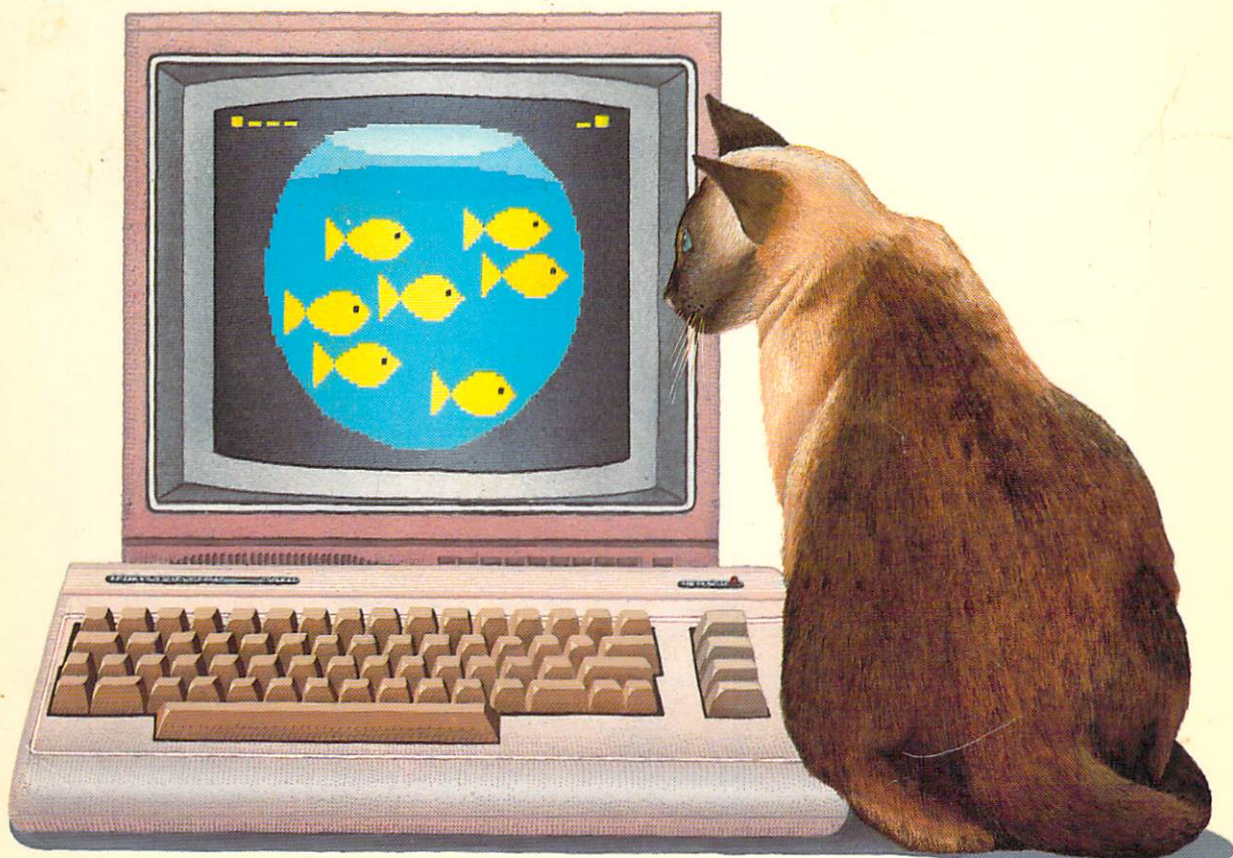


A down-to-earth guide
to the bewildering world of microcomputers

COMMODORE 64

An Intelligent and Intelligible Guide
for the Inquisitive Adult



John A. Heil and Jack Martin *Cathy Hoek*



COMMODORE 64



COMMODORE 64

An Intelligent and Intelligible Guide
for the Inquisitive Adult

John A. Heil and Jack Martin

A Banbury Book

Published by Banbury Books, Inc.
353 W. Lancaster Avenue Wayne, Pennsylvania 19087
Copyright © 1983 by Banbury Books, Inc.

All rights reserved.

No part of this book may be reproduced or transmitted in any form
or by any means, electronic or mechanical, including photocopying, recording
or by any information storage and retrieval system,
without the prior written permission of the Publisher, except where
permitted by law.

ISBN: 0-88693-067-7

First printing—December 1983

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

Commodore 64 is a registered trademark of Commodore Business Machines, Inc.

DISCLAIMER OF ALL WARRANTIES AND LIABILITIES

The authors and publisher make no warranties, either
expressed or implied, with respect to this book or with respect to the
programs or the documentation contained herein, their quality, performance,
merchantability, or fitness for any particular purpose. The authors and the
publisher shall not be liable for any incidental or consequential damage in
connection with, or arising out of, the furnishing, performance,
or the use of materials in this book.

ACKNOWLEDGMENT

The authors would like to thank Robert W. Stat for his invaluable guidance in the preparation of this book.



AN INTRODUCTION

This is a book about a language. A curious tongue, it is never spoken. Even more strange, almost no one ever bothers to read the works that are written in it. It is a silent voice used by people who want to communicate with a certain kind of machine: the computer.

Since you are undoubtedly new to the world of computers, you probably wonder why these clever devices haven't been taught to understand English. This would, after all, make living with the things far more comfortable. You could then simply address a computer directly, saying precisely what you mean and asking for exactly what you desire. There would be no question about your intentions.

Unfortunately, the English language is far too imprecise a vehicle for computers to work with. The meanings, subtleties and organization of the way we normally speak and write are exceedingly complex, individually colored and subject to a huge number of interpretations. And computers, thus far in their development still lacking true intelligence, find the act of subjective interpretation virtually impossible. Therefore, they must be conversed with using a method that admits no ambiguity. And English allows mountains of uncertainty. Consider, for instance, the following lines:

Bob, aged twelve, lobbed a spitball at his teacher.
Mrs. Higgenbottom really gave it to him.

It is clear to any reader that Mrs. Higgenbottom did not supply Bob with his spitball. Neither was she overcome by a sense of gratitude as a result of being pelted by it. That which she "gave" to misguided Bob wasn't anything he was

likely to cherish. But nowhere in those two lines is the direct statement that Bob's teacher was angered by his behavior. Still, any three-year-old could quickly understand the implications buried in this simple story.

If one considers the poet's task in this light, the real power of the English language becomes clear. A poet's purpose is to combine a finite number of words in such a way as to overwhelm the words themselves. Their hidden implications, the maundering dialogue of meanings, the concoctions of the imagination that operate in a poem are what give it a special quality. The language produces something inaudible, invisible, but definitely there.

Computers are spectacularly insensitive to both elegant prose and brilliant poetry. They deal instead with the precise, completely defined and absolutely unequivocal meaning of every symbol, word and punctuation point in their language. Every remark that a computer can understand has one meaning and only one meaning. There is a language filled with questions that are answered either yes or no. Nothing softens the edge of their insistence on accuracy. There are no fine shades of meaning or connotations that echo through your mind in the computer's tongue.

But, once the language has been learned and the exact definitions mastered, you can discover a remarkably broad range of meaning buried in its lock-step methods. After a while you can find a very real form of subtle power in the way a computer speaks and listens. While individual words hold no delightful implications, groups of them develop the kind of complexity that intrigues and satisfies. A really good piece of computer writing can fascinate with its sense of integrity, creativity and meaning.

To make this level of understanding accessible, you have to work with the language until you are comfortable with it. You have to experiment. You have to make mistakes. You have to let its expressions rattle around in your head, testing your babble on the computer, allowing ideas to ricochet through its workings to see how they are handled.

Of course, this is how we all learn to talk in the first place. We learn to interpret the mysteries of English through exposure to it. And due to our constant involvement with the language we speak, the subtleties and difficulties that it presents bother almost no one in day-to-day conversations. Due to the mind-boggling number of words we hear and utter in our lives, we become at ease with their meanings. We are familiar with, even casual about, the way implication, association, memory and meaning all work together. Our words, like a line of dominoes, fall with predictable results.

Which brings us to the "language" at hand: BASIC. This language is the one most commonly used in communicating with a computer. To a considerable number of intelligent, disciplined minds the syntax of BASIC is an ongoing source of frustration and irritation. Similarly, the punctuation, the very grammar of the language, bewilders. And the exact and exacting definitions of the words in BASIC, the precise but unstated meanings that lie behind short, apparently innocuous remarks, leave many people feeling stupid.

This is so because one learns a programming language very differently from a spoken tongue. No one is inundated from birth with a nearly constant torrent of BASIC commands. We don't become familiar with the language out of sheer exposure to it.

Instead, the average adult discovers the existence of this otherworldly language when he first sits down in front of a computer. Given the much ballyhooed speed and accuracy of the machines, we often assume that the whole business ought to be quick and easy. So, like a child in kindergarten, we set out to compose "Hamlet" using a language we don't understand. The result, needless to say, is foreseeable.

The purpose of this book is to familiarize the reader with the words, sentences and paragraphs of the BASIC language, in particular, with the BASIC that operates within the 64. This perfectly wonderful machine offers a painlessly inexpensive way of experimenting with a computer. It is impressively versatile for its price. And what's more, it contains a form of the BASIC language that is very much like that on other, larger, even more powerful computers. Essentially, this book and the Commodore 64 offer a reasonably thorough introduction to the world of computers for the price of a new sports jacket.

Thus, there is nothing intentionally grim about this tome. It will not turn you into a first-class programmer. It does not offer an in-depth mastery of your computer's operating details. It is, by no means, the definitive guide to the 64. However, this book is a patient introduction to the machine that enables you to build a sound programming foundation. The basic concepts and approaches offered here make it possible for you to understand more weighty texts and so broaden your knowledge of computing.

In the field of literary criticism there is an imperative which the critic must adopt before beginning a poem or book or play. The directive is simply to accept the invitation of the author. That is, one must suspend one's resistance to the style, language and meaning of a work. One must become credulous and open-minded before reading. This rule is important because the implications of a work of literature require that the reader allow the associations and connections in it to be controlled by the writer.

In learning to read and write in BASIC this same advice proves useful. It is utterly useless, and in fact counterproductive, to bring expectations of any kind to the language at hand. Accept its parameters, its implied limits, its meanings. In so doing you will undoubtedly discover that apparently trivial words, seemingly juvenile sentences, can be assembled in such a way as to have extraordinary results. While not quite the stuff from which poetry can be made, the language has a certain logical grace. And the subtleties it allows, the fine-tuning it makes possible, can be very satisfying indeed. All you need to do is practice muttering to yourself until what you say means precisely what you had in mind.



CHAPTER 1

This book is intended to introduce you to the world of computing via your 64. The idea of doing so isn't exactly revolutionary. In fact, it's not unique to the authors of this book. But this text approaches the matter at hand somewhat differently from most. Here we will try to concentrate on those aspects of the 64 that are typical of microcomputers. The machine's peculiarities, idiosyncrasies and more unusual functions are largely ignored in these chapters.

This approach is being taken because it is unlikely that you, the reader, will forever limit yourself to this particular machine. Given the huge strides that the microcomputer industry makes every year, it is impossible to imagine that you will use your 64 and it alone for your computing during the next five years. While the machine is nothing short of a marvel in regard to its price and power, it is the sort of computer that will be surpassed in both categories given only a little time. In all probability the folks at Commodore are right now concocting a computer that will put the 64 to shame at half the price. And certainly scores of other computer firms are trying to do the same thing.

Thus, perhaps the most important benefit that can be derived from your 64 is an introductory understanding of personal computing. If this book and your computer manage such an introduction, then they will have served as inexpensive and rather painlessly disposable learning tools.

This text is also geared to the most typical computer language available on personal computers: BASIC. While your 64 can be conversed with in a number of languages, BASIC is by far the most common and easiest to learn. Oddly enough, though many programming languages are identical, or nearly so, from computer to computer, BASIC is not. But the central vocabulary and

essential structure of BASIC are the same the world over. So, if you develop a reasonable familiarity with Commodore's BASIC, you will be able to pick up the particular dialect of the language as it is spoken on other machines.

As a point of interest, "BASIC," like so many other computer words, is an acronym (Beginner's All-purpose Symbolic Instruction Code). It is known as a "high-level" language because it is more like spoken English than it is like the native "machine" language that the 64 speaks. At first, to be sure, BASIC won't seem much like anything you've ever spoken. But after a few hours you'll find yourself translating easily even complex "sentences."

Assembling a 64

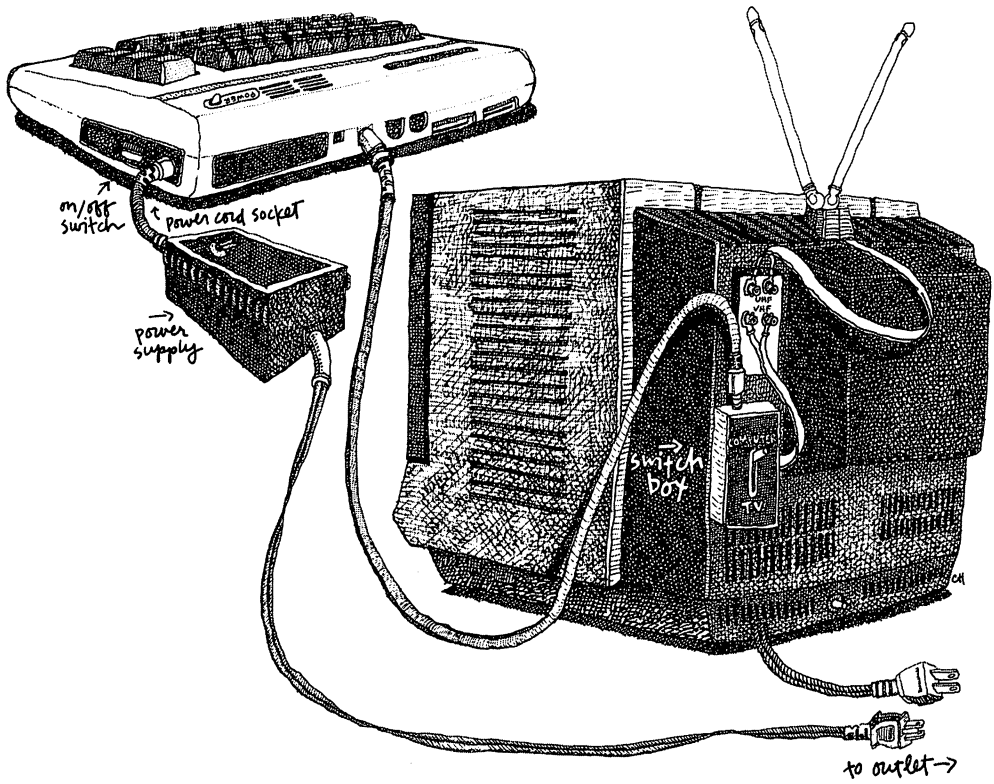
In order to begin, of course, you have to set up the 64 so that it will run. To do so, find a pleasant work area where you'll have some elbow room. The computer itself isn't large or cumbersome, but you'll have to move your television or display screen nearby and you might like some room for a notebook, your Commodore 64 manual and this tutorial.

Once you unpack your 64, you're sure to be bewildered by the wires and plugs that come with it. Start by finding the smallish black box that measures about 2''x2''x4''. It is the power supply. It has two cords, each emerging from a different side of the box. One cord ends with a standard plug that will fit into any wall socket; the other bears a plug with seven tiny sprockets. This odd-looking plug goes into the computer. Insert it into the female socket labeled POWER. Next, plug the standard plug into a wall socket.

Now, find the RF modulator. On our system it consists of two parts. The first of these is a silver box with two wire leads protruding from it and an adhesive surface on the back. Stick this box onto the back of your television. Find a screwdriver and connect the two leads that come from the modulator to the VHF antenna inputs on your television. Again, either lead can be connected to either input. Tighten the screws on the television so that the wires don't disconnect while you're at work. Turn the channel setting to either 3 or 4, whichever has the poorer reception in your area.

Now, adjust the sliding lever on the side of the RF modulator to the "computer" setting. When you are finished computing, merely slide the lever to the "television" setting and you may watch television programs without any interference from your 64. You don't have to disconnect the wires from the television, just flip the switch.

There's another part to the RF modulator. That's the long, thin black cord. Plug one end of the cord into the allotted space in the box that you just attached to your television. The other end of the cord, like the first you plugged in, is single-pronged. At the rear of your computer, toward the center of the back panel, you'll find a single-prong hole. Insert the prong into this hole.



One further note. In the event that you don't have a television set, you'll have to purchase a special monitor, or screen. If you do so, your RF monitors will not operate properly. You must purchase a special monitor cable to wire your computer to such a monitor. These cables are available at any computer store but may not be sold where you bought your 64. To save time and trouble, call a computer retailer and tell him that you need wire to connect a Commodore 64 to a monitor. He'll know if he has what you need.

The cable in question ends with a round plug that holds five prongs. This end plugs into the back of your computer. And the other end will have a single prong. This plugs into the back of your monitor.

Now, turn your television on and turn the sound down all the way. Finally, locate the power switch on the right-hand side of your computer (the toggle switch with the ON label). Turn on your 64.

Screen and Keyboard Basics

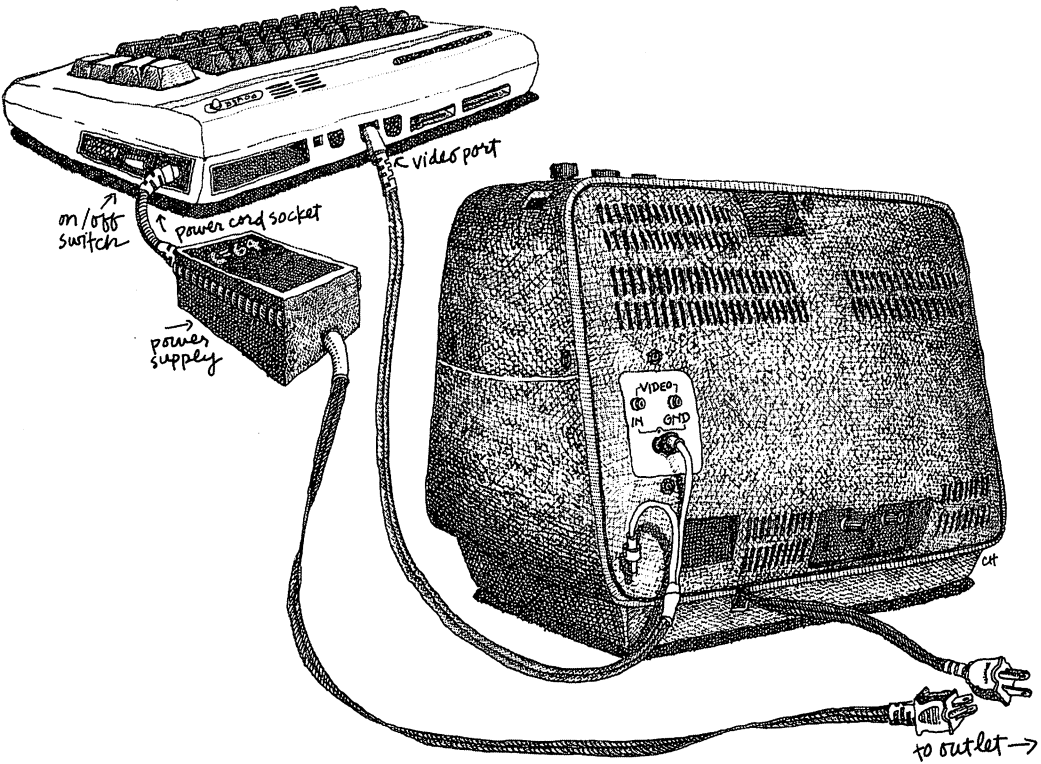
After a few seconds, but not more than a minute or something is wrong, you should see a message on the screen. It should look something like this:

```
* * * * COMMODORE 64 BASIC V2 * * * *  
64K RAM SYSTEM 38911 BASIC BYTES FREE  
READY
```

The first line on the screen informs you that you are being addressed by the computer in its BASIC language. COMMODORE 64 BASIC V2 means that you are looking at Commodore Business Machines' BASIC language, version number 2. You are also being told how much memory is available for you to fill. That's the notice about 38,911 bytes. And, most important, you are being told that the computer is READY to hear from you. Whenever you see the READY message, you will also see that little blinking box just below it. Taken together they indicate that the computer has finished doing whatever had previously involved it. The small box will continue to blink until you take some action. The READY message is called a prompt for obvious reasons: it prompts you to do something. And the blinking box is called a cursor. It indicates the location on the screen where something will happen the next time you touch a key on the keyboard.

Type something—anything. Now take a look at it. Your typing will appear on the screen in capital letters only. If you hold down the shift key (the one at the lower left of the keyboard marked SHIFT) and type some letters, you'll find that the 64 responds by throwing all sorts of oddly shaped squiggles on the screen. This is one of those Commodore 64 idiosyncrasies mentioned earlier.

The keyboard is equipped with two "modes" or ways of behaving. The mode that greets you when you turn on the machine is called the graphics mode and transforms the keyboard into a kind of graphics generator. If you look at the front of the keys, not their tops, you'll see the remarkable number of boxes, triangles, hearts, etc., that you can make in this mode.



The 64 also has what is known as a text mode. It takes control of the computer if you simultaneously hit the shift key and the key next to it marked with the Commodore logo.

Having performed this operation, take a look at the screen. Everything you typed there has been changed. The silly little squiggles are now capital letters and the capital letters that you typed when we first began are now lower-case letters. While the point is hardly worth belaboring, this resulted because the computer is now interpreting your earlier typing in a different way. And some of the differences are more than the alteration of the screen. In fact, the differences between the graphics and text modes of the 64 are significant enough that we will avoid the text mode altogether. It simply makes the computer behave in a very uncharacteristic fashion.

The point is, when using this book, you simply turn the computer on and work in the mode that is made available to you automatically. So, shift back to the graphics mode now.

If you've been typing things as suggested, one of two events has almost certainly occurred. First, you typed away until a line was filled with characters and/or symbols. Having come to the end of the line, you probably hit the return key so that you could start on another line. If you did so you were inevitably greeted with:

```
?SYNTAX ERROR  
READY
```

Disregard the message. Regardless of the syntactical perfection of the material you typed, it doesn't conform to the BASIC language's syntax. Again, ignore this strange response for now.

If you've been fumbling around the keyboard for a while and not hitting the return key, you've noticed that the cursor (blinking box) "returns" all by itself. Thus, you can keep right on typing without ever hitting RETURN.

By now you probably have a screen filled with gibberish, not to mention several spelling errors that have resulted from your efforts to become accustomed to a new keyboard. The 64 has a wonderful feature that allows you to disappear all this typing. Hold the shift key down and press the key in the upper right of the keyboard marked

```
CLR  
HOME
```

Presto. A clear screen and your cursor is in the upper left-hand corner, waiting to go. Type the following:

```
I CAN'T SPEEL
```

Now hit the key in the upper right of the keyboard marked

```
INST  
DEL
```

This is the delete key and it backspaces, removing errata one letter at a time so that you can edit what you've typed. Type awhile, simply treating the computer as a typewriter, and get used to editing your typing with the INST/DEL key.

If you've owned your 64 for some time, the chances are good you've worked with the guide that came with it. If so, you are familiar with several other keys that can make editing easier. Cursor arrows and the insert function are useful aids when working with text on the screen. Here, we'll review a few of the more essential keyboard functions of the computer before starting. If you spend a few moments with these notes, you'll save yourself a great deal of time and frustration later on.

The return key is among the most important on the keyboard. In almost all cases it is the tool that directs information typed on the keyboard to be considered by the computer. If you type a few letters, the keyboard will display them on the screen. But the computer's memory and processing facilities don't pay much attention. Only when you then hit RETURN does the BASIC language in the 64 analyze and react to what you have typed.

The cursor keys are located in the lower right-hand corner of the keyboard. They are labeled

↑ CRSR ↓ and CRSR ←
→

Pressing either one of them moves the cursor down or to the right. If you hold down either of the shift keys (there are two of them) and then press a cursor key, you reverse the direction of the cursor's movement on the screen. These keys allow you to roam around the screen at will, editing anything you find there.

Type:

INSERTIG

This word is a misspelling of "inserting." With the cursor just to the right of the letter G, hold down the shift key and press the horizontal movement

cursor key, the one labeled CRSR. Note that the blinking box backs up to rest over the G. Now, hold down SHIFT again and hit

INST
DEL

again. Note that a space has been inserted between the I and G so that you can fix the word. Type an N and then cursor away from the word.

You can also type right over things. Type:

ANYTHING AT ALL

Then, using the cursor keys, move the cursor to the A in ANYTHING and type:

NOTHING MUCH

To remove the letters not overprinted, simply hit the space bar at the bottom of the keyboard.

Finally, a few computer conventions have to be absorbed before you can accurately communicate with your 64. The first of these results from the fact that computers don't react to the appearance of something but rather consider its precise substance. Thus, the letter l (ell) in its lower-case form is definitely not the number 1 (one). If you are accustomed to typing an ell when you wish to convey the meaning of the number one, you'll have to rid yourself of the habit. The computer cannot accept one for the other. This same situation arises insofar as the letter O and the number 0 (zero) are concerned. You must use exactly the character you mean.

In the graphics mode (the one we will use here and which is automatically functioning when you turn on the machine), the 64 makes distinguishing these keys simple. The letter O and the zero look entirely different because the zero has a slash through it. Type both and look. And the letter L is always capitalized in this mode so that it is difficult to confuse with the number 1.

You should also know that BASIC doesn't make the same inferences people do about numbers and money. In fact, the computer will perform extraordinary financial analyses at terrific speed. But, as it does so, no real sense of monetary value is conveyed. That is, the computer simply works with numbers, not with money. Thus, you don't type \$95.95 to represent ninety-five dollars and ninety-five cents. The dollar sign is superfluous unless you have allowed for it specifically in your program.

Finally, don't use commas in numbers. One hundred thousand is typed "100000" on a computer. The expression "100,000" will create a variety of problems for your 64, so avoid it.

These rules having been stated, it's time to begin talking to the machine. But, before doing so, let us say that this book is written for the owner of a Commodore 64. Of course, you have to own a television or other monitor in order to see what the computer is doing. But no other piece of equipment is required. Still, the last third of this book includes some fairly long programs. Some that involve many minutes of careful typing and that require even more time to test and correct. If you don't own a recording device of some kind, you may find that you are supremely annoyed at having to banish such work forever. Having perfected something, it is frustrating then to dispose of it, keeping no record at all.

Thus, though you needn't do so right now, you should probably purchase a tape recorder to save your programming. What's more, though it is more

expensive than some alternatives, we strongly suggest you buy Commodore's tape recorder. It is built for their machines and makes for extremely easy saving and loading of material into and out of your computer.

Our advice, cautions and rules behind us, it is now time to begin talking to the computer.



CHAPTER 2

The Commodore 64, like all microcomputers, shares certain traits with huge corporate and governmental computers. It operates in much the same way, employing many of the essential elements of any electronic computing device. At the same time, it possesses marked similarities to a very small infant.

On the one hand, the 64 is responsive, powerful and accurate. And on the other hand, it is spectacularly ignorant, immensely uncommunicative and genuinely helpless. Thus, it can serve or annoy, delight or frustrate.

Oddly enough, the difference is simply a matter of learning to command in a proper way this strange typewriter. All you have to do is learn to assert yourself in a manner that the machine will respect. Put differently, you have to learn to speak its language.

To begin, try addressing your computer in a simple, adult fashion. Type:

3+2=

and then hit RETURN.

Apparently you have just been ignored. The cursor zipped to the next line, but nothing else occurred. It's possible the computer knows the answer and isn't telling. Or the thing isn't even going to have the decency to consider your question.

Perhaps a more challenging question might help. Type:

IS BEAUTY TRUTH AND TRUTH BEAUTY?

Now hit RETURN. At least you aren't being ignored. Still, if the syntax of this immortal query was good enough for Keats, you'd think it would be good enough for your computer.

Printing Without a Printer

Rather than continue instructing you to type unworkable strings of letters and numbers, we'll now offer you the first remark you can make to your computer that it will find intelligible. Given the well-known capacity of computers to perform mathematical miracles, we'll begin with a simple numerical exercise. But before instructing the 64 to add a couple of numbers, you must tell it what to do with the result once it has discovered that result. So, you have to issue the computer a command in its BASIC language. You have to give it an imperative.

Try typing:

```
PRINT 3+2
```

and hit RETURN.

The answer, of course, is 5 and appears almost instantly directly below your typed-in line. But while there's a certain satisfaction to getting the computer to do slightly useful work, you can add 3 and 2 as quickly and accurately as it can.

Kindly disregard this minor point for the moment. Instead, concentrate on the PRINT command involved and pay some attention to what your computer seems to know the instant you turn it on.

The sentence PRINT 3+2 has very real meaning to the 64. Translated, the short remark reads, "Take the numbers 3 and 2, add them together and then display the result on the screen." The computer, programmed to understand the command PRINT, also knows the values of the numbers 3 and 2. What's more, the thing can add. The BASIC language, stored permanently in your computer's memory, contains the definitions and working instructions necessary for the machine to understand your request and act on it. The language knows that it must pay attention to the keyboard when you type things and it knows what to do with a wide variety of instructions you might issue. Thus, PRINT 3+2 isn't as simple a request as it might appear.

Next, type:

```
PRINT 3+2+8549+297+11
```

and hit RETURN. Instantly,

```
8862
```

appears on the screen. Now, in all honesty, even the most acutely mathematical among us can't manage the sum in question with the alacrity of a computer. Type:

```
PRINT 3*2
```


and hit RETURN. The * is in the upper right-hand area of the keyboard. The result

6

should indicate to you that * is the computer's symbol for multiplication. It is used rather than the more traditional X because X is a letter. The asterisk is used only for multiplying. Now type:

PRINT 3/2

and hit RETURN. This symbol, signifying division, is the lower-case form of the / key. The result

1.5

appears instantly. Finally, type:

PRINT 3-2

and hit RETURN. The minus sign is next to the plus. Your answer

1

appears quickly. Now try:

PRINT 2-3

and hit RETURN. The result

-1

is produced in a flash and lets you know that the machine is comfortable with negative numbers as well as positive ones.

Now that you know how to add, subtract, multiply and divide, how about some truly arcane math? Type:

PRINT 5 ↑ 8

and hit RETURN. (The ↑ is on the far right side of the keyboard.) On the screen you see

390625

While it may not be immediately obvious to you, 390,625 is 5^8 . Thus, ↑ means "to the power of" and so $5 \uparrow 8$ means $5*5*5*5*5*5*5*5$. Next, type:

PRINT 16 ↑ 3*47/367.88

and hit RETURN. The result

appears.

It is interesting to note that your computer appears to deliberate $3+2$ as briefly as it considers the problem $16 \uparrow 3^{47/367.88}$. In fact, this is not the case. The latter problem takes quite a bit longer to solve. But in the world of computers it is important to remember that time has a somewhat relative meaning. The time it takes a computer to perform an action is often described in terms of nanoseconds. A nanosecond, for those who are curious, is one-billionth of a second. This is an almost unthinkably brief period. So brief, in fact, that light can travel only about nine inches during a nanosecond.

It's logical, therefore, that one cannot readily detect the difference between a hundred nanoseconds and a thousand of them. Either period of time is still too brief to be noticed by our sensibilities. But in long or repetitive computer programs that perform thousands of calculations, the addition of lots of nanosecond-long actions adds up. Adds up to what might loosely be described as a moment or two.

Before continuing, try some Byzantine sorts of calculations using the PRINT command. Don't be concerned if the line numbers and symbols are too long to fit on one line of the screen. The computer doesn't care. Just don't fill more than two lines of the screen with any single PRINT command at this point.

Also notice that you are likely to produce some very odd-looking numbers. At this point don't concern yourself with numbers that have letters in them. The computer is altering the form of these numbers for a reason that will be explained soon.

On Words and Numbers

Thus far you have managed to turn your computer into an awkward and expensive calculator, capable of doing what a \$5.95 Sharp™ can accomplish and still fit in your vest pocket. No mean feat, but nothing to exult over. Type:

```
PRINT 3+2
```

and hit RETURN. Next, type:

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

and hit RETURN.

In the first case you are given the total as calculated by your computer:

```
5
```

In the second case, you are offered something else:

```
3+2
```

The two results, 5 and 3+2, are vastly different. The distinction between them is the result of the quotation marks included in one of the PRINT statements. Put simply, quotation marks instruct the computer to display on the screen EXACTLY that which is between these punctuation marks.

But more important, the computer is making a not altogether obvious distinction. When you surround anything with quotation marks, you tell the computer that the thing surrounded is a word and not a number, and is to be treated as such. It may strike you as odd that "3+2" is a word. In fact, the whole idea probably seems absurd. Still, computers are ignorant beasts and so accept and assume that which they are constructed to accept and assume. And your 64 regards anything inside quotation marks as a word. That's all there is to it.

Even more remarkable, the computer blindly records "words" of all sorts. It never discriminates, makes judgments or offers criticisms. Like a stenographer in a courtroom, dutifully noting the testimony of a witness, the record is verbatim. Anything offered to a computer in the form of a word, surrounded by quotation marks, is absorbed intact by the machine. Every character, punctuation mark and space is put in the machine's memory. Thus, just as in a court, if the accused happens to be a misbegotten liar intent on testifying to obvious fabrications, the record will preserve them.

Recording of this kind is uncolored, objective. No editorializing, analyzing or throwing one's two cents in is allowed. And so, errors, misspellings and nonsense of all sorts are perfectly acceptable to a computer when offered in the form of a word. For instance, type:

```
PRINT "3+2=7"
```

and hit RETURN. You then see:

```
3+2=7
```

on the screen. Now, from recent experience you should be certain that the 64 is perfectly capable of adding 3 and 2. However, it is senseless to criticize the computer's judgment in the matter. Since it is dealing in words, not numbers, whenever you use quotation marks, it will accept and report on anything it is offered.

Mismatches

To return to the original distinction being drawn, computers segregate numbers and words. Generally, they regard words as something to be recorded and numbers as something to be recorded and/or crunched. What's more, they rebel when asked to manipulate one directly with the other. Just as it would be ridiculous to ask that TRUTH be multiplied by 6, it is useless to ask that the word three be multiplied by the number 2. Type:

```
PRINT "THREE"*2
```



"3+2" is a word. So is "FIVE." But 5 isn't.

and hit RETURN.

The response, TYPE MISMATCH, indicates that the computer cannot mingle words and numbers when performing an arithmetic (strictly numerical) function like multiplication. The "types" of the things you're working with "mismatch." Type:

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

and hit RETURN. Again, you have tried to match a word and a number. This is because BASIC regards "3" as a word. Now type:

```
PRINT "3 IS A NUMBER"
```

and hit RETURN. The computer prints these "words" for you on the screen:

```
3 IS A NUMBER
```

In fact, it prints all four words, including the "word" that looks like a number to most of us.

It's also interesting to observe that punctuation marks, numerical operations and blank spaces can be turned into words. Type:

```
PRINT ";      *"
```

and hit RETURN. Hit the space bar four or five times between the semicolon (;) and the asterisk (*). On the screen you see:

```
;      *
```

Note that the blank spaces have computer substance in the sense that they are recorded and delivered to you. They are words. Type:

```
PRINT "A B C D"
```

and hit RETURN. Hit the space bar two or three times between each letter.

And, just for fun, observe the letterless word. Type:

```
PRINT "      "
```

and hit RETURN. Invisible to the naked eye, your computer printed the spaces between the quotation marks.

At this stage, you have discovered that numbers and words or letters are incompatible. Your computer is stalwart in its refusal to manipulate one with the other. In a very real sense, the machine won't multiply words. Shortly you'll discover how desperately it loathes trying to spell numbers. However, it will print either words or numbers and will do so at the same time. Type:

```
PRINT "ADD 3+2:" 3+2
```

and hit RETURN. On the screen you see:

```
ADD 3+2: 5
```

The "ADD 3+2:" is printed as a word and the sum of 3+2 is printed as a number. Now type:

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

and hit RETURN. The computer responds with

```
3+2=5
```

Somewhat perversely, but with a logic you should now be beginning to grasp, "3+2" is a word and 3+2 is regarded as two numbers and an instruction to add them together.

Variables

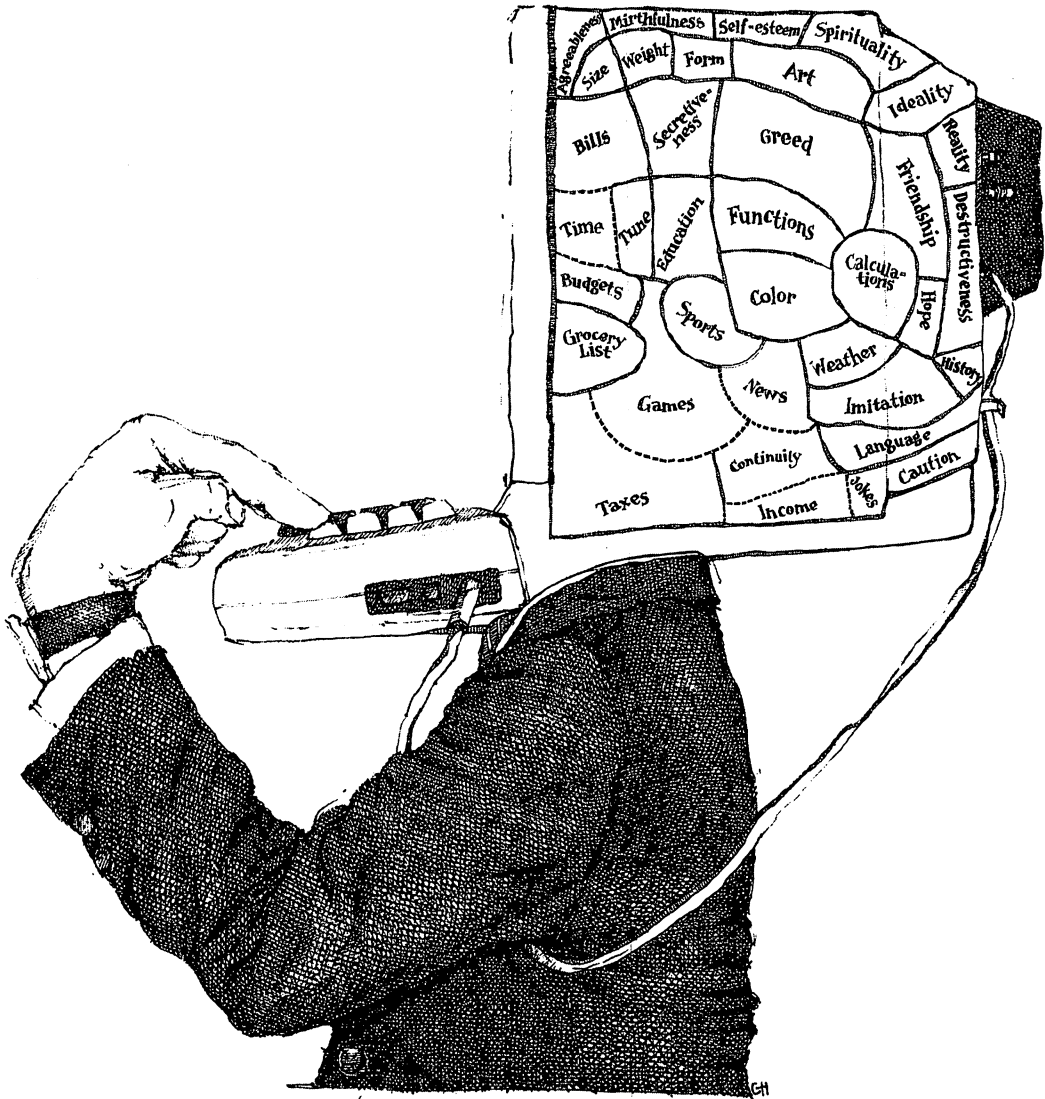
If you are developing a certain self-confidence because you now have mastered the distinction between numbers and words or letters, be prepared for a minor complication. There is another kind of character that the computer can deal with and it isn't a number, a letter or a numerical operation like "+." It is commonly known as a "variable." But we'll consider variables somewhat obtusely, even archaically, likening them to the great drawings of the human head that phrenologists created in the nineteenth century. You've probably seen such maps of the human mind, emotions, and memory and thought processes cordoned off like misshapen squares on a checkerboard. Phrenologists believed, you see, that certain parts of the brain held certain functions, were the repositories of facts, figures and mental actions. Knowledge, they believed, had location. Physical, actual location.

The idea that information can be stored in a place has seemed odd for many years. The two concepts don't seem compatible. Still, one often hears, "Somewhere in the back of my mind I remember your face." It would be stranger to hear, "I have stored my recollection of your appearance in the brain cell labeled DISTANT MEMORY No. 4657" but, actually, the two remarks are much the same.

The idea of giving information location is central to working with computers. Seen two-dimensionally, there appears to be nothing odd about an accountant writing numbers on a checkerboard-like grid so that a discrete fact can be given a particular location. However, modern computers go one significant step beyond an accountant's worksheet: they establish locations as concepts and allow you to substitute one number for another in each location.

This last idea may seem less than monumental at first. Nevertheless, it is crucial to understanding how computers store and work with information. So, we'll dwell on the thought for a moment.

Consider the question of your income. From every mature adult's point of view, income is an important concept. It is not a particular amount of money, goods or services received. It is, instead, an idea. Changing the particular number associated with your income will alter the way in which you live. But it will not change the concept of income or the basic manner in which



Computers contain memory locations very much like those on a phrenological map.

it is important to you. Thus, in your mind, the idea of income is different from the amount \$25,000. At some point the two might well be the same. But the idea simply contains the number temporarily, until your next raise.

Computers have the wonderful capacity to store and work with immense labyrinths of ideas. Provided you can construct a logical geography of concepts, interrelating them precisely and giving each a discrete location, then one specific example of an idea in question works its way through a computer program just as well as any other specific example. Changing your income and tax bracket will affect the way you live but it will not alter the manner in which your income benefits your standard of living or change the way taxes affect your disposition.

To return to the image of a nineteenth-century phrenological skull, recall that the feeling of greed was given a sizable and prominent domain. But the zone labeled greed on these maps did not indicate the specific material thing this chunk of mind might lust after. Instead, the ability to lust was simply located. Now, we all know that greed can hamper logical thinking, motivate someone to superhuman efforts and compromise one's sense of right and wrong. Such relationships exist regardless of whether one feels greedy toward money, automobiles or artichoke hearts. And at the same time, we all know that one's sense of right and wrong is less compromised by artichoke hearts than by large sums of money. The process is the same but the results vary by degree.

Computers contain memory locations very much like those on a phrenological map. The only difference is that you can program the meaning of the locations. That is, you establish a location by giving it a name and then you put something into that location. Labeled properly, the location will never change and its contents will be neither lost nor altered.

From the discussion of numbers and words earlier, you know that computers draw a strict distinction between the two. In the business of assigning locations a label, this distinction is just as strict and absolute. A location or variable can contain either numbers or words, but not both.

Since we've recently investigated a series of mathematical calculations, it seems only fitting that we consider the business of words for a while. Thus, we'll establish a few memory locations in your 64 and put some words into them. Before we begin, however, please make a note of the following bits of jargon.

VARIABLE: Variable is the word used to describe locations in a computer's memory. The actual site of remembering is called a variable because you can vary what you put in any particular location.

STRING: A string is a word. That is, a string is anything that your 64's BASIC has been programmed to regard as a group of letters, keystrokes, spaces or what-have-you that aren't numbers or the descriptions (labels) you've given a variable. Oddly, "3+2" is a string. 3+2 is not.

STRING VARIABLE: A string variable is the name given to a type of location in computer memory reserved for string occupancy. Put differently, it's a kind of location that will accept only words or strings.

Strings

Now that you have considered the three tidbits of computer argot a few times, suffice it to say that there is a certain logic to calling words strings. For instance, it stretches the meaning of "word" to include "3+2" in the category. Hence, the substitution of "string" for "word." Similarly, the description "location" only half implies the full power of a "variable." A variable is something you can manipulate, vary, alter, work with. It's a terrifically powerful capacity, as you'll soon see.

Now, think of that part of your computer which is designed to set up variables that contain strings (locations to put words in). This area of computer memory is very much like the dionysian portion of the human mind, dedicated to the forces of creativity and energy. Unlike the apollonian area of the mind with its analytical and logical thought processes, the strings you put into your computer can't be multiplied, divided, added or subtracted. Thus, they are only strings, not numbers. The variables that accept them will ingest characters of any kind, even numbers, but they will regard them as strings. Confused? Shortly all should become clear.

In order to give a variable a label you have to identify that label as one that will describe either a variable for strings or a variable for numbers. If the logic here seems convoluted, you're right. It seems sensible that a variable ought to accept anything you put in it and thereafter behave according to its contents. But computers don't work in that fashion. Instead, you must describe the variable's character before you put anything in it. If you do so improperly, the variable will rebel.

By way of demonstrating this annoying habit, type:

```
LET A="A STRING"
```

and hit RETURN.

You have been greeted with that message we encountered earlier, **TYPE MISMATCH**, because the variable A is reserved for numbers. You can't put strings in it.

Now, the folks who designed the BASIC language may well have been odd, scientific sorts with plastic shirt protectors in their pockets and slide rules on their hips, but they certainly didn't lack a sense of humor. When it came time to decide on a method for identifying those variables which are reserved for strings, they used a symbol that is universally regarded as numerical. Thus, whenever you want to describe a variable that is to contain strings (words or letters), you do so by putting a dollar sign at the end of it.

At this juncture we will return to Mr. Keats for a moment. Type:

```
LET BEAUTY$="TRUTH"
```

and hit RETURN. Then type:

```
LET TRUTH$="BEAUTY"
```

and hit RETURN. Next, type:

```
PRINT TRUTH$
```

and hit RETURN. Immediately

```
BEAUTY
```

appears. Finally, investigate the variable called BEAUTY\$ by typing:

```
PRINT BEAUTY$
```

and hitting RETURN. Naturally,

```
TRUTH
```

is produced.

Variable Idiosyncrasies

During the above exercise you established two string variables, BEAUTY\$ and TRUTH\$, and temporarily inserted the strings "TRUTH" and "BEAUTY" into these variables. As you might expect, the statement LET informs the computer that you are establishing a variable and defining its contents. Please note that the variable appears to the left of the equals sign. Reversing the order of appearance is not grammatically acceptable to BASIC. Try it. Type:

```
LET "TRUTH"=BEAUTY$
```

and hit RETURN. You are quickly scolded with

```
?SYNTAX ERROR
```

There's another interesting point about variable names. The variables BEAUTY\$ and TRUTH\$ are attractive and straightforward in our minds. But Commodore's BASIC doesn't see them as we do. It reads only the first two letters of the variable's name and the dollar sign. Thus, BEAUTY\$ is identical to BE\$ or BEAST\$. To demonstrate this to yourself, type:

```
PRINT BEAST$
```

and hit RETURN. The computer responds with

```
TRUTH
```

Now type:

```
PRINT TRITE$
```

and hit RETURN. Of course, you are given

BEAUTY

Not Keats, the 64 considers TRITE\$ precisely the same as TRUTH\$.

It will come as no surprise for you to discover that BASIC possesses a few other idiosyncrasies. Most of these have to do with the considerable number of commands in the language. These words, like PRINT, are reserved by the computer for use as commands and nothing else. Or almost nothing else. Type:

```
LET PRINT$="WITH OR WITHOUT INK"
```

and hit RETURN. You are told you have made a

```
?SYNTAX ERROR
```

Try:

```
LET POKER$="CARD GAME"
```

and hit RETURN. Or

```
LET BREAD$="ANTE MONEY"
```

and hit RETURN. Next type:

```
LET LONG SUIT$="WINNER"
```

and hit RETURN.

The syntax errors that are showing up all over the place are the result of your invading territory that BASIC has reserved for commands. Clearly, PRINT\$ uses the command PRINT in its variable name. Doing so is forbidden. POKER\$ contains the command POKE, BREAD\$ has READ buried in it and LONG SUIT\$ includes ON.

Happily, you needn't memorize all the reserved words in the BASIC language in order to avoid mislabeling a variable. Whenever you make some inadvertent error, the computer will inform you of the gaff. Thus, feel free to use reasonably descriptive variable labels so that you can remember why you put something where you put it. Reading through a program several days after you've written it can be a terrifically frustrating chore if you have to wade through a sea of randomly labeled variables trying to recall the logic of your work.

One brief note: reserved words aren't totally verboten. You can use them so long as they are part of a string. BASIC is indifferent to the contents of a string. For instance, type:

```
PRINT "PRINT"
```

and hit RETURN. No difficulty here. You need only be careful in labeling variables. Think of the problem as being the same as sitting in someone else's reserved loge in the opera. Sooner or later you'll be the one who has to stand up and move. In labeling variables, you'll be the one who has to think up a new name.

One final, fairly peculiar point. When BASIC was first written (at Dartmouth, by the way), its designers were interested in making the language as easy to “read” as possible. Thus, they concocted the notion of using LET to introduce the labeling of a variable and the description of that variable’s contents. Which makes perfect sense. After all,

```
LET BEAUTY$="TRUTH"
```

can be easily translated into “Let the string variable called BEAUTY\$ contain the bunch of letters TRUTH.” However logical and accessible this sort of command may seem to you, you would find it exceedingly tedious, after a few thousand lines, to type LET every time you wished to define a variable. Thus, virtually all modern BASICs, including Commodore’s, dispense with the LET. Type:

```
A$="LET IS GONE"
```

and hit RETURN. Then type:

```
PRINT A$
```

and hit RETURN. At your option you may use LET or ignore it. The 64 will ignore it in either case.

Not many pages earlier we made the glib observation that it is relatively senseless to try to multiply words. It was also pointed out that adding, subtracting and dividing them was fruitless. This is only true in part. To be sure, you cannot add two words together in the same way you add numbers. A third word is not created. However, you can tack the contents of one string variable on to the contents of another. For instance, type:

```
PRINT TRUTH$+BEAUTY$
```

and hit RETURN. The resulting display on the screen,

```
BEAUTYTRUTH
```

is rather ugly. The 64 has literally squashed the two words together, making a single mess on the screen. There are several ways to correct this situation and before long we’ll get to them. At this point, however, you might try inserting a space into the strings in memory. For instance, type:

```
BEAUTY$="TRUTH "
```

and hit RETURN. Then:

```
TRUTH$="BEAUTY "
```

and hit RETURN. And check your work by typing:

```
PRINT TRUTH$+BEAUTY$
```

and hit RETURN. Better, don’t you think?

How One and Two Equal Two

Before continuing, you should contemplate a rather remarkable occurrence that you have just witnessed and probably overlooked. Moments ago the string variable TRUTH\$ contained the word BEAUTY. But in order to separate BEAUTY from TRUTH on the screen's display, you redefined that string variable so that the contents of the variable contained a space as well as the word. It is interesting to ponder what became of the first definition, the word BEAUTY without the space.

The answer to this question is simple enough: it disappeared. Vanished. It has been electronically zapped into nothingness by the 64's memory.

This point is important for a number of reasons. But the most crucial is to remember that the metaphor of "locations" in memory is just that, only a metaphor. Similarly, computer types sometimes refer to these "locations" as "buckets." However, buckets, like locations, are physical entities that conjure up certain images in our minds. And it seems logical that if you put a cup of water in a "bucket" and then put two more cups in it, there will then be three cups in the bucket. This is certainly true of the galvanized variety of bucket. It is not true of a variable. The addition of more information into a variable first erases the original contents and then writes the new contents. Try this. Type:

```
BUCKET$="ONE CUP"
```

and hit RETURN. Then type:

```
PRINT BUCKET$
```

and hit RETURN. Your 64 responds, displaying

```
ONE CUP
```

Next, type:

```
BUCKET$="TWO CUPS"
```

and hit RETURN. Finally, type:

```
PRINT BUCKET$
```

and hit RETURN. The screen quickly reports

```
TWO CUPS
```

Your original cup has evaporated in the twinkling of a microprocessor.

This same fate awaits numbers. Type:

```
A=1
```

and hit RETURN. Now, type:

```
A=2
```

and hit RETURN. Finally, type:

PRINT A

At once you see

2

on the screen.

It is entirely possible that you regard this enforced forgetfulness as extremely limiting. It is not, however. Your computer can be made to recall almost anything, combining and recombining information with staggering flexibility. It just can't manage the feat in a single variable location. You need a batch of them. To investigate this process, let's use numbers for a while. Those offered below were typical prices in the summer of 1983. Just for fun, let's tally the average investment one might have made in a Commodore 64 at that time. Type:

$A = 199.95$

and hit RETURN. This variable, since it lacks a dollar sign, is used to record numbers, in this case the retail price of a 64. Next, type:

$B = 79.95$

and hit RETURN. You've now stored the cost of a tape cassette player. And, since you might well be a bookish sort, not inclined to watching television, you would have had to acquire a color TV in order to be able to blast rocket ships out of the sky on your computer. Since this is an imaginary exercise, let's buy a Sony. Type:

$C = 499.95$

and hit RETURN. Naturally, you'll need paddles for your computer, several hundred dollars' worth of arcade-style games and an assortment of how-to manuals like the one you are reading now in order to get the most from your computer. But we'll ignore such expenses for the time being. To calculate your entire investment, you need a variable in which to store the total price. Type:

$SUM = A + B + C$

and hit RETURN. Now, type:

PRINT SUM

and hit RETURN. The screen display indicates that your investment to date has totaled \$779.85.

Of particular interest in the above exercise is that you've stored information in memory in discrete locations. Each variable houses a specific number. What's more, you've combined the contents of the original three variables in a fourth location, called SUM. The integrity of the original numbers hasn't been compromised, yet you've performed some useful work by combining them.

To investigate the unaltered status of your original numbers, type:

PRINT A

and hit RETURN. A perfectly untouched

199.95

appears. Variable A has been worked with but not changed.

What Isn't There

You may have asked yourself what resides in a computer's memory locations the instant you turn the thing on. The answer to this question is at once outrageously complex and fairly simple. In the interest of time and space, the complex explanation will be shunned here. Instead, we'll concentrate on those locations that can be defined by Commodore's BASIC language in the form of variables.

Since a location in memory or a variable for storing numbers can be defined using any two letters or numbers (so long as the first digit is a letter), it stands to reason that there are carrels of numeric memory in the 64. And in the past few pages we've managed to insert a relatively small amount of information into memory. The contents of the as yet unused locations is, therefore, a mystery.

At this point you should learn, and commit to a kind of profound resolve, the single most important thing about working with computers: if you wonder something about them, ask the computer. By way of example, and since we've not defined the variable Z, request that the 64 inform you as to the contents of Z. Type:

PRINT Z

and hit RETURN. The computer instantly prints

0

Because Z is a numeric variable, its contents will always equal zero until you put something into that variable. This is the case with all numeric variables.

String memory is unique in this regard in that it equals nothing (in the sense of blank space) until you begin to assign values. Type:

PRINT ELEPHANT\$

and hit RETURN. While nothing particularly dramatic appeared on the screen in response to your request, the computer did in fact display the contents of ELEPHANT\$. But, as you know, the variable contains nothing. Which appeared. Note that there is a two-line space between your request and the READY prompt:

PRINT ELEPHANT\$

READY

The extra space is the screen's way of printing nothing at all. To demonstrate this more clearly, first establish a string variable with a dotted line in it. Type:

```
X$=" - - - - -"
```

And then hit RETURN. Then, type:

```
PRINT X$:PRINT Y$:PRINT X$
```

This last set of instructions can be translated to read, "First print the contents of X\$, then on the next line, print Y\$. Finally, on yet another line, print X\$ again. (Note that the colons allow you to put one instruction with another.) Now hit RETURN.

The screen then appears as below:

```
- - - - -  
- - - - -
```

The space between the two dotted lines is the computer's display of nothing, Y\$.

Naming Variables

Somewhat earlier it was pointed out that you can label a variable using just about any combination of letters and numbers you wish. You already know that the 64 will regard only the first two as significant and will check the end of the label to discover if a dollar sign is there. From a functioning standpoint, all the other letters are ignored. They may help you identify a variable, but they play no role in the 64's handling of them.

You should also know that numbers as well as letters are acceptable in labeling variables. The only restriction is that the first digit has to be a letter. To demonstrate, type in the following, being sure to hit RETURN at the end of each line.

```
HH=5  
H=4  
H3=10  
PRINT HH  
PRINT H*H3  
PRINT H3*HH
```

As with the use of certain commands in the labeling of variables, there are some taboo letter combinations that are peculiar to the 64 and therefore can't be used in the naming of variables. ST and TI are units of Commodore's BASIC and are therefore out-of-bounds. Try to remember to avoid them in your programming. If you forget, of course, the computer will remind you of your error.

An Oddity

There is a third sort of variable you should know about. It is called an "integer variable" and has the interesting habit of ignoring anything that appears following a decimal point. Type (being sure to notice the percent sign in the variable):

```
FR%=9.999
```

and hit RETURN. Then, type:

```
PRINT FR%
```

and hit RETURN.

```
9
```

appears before you.

Yet again, the Puck of computerdom makes a guest appearance. The percent sign indicates your wish for an integer variable, no decimals allowed. If you enter them, they are simply disregarded. And, of course, the folks who bring you BASIC have therefore elected to use this decimal-linked symbol to represent the abandoning of decimals.

You should note the manner in which the 64 interprets an integer variable. It doesn't make any value judgments or presume to round off your numbers. It merely reads the number until it reaches the decimal point and promptly truncates it. If you're working with integer variables, `VA%=9.99999` will never be rounded to 10.



CHAPTER 3

All of the keyboard calculating that you have been executing thus far involves an interesting sense of order and immediacy. Typically, whatever request you make of your computer is granted at once. At the outset, of course, this may have seemed perplexing. The instruction

A=100

followed by the hitting of the return key yields apparently nothing at all. The computer simply responds by announcing that it is once again READY. Of course, you now know that you have in fact established a numerical variable labeled A and have assigned it a value, 100. You also know that, in spite of the fact nothing appeared on your screen, some tangible change in your computer's memory has occurred.

This alteration of memory happened the moment you hit RETURN. The change was accomplished in an instant and will remain in place until you either redefine A or blank the computer's memory. (More on this enforced forgetfulness shortly.)

The point here is that your computer performs its assigned task the instant you instruct it to do so. The chore having then been completed, the status of the variable at hand remains unchanged thereafter. Because such instructions occur instantly and automatically, they are known in computer jargon as operating in the "direct mode" or "immediate mode." These particular descriptions are quite apt. However, this method of addressing your computer offers profound limitations. For instance, a computer does

only what it is told to do. It draws no inferences whatever. An example will help demonstrate this annoying tendency. Type:

NEW

and hit RETURN. You have just caused the memory of your 64 to forget, or disappear, anything that might have been stored there. Typing NEW is a good habit to get into whenever you are about to begin a new task. Forgotten bits of programming or overlooked variables may reside in memory. If you leave them floating around in there, you could easily confuse yourself by creating relationships that you hadn't intended.

Now type:

A=1

and hit RETURN. Then:

B=2

and hit RETURN. And:

C=3

and hit RETURN. Finally, type:

D=A+B+C

and hit RETURN. Now, to discover that $1+2+3=6$, type:

PRINT D

and hit RETURN. The 64 delivers you a 6.

As was mentioned earlier, establishing variables in this way and then assembling (adding, subtracting, etc.) them in a separate variable allows you to do work with numbers without permanently altering them. Better yet, it would appear that the definition of D, the sum of $A+B+C$, will remain equally unchanged regardless of any subsequent changes to the value in any of the three prior variables. Thus, D should constantly change as the values of A and B and C are changed.

Unfortunately, this is not exactly the case. Type:

A=10

and hit RETURN. Then, type:

PRINT D

and hit RETURN. In spite of the change in the value of A, variable D still equals 6. Clearly, the status of D does not reflect the changed status of A. If it did, D would equal 15.

The difficulty here lies with the nature of the direct mode. When you are working in it your instructions are taken literally and in their most limited sense. The moment you typed $A=10$, the value of A was changed. However,

only the value of A was altered. Nowhere in your command was there any mention of recalculating the value of D. Since you didn't "address" that variable in the sense of ordering it to refigure its value, D remained blithely indifferent to the change you made in A.

Put differently, at the time you wrote the original definition of D, you commanded that variable to be assigned a value equal to the sum of A, B and C. When you hit RETURN following the definition, your computer queried those variables, retrieved the values in them, added them together and stored the sum in D. When you changed A, you did not also inform variable D of this alteration. Thus, not having been asked to re-inquire as to A's value, D can hardly be expected to do so.

It should be obvious to you, therefore, that the order in which you ask a computer to perform a task is crucial. Precisely the information you wish to be employed in the calculation of any result has to be in memory before the result is derived. That is, everybody has to know what everybody else is doing or the parade will resemble a fire drill in the dark.

The solution to this difficulty is simpler than you might expect. To be sure, it is important to consider the order in which computers perform their duties. But since the machines are so blindingly fast, you can simply ask them to recalculate everything, absolutely everything, every time you make a change in one variable at hand. The mechanism that allows you to do this is known as a BASIC program. Just for the record, here is a bona fide definition of the beast:

BAS•IC COM•PUT•ER PRO•GRAM — A set of instructions for solving a problem, written in a series of BASIC statements, all preceded by line numbers. The line numbers tell the computer the order in which to execute the statements. Many programs involve the collection of data, its subsequent processing and the presentation of its results.

Line Numbers

The salient feature in the above definition is the question of "line numbers," or, more to the point, "lines."

A line is a bunch of stuff typed into a computer followed by the hitting of the return key. PRINT A is a line. It is a perfectly valid instruction that your computer can both understand and perform. If you type PRINT A and then hit RETURN, your computer will perform the act immediately. If, however, you grace this particular line with a number, you will have created another sort of creature altogether. Type:

```
10 A=10
```

and hit RETURN. Then type:

```
20 PRINT A
```

and hit RETURN. The computer has now ingested two "lines" of "code." It

has stored each line in its memory and will execute the instructions contained in them whenever you ask. Type:

```
RUN
```

and hit RETURN. Presto, you are presented with:

```
10
```

Your computer has searched its memory for all the numbered lines that are stored there and has performed the commands on those lines. Thus, it has first set up a variable labeled A and deposited a 10 in it. Then it located line 20 and printed the contents of variable A.

You might make a mental note that the definition of a line of code (code in this book meaning anything you write in BASIC) is always “punctuated” by hitting RETURN at the end of the line. Hitting the return key sends the information on the line into memory. Since this is the case, we will no longer ask you to “Type” and then “hit RETURN.” Whenever you see a line of code in slightly boldface type, you may regard it as ending with an instruction to hit the return key. So, type the line and then hit RETURN. No need for us to clutter this text with that redundant instruction. Of course, a “line” in this book may well wrap around your screen, filling more than one line there. Within certain limits, none of which are exceeded in this text, the Commodore 64 couldn’t care less. What happens on the screen is what happens on the screen, but a line of code only ends when you hit RETURN.

A Program

Now, after much preparation, we’ll write a small program. Type it, line for line, as it appears below.

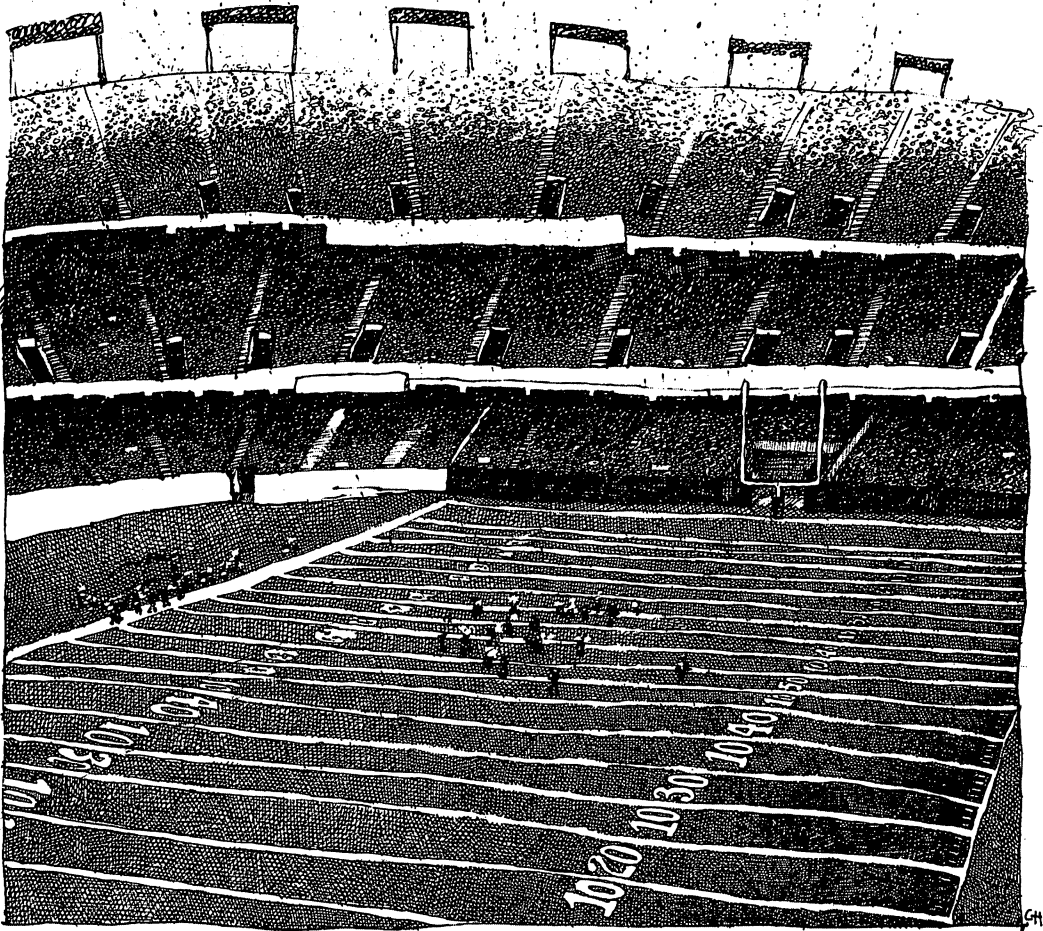
```
NEW  
10 A=5  
20 B=3  
30 C=A+B  
40 PRINT C  
RUN
```

The screen then displays an

```
8
```

This infinitesimal bit of programming ought to be decipherable to you. Still, for practice, you should try to “read” the program, line for line, and translate it. This business of translating programs and program fragments will soon become second nature to you. And to make things easier, BASIC has a terrific function built into it so that you can snoop around your programs whenever you like. Type:

```
LIST
```



Line numbers have nothing to do with football.

(We assume you hit RETURN.) You should see the program on your screen exactly the way you typed it a few minutes ago. The 10, 20, 30 and 40 at the beginning of each line are the program's line numbers. These identifiers are integral parts of a program that's written in BASIC. They are crucial because a program is executed in the order of its line numbers. That is, the computer searches for the lowest line number in memory and then performs whatever is written on that line. Then it searches for the next highest line number and performs the instructions written there. This process continues, on a line-by-line basis, until the computer has run out of lines to perform. It then stops.

The particular four-line program you see before you can be translated as follows: Line 10, the first line in memory, establishes the variable A and sets its value at 5. Line 20 describes variable B and sets its value at 3. In a nutshell, the first two lines collect data for the program. Line 30 involves the processing of this data. It adds A and B and deposits the sum in C. The last line reports on the result of the processing. It commands the 64 to print the sum of the variables A and B.

You should know that we might have assigned the first line in this program the number 1 and the second the number 2 and so on. But it's generally a good idea to assign line numbers in increments of ten, as we did. You'll probably find yourself tinkering around with even a simple program, and having the extra space at your disposal will save much time and energy later on.

Now, let's change the value of one of the variables. Type:

```
10 A=10
```

and run the program (type RUN and then hit RETURN).

Unlike the difficulty in the immediate mode that resulted from such a change, our program performs its duties perfectly. This occurs because all the collecting of data, its processing and the reporting of the results are performed from start to finish every time we type RUN. Type LIST again.

Please note that line 10 has been changed. It originally read "10 A=5" but now reads "10 A=10." Just as when you alter the value of a variable the original value is disposed of, so too the rewriting of a line in a program disappears the original line.

Built-in Help

There are a substantial number of other aids built into the BASIC language. For instance, type:

```
40 PRINT C
```

being especially careful to misspell PRINT. Then run the program. As you can see, the computer informs you that you have made a syntax error in line number 40. Obviously, this is a fairly useful tool when working with a short program. But when you begin to concoct much longer constructions, you will

find yourself extremely grateful for this pinpointing function. You should fix line 40 before we proceed.

Your computer will also put your lines in order for you. Try typing this:

```
50 PRINT D
30 C=20
40 D=A+B+C
```

Then list the program. As you can see, regardless of the order in which you write a program, it will be listed and executed sequentially, by line number.

At this point your 64 may be beginning to seem a fairly substantial piece of machinery. It is adding numbers for you with wonderful speed and accuracy. And since you are familiar with the other mathematical functions available to you, you might easily rewrite this program to raise any number to the power of any other number. For those who might feel somewhat intimidated by that particular observation, such a program appears below.

```
NEW
10 A=5
20 B=3
30 C=A ↑ B
40 PRINT C
```

Still, the rewriting of entire lines in order to alter a single variable in a calculation is clearly tedious. Should the numbers become cumbersome, the process could easily get to be a nuisance. Some remedy has to be found for this problem.

One other note before moving on. In the short programs we've written so far, a variable has been established wherein the data is added together. This variable doesn't do anything on the screen. It simply gathers the existing data and then performs the work we want done. After that calculation is complete, another line causes the result (in the form of the variable) to be printed on the screen.

This style of programming uses what is known as an accumulator, or a variable that holds various calculations, accumulating them until they are needed for further calculations or printing on the screen. The use of an accumulator is as old as the first electronic computer, ENIAC. And it is often used in modern, ultrasophisticated work. But it isn't always necessary. For some applications you can shortcut this extra, accumulating variable. As evidence, try the two programs below. Remember to type NEW before entering each one.

WITH ACCUMULATOR	WITHOUT ACCUMULATOR
10 A=3	10 A=3
20 B=5	20 B=5
30 C=A+B	30 PRINT A+B
40 PRINT C	

Working With Strings

Having looked briefly at programs involving numeric variables, let's see how string variables are treated in a similar program. Type NEW and then enter the program shown below.

```
10 A$="JOHN"  
20 B$="BROWN"  
30 PRINT A$+B$
```

Before you run this program, take a moment to review some of the principles of string variables. As in the earlier numeric programs, lines 10 and 20 are collecting and defining data. In this particular case the data consists of words, or strings, and so each variable is described using a dollar sign. Also, the strings in question are surrounded by quotes. Were they not, a TYPE MISMATCH would appear on the screen the instant you tried to run the program. Line 30 demands that the contents of A\$ and B\$ somehow be merged and printed on the screen. As you know from our experiments of this sort in the immediate mode, it is likely that they will be joined end to end without a space between the two words. Run the program and find out that this is the case.

Our earlier solution to this ungainly treatment of words was to insert a space after each word inside the quotation marks. But there is another method of accomplishing the same end that won't require you to make this effort every time you wish to change a string variable. Type:

```
30 PRINT A$+" "+B$
```

It is important that you hit the space bar between the opening and closing quotation marks. Now run the program and you'll see:

```
JOHN BROWN
```

You may be wondering why BASIC doesn't insert a space after a word automatically. After all, every word is followed by a blank space. That is, unless it is followed by a comma, quotation mark, period, etc. And, there may be times when you'd like to manipulate text so as to join words or letters together. Consider the following example:

```
10 A$="ADAM"  
20 B$="ANT"  
30 C$="HEAVY"  
40 D$="SET"  
50 E$="CAVE"  
60 F$="MAN"  
70 PRINT A$+B$+" "+C$+D$+" "+E$+" "+F$
```

This program illustrates the absurd, but it does demonstrate how you can manipulate strings. There are other ways to have your computer write ADAMANT HEAVYSET CAVE MAN, but we'll discuss them anon.

Your screen and your computer's memory may well be something of a mess at this point. So type NEW. As a matter of habit, pause for a moment between the keyboarding of the word NEW and the striking of the return key. Try to accustom yourself to this pause since it may prevent you from sending important programming to computer heaven long before you wish it to depart.

To obtain a fresh screen, hit the shift key with one hand and the CLR/HOME key with the other. The blinking cursor should then appear at the upper left corner of the screen and nothing whatever should clutter the view.

Mixing Strings and Values

Having now worked with both mathematical calculations in a simple program and strings, perhaps we should try combining the two in some useful way. Enter the following program.

```
10 A=5
20 B=3
30 C=A+B
40 PRINT C
```

Then run the program. Sure enough, five plus three equals eight. However, line 40 doesn't make this entirely clear. It simply informs us that the result of the program's work is to print the digit 8 on the screen. A more thorough explanation of what's occurring might be helpful. Amend line 40 such that when the program is run you see what happened in the development of the result. One solution appears below.

```
40 PRINT A"+"B"="C
```

This line is commanding the following to be printed on the screen: the value of A (5 in this case), a plus sign, the value of B (3 at this point), an equals sign and then the value of C (which is A plus B). It is necessary to ensconce both the plus and equals signs in quotes in order for them to appear on the screen. Symbols, just like letters, words and spaces, are printed out only if quotation marks surround them.

Now list the program so you can consider its new form. Then run it and watch the program issue the following report:

```
5 + 3 = 8
```

You may wonder, as you gaze at the tidy form of this program's display on the screen, why there are spaces between the numbers and symbols that appear there. Line 40 of the program seems to request that all the material to be displayed be squeezed tightly together. And, more perplexing, when we worked with string variables before, the issue of separating words required some real effort to resolve.

The explanation is straightforward. BASIC always identifies the "sign" of a number. That is, positive or negative, greater than zero or less. But in the

case of positive numbers, the sign is not printed, even though a space is left for it. To illustrate this, enter the following:

```
20 B=-5
```

(That's a negative 5.) Run the program.

As you can see, our program isn't as elegant as it first seemed to be. You might also note that zero is allotted a space for a sign but the 64 considers the number positive.

READ and DATA

At this stage we've used both numerical calculations and strings in a single line of a program in order to explain on the screen what's going on in the program. Though the process is now more clear, working with the program is altogether cumbersome. In order to vary either of the numbers to be added together, we must rewrite a line of the program. In order to change both variables we must rewrite two lines. Obviously, if we were trying to calculate the angle of re-entry of the space shuttle as it fell like a rock from the skies, this might prove a perilously slow process. Fortunately, BASIC offers a better way of putting data into a program. Try making the following changes in the program as it now exists.

```
10 DATA 5,3  
20 READ A,B
```

When you list the program you should be greeted with the following:

```
10 DATA 5,3  
20 READ A,B  
30 C=A+B  
40 PRINT A"+"B"="C
```

The new line 10 is called a DATA line. The statement DATA always is followed by a bit or bits of information that are assigned variable labels on the READ line. That is, the command READ can be translated into something along the lines of, "Find the DATA line and assign the first bit of information in it to the variable location that is described in the first slot following the command READ. Then, pick the second bit of data and assign it to the second variable. Continue until you have exhausted the variables in the READ line."

For the sake of clarity, lines 10 and 20 of the program now deposit the number 5 in variable A and the number 3 in variable B. Were there more variables and more data, more assignments would be made.

The information in a DATA statement might be numbers, words or a melange of both. All that is required is that each unit of information be separated by commas.

The beauty of the DATA and READ statements is that they make it easier for you to plug in new numbers for addition. You now need retype only one line to change all the components of a calculation. If you haven't already,

run the program. Now, try this change:

```
10 DATA 4,6
```

Having run this alteration, try:

```
10 DATA 678.84,1987.8635
```

More Idiosyncrasies

One of the quirks of string variables is that anything labeled as such is read by BASIC as a word, not as a value or variable. Which is to say that "A" is a letter (string) and A is the label one can assign to a numeric variable. This isn't too difficult a concept to swallow, except when numbers are identified as string variables. Consider the following example (don't forget to type NEW before entering):

```
10 DATA 1,2,3
20 READ A$,B$,C$
30 D$=A$+B$+C$
40 PRINT D$
```

Before running this program, please contemplate the fact that BASIC might respond in any of three ways to this program. First, it might add the numbers together and print a 6. Or it might regard the numbers as strings and print all three. Or, since the information in DATA lacks quotation marks and the variables in the READ command are string variables, BASIC might conclude that there is a TYPE MISMATCH involved. Run the program.

As you can see, your answer is 123. The data on line 10 was read as strings, even though the numbers all lack quotation marks. Being strings, they were "strung" together, not manipulated. Granted, this seems to confound the normal rules of working with strings and string variables. Such is the peculiarity of READ and DATA.

Examine a variation on the READ and DATA theme.

```
10 DATA 1,2,3
20 READ A,B,C
30 D=A+B+C
40 PRINT D
```

Now, of course, the result is 6. We've crossed the road and have entered the numerical hemisphere of the 64's memory. In that area of grey matter, numbers are possessed with their values so the addition of the three integers in DATA yields their sum. For the sake of interest, make a data substitution similar to the one attempted while dealing with string variables. Type:

```
10 DATA 1,"2",3
```

and then run the program. From the result, a syntactical error, you should conclude that string variables will gobble up just about anything, but that

numeric variables are much more choosy. However, there are often cases when you will want to combine numeric and string variables in READ/DATA statements.

```
NEW
10 DATA SHOES,47.50,TROUSERS,23.50
20 READ A$,A,B$,B
30 PRINT A$;A
40 PRINT B$;B
50 PRINT "TOTAL = "A+B
```

In the above program, SHOES, a string, becomes the contents of variable A\$, and the value of 47.50 is recorded as the contents of variable A. The string variable B\$ reads in the string TROUSERS, and B, the fourth variable to read in a value, takes 23.50 for its contents. Run the program.

The resulting screen should look something like this:

```
SHOES 47.5
TROUSERS 23.5
TOTAL = 71
```

One can quickly see how extra data could be added to this program as additional clothing purchases were made over time. However, one perplexing element was introduced here that could be causing some trouble: the semicolon in the PRINT statements.

Working With the Screen

A semicolon is used to separate different elements of a line. This bit of punctuation is used when the two or more elements are either to be considered or printed one directly after another. That is, without pause or spacing. The punctuation is necessary because BASIC could easily confuse variable labels if they weren't distinguished one from the other. For instance, PRINT A B would be read by your computer as "Print the variable AB" while PRINT A;B reads "Print the variable A and then print the variable B, leaving no space between the two."

In the case of our garment program that totals the price of shoes and trousers, one's eye is disturbed by the fact that the dollar amounts don't line up in a tidy vertical column. But all things are possible. You just need the proper punctuation. The problem here arises because the lengths of the strings in question—shoes, trousers and total—are inconsistent. So when you squeeze the numeric variables next to them, a somewhat ragged screen results. Substitute the following lines:

```
30 PRINT A$,A
40 PRINT B$,B
```

and run the program. Replacing the semicolons with commas improves the look of this screen considerably. The commas indicate to BASIC that the screen is to be divided into vertical columns. Eleven characters are allotted to each column. When the program runs, the first variable in the PRINT statement appears flush to the left-hand side of the screen. Then the comma goes to work and moves the second element in the PRINT statement to the twelfth character position on the screen. Then the second element of the PRINT statement is produced. (If you're counting characters, remember there is an invisible plus sign in front of the numbers we're working with.)

To further explore these columns, type:

```
PRINT "X", "X", "X", "X", "X", "X",
```

As you can see, the screen is divided into consistently spaced columns that organize material beautifully. What's more, BASIC takes into account the possibility that a number or string might be too large for one column. Type:

```
PRINT "X", "SOMETHING TOO LONG", "X"
```

BASIC skips a column as a result of the overflow and then picks up where it left off.

The order in which BASIC considers commands and statements has been discussed. In the immediate mode, it responds immediately, and only once, to your instructions. In that mode it does only what is being asked. In a line-numbered program all of the commands and statements in the program are executed every time you run the program. The computer begins with the lowest numbered line and proceeds, line by line, until everything has been done.

As a result, programs make repetitive executions of a process much more convenient. However, you must still pay attention to the order in which things happen. For instance, enter the following:

```
NEW
10 PRINT A
20 A=5
```

Run this program. Clearly, at the moment the computer arrived at line 10, no value had been assigned to A. So, therefore, the 64 dutifully reported that the value of A was zero. Immediately thereafter, this program assigned A the value of 5. Can you guess what will happen if you rerun the program? Try it. Variable A still equals zero. It would appear that the running of a program clears the computer's memory so that no previous information is assumed. If you recall, this is not the case in the immediate mode. To demonstrate, type:

```
NEW
PRINT A
A=5
PRINT A
```

Hide and Seek

It would appear that computers read programs from top to bottom. However, there are exceptions to this rule. By using several different types of commands, you can detour the workings of a computer as it proceeds through a program. One such detour is the READ/DATA statement. Regardless of the location of the DATA statement, READ will search for it. That is, the data need not precede the READ command. In fact, a READ command will search through an entire program, be it composed of 4 or 4,000 lines, until it finds a DATA statement to read. Observe:

```
NEW
10 READ A,B
20 PRINT A
30 PRINT B
40 DATA 5,8
```

Run this program. When the computer arrives at the READ command it searches for data. Finding it, the values 5 and 8 are read. Then the computer proceeds from line 10 to line 20 and then to line 30.

This phenomenon does not contradict the basic idea of proceeding from top to bottom in a program. Instead, there is simply a forced U-turn that the READ/DATA team is capable of making. READ won't make this detour unless it is required to do so, however. READ searches, from the very beginning of the program to the very end, until it finds data. And it will accept and work with the first data that it discovers. Add this line to the program:

```
5 DATA 1,3
```

Run the program and then list it. (Type: LIST.) Now, try this:

```
5 DATA JOHN,MARY
```

And run the program. The READ command isn't pleased with the strings it is discovering on line 5. What's more, it can't ignore this data and search for more appropriate data. Such are the rules.

But there are ways to use more than one READ statement in a program. Consider this:

```
NEW
10 DATA JOHN,MARY
20 READ J$,M$
30 PRINT J$,M$
40 READ A,B
50 PRINT A,B
60 DATA 4,7
```

In this case the computer makes a note to itself. A kind of internal memo is generated that annotates the use of the data on line 10. Since these strings are perfectly acceptable to the READ command on line 20, they are read cor-

rectly and then printed on line 30. Then the computer is instructed to read again on line 40. It consults its memo pile and is informed that the data on line 10 has already been read. So, the READ command on line 40 goes in search of other data. Finding it on line 60, and since it is appropriate to the numerical variables in the READ statement, the data is read and then printed on line 50.

One of the more attractive features of a READ/DATA program is the ease with which it may be revised. Thus, data is often isolated at the very end of a program so that when the program is listed, the data is the last material to appear on the screen. This makes revising quick and easy.

READ/DATA Relations

There are idiosyncrasies to this command, however. Try this:

```
NEW
10 DATA HOME,ON,THE,RANGE
20 READ A$,B$,C$
30 PRINT A$,B$,C$
```

Run this program. Your screen should now appear as below:

```
HOME      ON      THE
```

What you have in your program are three variables and four pieces of data. The 64 assigns the first bit of information to the first variable, A\$; the second bit to the second variable; and so on. Since you have not given a string variable in the READ statement that would read in the fourth bit of data, this data goes unnoticed by your computer.

Now type:

```
20 READ A$,B$,C$,D$,E$
30 PRINT A$,B$,C$,D$,E$
```

Now list the revised program and consider it. At this point you have a greater number of variables in the READ command than you do strings in the data. Run the program. On the screen you'll see:

```
? OUT OF DATA   ERROR IN 20
```

An OUT OF DATA error message means just that. In line 20 your computer was reading the data in line 10. It successfully paired A\$ with HOME, B\$ with ON, C\$ with THE and D\$ with RANGE. When the invisible eye of the READ command arrived at the last variable, E\$, there was no information in the data to correspond to it. You were literally out of new data to read so the error message flashed on the screen. Note that the program halted at this error. It did not continue to line 30 to print the data that had been successfully read.

The rule then is, you can have too much data if you wish. But READ is a form of demand. The computer is forced to obey and, if it is somehow prohibited or unable to do so, it will rebel.

Housekeeping

Having digested some of the principles governing the behavior of READ and DATA, it's now possible to use this statement to construct a somewhat more complex program. But, before doing so, we'll introduce a trick.

The PRINT command is a somewhat more esoteric instruction than you might think. On the face of it, PRINT means "display this on the screen." And, in fact, the command works very much that way in most cases. However, the literal meaning of PRINT is, "to cause the computer to react as though the following keyboard character has been pressed." In the case of letters, numbers, punctuation marks, etc., there is absolutely no difference between these two definitions. But they are different. Consider the key in the upper right-hand corner of your keyboard marked

CLR
HOME

Pressing this key causes the blinking cursor to jet its way to the upper left-hand corner of the screen. This position is known as HOME. If you hold down the shift key and press the CLR/HOME key, the cursor not only zips HOME, but the screen is cleared at the same time. Once again, the keyboard is changing the way the screen looks. (If you haven't tried these two functions, do so now.)

Clearing the screen and moving the cursor to the upper left corner is a very nice way of beginning something. No random marks appear anywhere. You are working with a tabula rasa. Thus, before you run a program it might be nice to hit the CLR/HOME key. The only problem is, you then have to type RUN to get the program going. And, of course, doing so leaves RUN printed on the screen.

The solution is to write a line into your program that will simulate the pressing of the shift key and the hitting of CLR/HOME. Computers handle this sort of function in a straightforward, if somewhat awkward, way. They number every key and character on the keyboard. Doing so allows you to refer to anything that can be done by hand by using the code number for the keyboard action. Observe:

```
NEW  
10 PRINT CHR$(147)
```

Then, before hitting the return key at the end of line 10, type a lot of garbage on the screen, filling it with random characters, words, complaints, whatever. Finally, hit RETURN, then type RUN.

Presto. This little program has cleared the screen. This is the case because the keyboard action in question is known to the 64 as CHR\$(147) or "the reserved string variable for character number 147." This PRINT statement can be translated as, "Now make the computer behave as though an invisible hand has pressed CLR/HOME." Take a moment to haul out the



A cursor is not a foul-mouthed follower of hockey teams.

booklet that came with your computer. If you flip to the appendix of ASCII and CHR\$ codes, you'll see that almost every key has a symbol that can appear on the screen and that each is numbered. What's more, actions such as RETURN, switch to lower-case and move the cursor can also be referred to. Try printing a few of these CHR\$. You can use the immediate mode if you like, not bothering to number the lines of your instructions. Simply type:

```
PRINT CHR$(97)
```

and you'll be blessed with a spade. Experiment for a while. And then input the following:

```
NEW
10 PRINT CHR$(147)
20 PRINT "TWO NUMBERS TO SUBTRACT"
30 DATA 8,9
40 READ A,B
50 C=A-B
60 PRINT A "-" B "=" C
```

Running this program will show you just how nicely the CLR/HOME CHR\$ is. What's more, you can see that the computer isn't bothered in the least when dealing with double negatives. Type:

```
30 DATA 8,-9
```

Try changing the data in this program yet again to produce some larger, more complex results. Insert some decimals if you like.

More Hidden Gems

Before leaving READ/DATA, you may have wondered what, other than the ASCII and CHR\$ codes, might be buried in your computer's memory. Well, the answer is a resounding one: lots. The 64 is packed with nifty information that you can call on in order to perform a wide variety of functions. We'll touch on only one now, but others will surface with stunning regularity as we proceed. Type:

```
PRINT  $\pi$ 
```

(The π symbol is the upper-case form of the \uparrow . To type it on the screen you must hold the shift key down and hit the \uparrow key.) When you hit RETURN the following number should have appeared on the screen: 3.14159265. This is a rounded form of a much larger number. And it is a useful one. π , pronounced "pie" and spelled "pi," is the number representing the universal relationship between a circle's diameter and its circumference. That is, if you measure the circumference of a circle and measure its diameter, you will discover that the circumference is 3.14159265 times longer than the diameter. This forgotten theorem of Euclidean geometry was once drilled into you by a teacher. Likewise, you were quizzed, perhaps many years ago, on the area of a circle.

There is a reasonable chance that, at the time of the quiz, you recalled that the area of a circle is discovered by multiplying π by the radius of the circle, squared. (Psst—the radius is half the diameter.) Written mathematically: $\text{Area}=\pi R^2$ or, to the 64: $\text{AREA}=\pi * R \uparrow 2$.

Among the dilemmas faced by clever young people in geometry classes is recalling pi. Most computers, the Commodore included, do away with this problem. All you have to do is remember the formula. Type:

NEW

```
10 PRINT CHR$(147)
20 PRINT "CIRCLES ARE EASY TO WORK WITH"
30 PRINT "IF YOU KNOW THE VALUE OF  $\pi$ "
40 PRINT "AND  $\pi$  ="  $\pi$ 
50 READ R
60 AREA= $\pi * R \uparrow 2$ 
70 PRINT "IF THE RADIUS =" R
80 PRINT "THE AREA =" AREA
90 DATA 5
```

Then run the program and you will be told that the AREA equals 78.5398164.

Type LIST and take a look at your program. Note that we use π as both a string, when we surround it with quotes and print it on the screen, and as a number, when we print it on the screen not surrounded by quotes. This same thing happens when we print AREA surrounded by quotes and when we print AREA as a result of the calculation.

Finally, notice that the data is at the very end of the program. This is the case because when you feel like computing the area of another circle, you can simply type LIST, find the data's line number easily and then type new data into the program. Try it.

Having worked your way through some fairly knotty stuff, pause for a moment. Take the time to translate into English the program that calculates the area of a circle. Put differently, read the program and follow its instructions as though you were the computer. Practice predicting what the 64 will do, simply by reading the program. Hint: when you get to line 50, remember that a READ command searches for data, finds it and then returns to the READ command.



Programming is not an arcane science exclusively reserved for that special breed of creature who used to carry a slide rule.

CHAPTER 4

At cocktail parties one is capable of distinguishing the normal people from the computer types by the way they refer to striking the keys of a keyboard. The normal folks tend to say things such as "Type the stuff." The computer people often can be overheard saying, "Input the data." Now, most of us know what is really meant by both remarks: "Type the stuff." And it is often slightly annoying to hear this simple, direct and totally accurate statement turned into a string of buzz words. "Data," after all, is really information, stuff, numbers, words and the like. And "input" is obviously an affectation of speech with no defensible reason for being.

Computer people are guilty of using this computer lingo for a number of reasons. First, they tend to be cliquish. Computer people often think that they are the possessors of a secret, as though they have discovered a very special world and are determined to keep it special by not explaining the place. Worse, computer people tend to be numerically oriented. They are comfortable with complex mathematical reasoning. As a result, they sometimes tend to be somewhat less than articulate. And so they have created a language that is, without question, an affront to any sensible person. This language is confusing, awkward, ugly and largely unimaginative. And it keeps the secret a secret.

Sad to say, the parlance in question has become an institution. There is probably no way to alter it. It has been erected slowly and in bits and pieces by a great number of engineers and mathematicians. It will not go away no matter how much we wish it to do so. Thus, we are stuck with it. And, as they say, when you find yourself with a bushel of lemons, make lemonade.

Having fooled around with READ/DATA statements a bit, you may understand one of the reasons computer types say "data" instead of "information." For a moment, let's pause to consider the word a little more.

Computer information is different from most other kinds of information primarily because it is structured in a way that the computer can understand and work with. Just as you can't type "add 2 and 3 together, please" and expect your 64 to respond, you can't multiply a string variable. So, if the computer is looking for a string variable in data and you offer a number, your number will be treated as a word. You've given the computer "information" but not "data." This much should be clear from our earlier fiddling with READ/DATA.

What you may also have concluded is that data can be generated in a number of ways. For instance, the PRINT statement can be inserted into a program to simulate the typing of a character or other keyboard action. We accomplished this feat when we began a program with the instruction

```
PRINT CHR$(147)
```

in order to clear the screen and move the cursor HOME. Similarly, data can be bounced off satellites, zipped through telephone lines, read from a cassette deck or disk and typed on a keyboard. The minuscule wiring of a computer is indifferent to the source of its electronic impulses. It only cares that the impulses course through its veins in proper form.

Thus, data is at once more specific than information and more general as well. The word "data" implies the propriety of the information's form and also suggests a wide range of sources.

The ever-annoying "input" is a similar word. With similar connotations. "Input" doesn't mean "type." Instead, it means "to cause to be ingested by an electrical system designed to gobble up such data." Since you don't necessarily have to keyboard data into a computer, one often refers to "inputting" it.

Obviously, the above explanations won't make you feel a lot better the next time some computer snob mumbles something about inputting data till three in the morning. But at least you'll now know that he was probably just typing the stuff.

Input

The slightly circuitous introduction to this chapter is not without its method. For we have worked with READ/DATA statements for some time now. And it should be clear to you that one of the benefits of a READ/DATA statement is that it allows you to plug new numbers into a program with relative ease. But you've probably also concluded that listing the program and retyping an entire data line is a bit of a nuisance. And one that will grow when programs become more complex, requiring more data. In order to alter the workings of such a program, you must change the program itself, not just the information (data) that is being manipulated. Clearly, there ought to be a way to allow the computer to operate independently. That is, the program should

accept new data, work with it, deliver a result and yet never be intrinsically altered in so doing. If this could be managed, time would be saved and things might well be more clear to the soul who was sitting at the keyboard.

Which brings us to our next BASIC command. It is a clear, uncluttered command that says exactly what it means: INPUT. The command is designed to work with variables, both numerical and string variables, in order to get data into a program without permanently sealing it there. But, as usual, the easiest way to explain a BASIC command is to run it on the computer. Type:

```
NEW
10 INPUT A
```

and then run the program.

The moment you type RUN there appears a question mark on the screen and your cursor blinks silently next to it. Now type:

```
4
```

and hit RETURN. All of a sudden you are confronted with the omnipresent READY as though nothing whatever had occurred. But the 4 you typed has gone somewhere. To find it type:

```
PRINT A
```

It is unlikely that the result of your request is particularly alarming to you. But as you'll soon see, the implications are interesting. For the data you fed into the computer was ingested by it and assigned the variable A instantaneously. Even more important, that variable will continue to contain that specific piece of data until you either change the data, turn off your computer or type NEW. In spite of the reliability and stability of this form of data entry, however, you in no way alter the underlying program by inputting data into it. For instance, type LIST and study the one-line program at hand.

Clearly, there is no 4 in it anywhere. Were you to rerun the program and answer the question implied by the question mark with a new number, such as 11, the program would not be altered. But the contents of the location in memory in the computer that has been assigned the variable A would change. The 4 would disappear and 11 would reside there. Next, we should use this versatile command to perform some useful work. Type:

```
10 INPUT A,B
20 C=A+B
30 PRINT C
```

Now when you run this program you will be greeted first by a single question mark. Type a number and hit RETURN. Suddenly you are confronted with two question marks. This indicates that you are being asked a second question. Type another number and hit RETURN again. In a flash the sum of the two numbers you input is printed on the screen. You may find it interesting to know that INPUT is quite forgiving in regard to your inputting. It will continue to ask you for data until all of the variables on the input line have

been assigned values. But if you ignore a query by hitting RETURN without having input a number, INPUT will assume you mean zero. Run the program again and try this out.

And there is yet another method of answering the question mark that INPUT generates. You can literally treat the INPUT question mark as the word DATA in a READ/DATA statement. That is, when you are presented with the question, type in your answers separated by commas on a single line. Run the program once more and respond to the question mark as below:

35,56

and then hit RETURN.

INPUT is quite simple and straightforward to understand. You know that a READ command searches the program for a DATA statement and then takes its data from that statement. The INPUT command produces the same sort of search, but INPUT looks to the keyboard for the data in question rather than to the program itself.

Input Problems

INPUT is not invariably flexible and forgiving. You will recall that when we constructed a READ/DATA statement wherein there was insufficient data, the computer complained by announcing that you were OUT OF DATA. A similar event can occur if you give INPUT too much data. Run the program that you now have in memory and type:

1,2,3

and then hit RETURN. Two interesting things happen when you do so. First, the 64 prints

?EXTRA IGNORED

on the screen. Then it prints

3

on the screen. The EXTRA IGNORED message is the computer's way of telling you that you have input too much data. That is, extra data. At the same time, you are informed that the computer has disregarded this error, ignored it. The 3 that is thereafter printed is the sum of the first two numbers you typed in. After all, this program adds two and only two numbers together. Please don't fail to note that the BASIC language doesn't erupt as a result of your error. The program ran to its conclusion and gave you an answer. But along the way it also informed you that a discrepancy had arisen so that you wouldn't assume the answer that was delivered dealt with the question as you posed it.

Run the program once again. Then answer the first query by typing:

JOHN

and hitting RETURN.

?REDO FROM START

now appears on your screen and the question mark is repeated. (You know this is the first query because there is only one question mark, not two.) This error message is quite sincere. BASIC had discovered a flaw in your answer and wants you to answer the original question again, redoing your data entry from the very beginning of the input process. In this particular case, the flaw that has been uncovered is the incompatibility of your answer with the variable that has been assigned to that answer. You have responded with JOHN and BASIC has discovered that this data is destined for a variable that accepts only values, not strings. So there has been a TYPE MISMATCH error created and the computer is asking you to try again.

Because there is a question mark in front of the REDO FROM START message, many people read it as a question. To further aggravate things, the computer puts yet another question mark on the screen and blinks at you, expecting you to answer. Invariably, newcomers to computers read the REDO FROM START remark as a query, feel flustered and, since they're being given the opportunity to start over, answer with a YES. Doing so only produces yet another REDO FROM START message because YES, like JOHN, is a string and the variables in the current program are numerical variables. So YES doesn't help. You will save yourself considerable frustration if you remember that this particular error message is a command. You are being told, not asked if you would prefer, to redo your data entry.

Hints With Input

The program as it now exists requires you to recall exactly what sort of input is required and to know what will happen to it once your typing is complete. This is clearly an awkward situation that needs fixing. Fortunately, it is the very nature of the INPUT command to be querying (known as prompting in computer argot) for an answer to some specific question. Thus, the folks who created INPUT built right into the command a method of identifying the question being asked. Try this:

```
NEW
10 PRINT "TWO NUMBERS TO ADD"
20 INPUT "1ST NUMBER";A
30 INPUT "2ND NUMBER";B
40 C=A+B
50 PRINT A "+"B "=" C
```

and then run the program. Pretty nifty, don't you think? INPUT has this added feature at no extra charge: it behaves like a PRINT command so long as you surround the thing to be printed in quotes and separate, with a semicolon, the printed material from the variable to be assigned a value.

INPUT statements are perfectly willing to absorb strings as well as values. In fact, they behave very much like READ/DATA statements in this regard. You need only distinguish strings from values by labeling the variable with a dollar sign. Enter the following program:

```
NEW
10 PRINT CHR$(147)
20 INPUT A$
30 INPUT B$
40 PRINT A$,B$
```

and run the program. In response to the first question mark, type your first name. In response to the second, type your last name. Rather quickly you will then see your name printed as below:

```
JOHN      BROWN
```

This spacing is the result of the comma that is inserted between the variables on line 40. As was mentioned before, the comma divides the screen into columns and forces the 64 to print each variable flush to the left of each quarter of the screen. Let's fix this. To change line 40 you simply rewrite it:

```
40 PRINT A$;B$
```

Now run the program. Alas, we're once again confronted with squashed-together words. The semicolon is doing its job perfectly and so JOHN BROWN is showing up on the screen. How about this:

```
40 PRINT A$;" ";B$
```

Be sure to hit the space bar once between the two quotation marks.

Mixing Inputs

Having experimented a bit with both string and numerical variables in the INPUT statement, it might be instructive to try combining them in a single program. Just for fun, let's build one that uses all the speed of modern computing to do some calculating. Along the way you'll also learn how to build a program in chunks. Or at least you'll start learning.

So you'll know in advance, the purpose of the following program is to calculate how many hours you've slept in your lifetime. Begin by personalizing the program so that whoever sits in front of it can tell us his name. Type:

```
NEW
100 INPUT "FIRST NAME";F$
110 INPUT "LAST NAME";L$
```

You have certainly noticed that we are beginning a program with the line number 100. This is a fairly standard approach that constructs a program in blocks. Since we may well wish to give the program a title or have it perform some action before asking for the guinea pig's name, line numbers are re-

served for such afterthoughts. Similarly, as we think up other things for the program to do, we'll write those lines in chunks, giving each general task an increment of one hundred.

At this point we've asked our friend at the keyboard to tell us his first name. We've taken his response and deposited it in string variable F\$. This variable is used because the contents of the variable, first name, starts with the letter F. Then we've done the same thing with his last name, assigning it the variable L\$.

To test if everything is working as we wish, run the program. Then type your responses as you are queried on the screen. When READY appears once again, type:

```
PRINT F$
```

and then type:

```
PRINT L$
```

If all is well, your name should be pulled from the confines of the two string variables and printed on the screen.

Next we must find out how old our keyboarder is. Since this is a delicate question in many circles, let's be terse about the matter so that it can be disposed of quickly.

```
200 INPUT "YEARS OLD";Y
```

And now, in order to add some fine detail to our calculations, we should discover how many months it has been since the subject last enjoyed a birthday.

```
210 INPUT "MONTHS SINCE BIRTHDAY";M
```

To check that all is well with your program so far, run it and answer all four questions. Then type:

```
PRINT F$,L$,Y,M
```

You should then see something like the following on your screen:

```
JOHN      BROWN      35      7
```

Note that we used commas in the PRINT statement to divide the screen into columns. This keeps the data we've input separate and easy to check.

Assuming that you've successfully programmed these first lines, it's time to discover how much time our friend spends sleeping during the day.

```
300 INPUT "HOURS OF SLEEP";HS
```

At this point we know just about everything necessary to calculate the total number of hours spent in slumber thus far in a lifetime. Of course, it is possible to ask how many days, hours and minutes have passed since his last birthday. And we could query the subject on the frequency and duration of his Saturday afternoon naps or the number of particularly boring meetings he has

attended in a semiconscious state. But for the purposes of this exercise, we'll stop at the present degree of accuracy. Now let's multiply the elements of this calculation. We know that there are 365.25 days in a year (average, considering leap years). And we know that there are, on average, 30.44 days in a month. (Derived by dividing 365.25 by 12.) Finally, we know that there are as many nights as days in the average life. So, to discover how many nights our sleeper has been around, we must multiply his age times 365.25 and the number of months since his birthday times 30.44 and then add these two figures together.

```
400 N=Y*365.25+M*30.44
```

Next, we multiply the total number of nights in question by the average number of hours slept per night.

```
410 SL=N*HS
```

Having gathered all the data we care to gather at this point and then having manipulated it as we wished to, it's time to inform our friend how many hours he has been asleep during his time on earth.

```
500 PRINT "BELIEVE IT OR NOT, YOU'VE SLEPT"  
510 PRINT SL;" HOURS IN YOUR LIFE"
```

Run this program, answering all the questions, and watch the screen carefully as you do so. Then critique what you see there. That is, try to figure out what you don't like about the way the program treats you.

First, the screen is messy. So, let's clear the screen at the outset.

```
10 PRINT CHR$(147)
```

Test this change.

Getting Personal

One odd characteristic of this program is that, though we asked the person typing at our keyboard to tell us his name, we did absolutely nothing with the information once we had it. How about this:

```
520 PRINT F$;" ";L$
```

Don't forget to hit the space bar once between the two quotation marks.

Skip

Still another annoying factor is the density of the screen. Everything seems to appear in a terrific hurry and is pinched together, one line immediately below another. Since we're dealing in upper-case letters here, that makes for very dense reading. So it might be nice to double-space the screen, skipping a line between some of the material shown there. Doing so is

remarkably easy. Since each new PRINT statement begins on a new line, all we have to do is print some "nothing" in order to skip a line. Try this and see how you like it:

```
120 PRINT
```

Now run the program. Questions about the subject's name are now divided from questions about his age. Better. Now type LIST and hit RETURN.

As you can see, BASIC has put line 120 in its proper place. In spite of the fact you have concocted this program "out of sequence" as it were, the computer runs your programming lines correctly. It is constantly managing your thinking, reshuffling your thoughts like a set of file cards. And so that you can see what you've done at any time, BASIC lists your program in its proper order.

Obviously, this is a terrific editing tool. You can add anything anywhere at any time. Provided you've got a line number available at the location in question, but more about that in a moment.

Edit

To demonstrate another editing function built into BASIC, type:

```
121 PRINT
```

and run the program. Now the screen provides two blank lines between the prompts for your name and your age. This, it appears, is excessive. To remove the offending line of programming, type:

```
121
```

and hit RETURN. Doing so erases the entire line. Since there is absolutely nothing on the newly typed line 121, BASIC knows to remove all prior programming on that line and then remove the line itself. Similarly, to change a line you simply type over it in the sense that you rewrite it. Try this:

```
500 PRINT "BELIEVE IT OR NOT, YOU HAVE SLEPT"
```

Running the program now will show you that YOU'VE SLEPT has been overprinted by YOU HAVE SLEPT.

Since adding a double space was helpful once, perhaps we should try it again.

```
220 PRINT
```

And what about:

```
320 PRINT
```

The program now runs much more attractively. Things are separated from one another and everything is easier to read. However, though the calculations involved are impressive in an odd sort of way and though the speed of the program is nifty to watch, there's nothing very human about the writing

we've done so far. There's no sympathy for the subject, no interaction with him. Let's play with this idea for a moment in an effort to simulate (that is, to fake) some sort of interaction. Let's talk back to him.

```
130 PRINT "HELLO ";F$
140 PRINT
```

Be sure to leave a space between the HELLO and the second quotation mark. If you made a mistake, just retype the line. That's rather friendly, don't you think? Let's try sympathy.

```
205 PRINT "SORRY ";F$
206 PRINT
```

Running the program now should show you that we've left out a skipped line. Not much of a problem, though.

```
204 PRINT
```

An easy fix.

At this point your program runs rather smoothly. It lacks a number of possible improvements, none of which we're equipped to add at this time. So we'll ignore them, for now. However, it might be a good idea for you to run the thing a few times and make notes on your problems or criticisms. Answer things incorrectly, offering letters when numbers are requested. Try some very long numbers, such as 6.87654321 hours slept per night. That causes some confusion, doesn't it?

So that you can see the whole program at once, here it is in its entirety.

```
10 PRINT CHR$(147)
100 INPUT "FIRST NAME";F$
110 INPUT "LAST NAME";L$
120 PRINT
130 PRINT "HELLO ";F$
140 PRINT
200 INPUT "YEARS OLD";Y
204 PRINT
205 PRINT "SORRY ";F$
206 PRINT
210 INPUT "MONTHS SINCE BIRTHDAY";M
220 PRINT
300 INPUT "HOURS OF SLEEP";HS
320 PRINT
400 N=Y*365.25+M*30.44
410 SL=N*HS
500 PRINT "BELIEVE IT OR NOT, YOU HAVE SLEPT"
510 PRINT SL;" HOURS IN YOUR LIFE"
520 PRINT F$;" ";L$
```


Before moving on to another programming exercise, consider the difficulty we've created between lines 204 and 206. There's no room in this area of the program for additional lines. So if we wished to insert something there, we would have to retype several lines of programming in order to make space for it.

Some forms of BASIC include a function that automatically renumbers your programs, creating (usually) ten-line-number gaps between program lines. Unhappily, Commodore BASIC does not include this facility. So you should be careful when starting a program to make a rough mental sketch of what will be in it. Try to foresee those parts that will be large and allow room for them.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

CHAPTER 5

The INPUT command is a simple and straightforward device that performs a useful function. It allows you to build a program that will run over and over again, unchanged in spite of the fact you offer it a variety of information. It makes it possible for you to add all sorts of words and numbers to the process the computer is executing. And it even gives you a chance to correct errors by prompting you to REDO work that is wrong.

But the straightforwardness of INPUT is deceptive. In fact, there is a conceptual watershed presented by INPUT commands that you have to address if you are to understand the workings of modern computer programs.

INPUT is uniquely valuable because it produces a pause in your program. It forces the computer to wait for your response. And regardless of the lethargy with which you react to an INPUT prompt, the computer and the program running on it must wait.

This pause amounts to an interruption in the program that defers to the soul who is sitting at the keyboard. What's more, the action taken at the keyboard not only restarts the program but also changes what is going on inside your computer. Of course, no harm comes to the machine and the program is not permanently changed by the action of an input. But a temporary alteration or a forced redirection of the computer's activity does take place.

This suggests that the computer can be made to perform a number of tasks, the particular chore selected being the one you request. Unlike the small programs we wrote earlier, those containing INPUT are controlled by both the programmer and the person using the program.

In theory, at least, several different utilities can be programmed into the computer at one time. Thereafter, the keyboarder can select the use that interests him and then supply the particular bits of information he chooses. The trick, of course, is in discovering a way to control which task we want executed. And while INPUT commands can absorb our choice, we haven't yet dealt with a programming tool that can respond to a selection and make further choices inside the computer.

Which is to say that all the programming we've done so far causes actions to be taken every time the program runs. This absolutely reliable and predictable characteristic of a program is one of its tremendous powers. But it is also a limitation. Since we know that we can cause computers to do several things in reaction to our responses to INPUT, we need to be able to cause the computer "not" to do something we don't want it to do.

To discover how this sort of control can be established over your 64, we will have to build a program that can do more than one thing. We'll have to concoct a program that performs several different actions under the control of the person sitting in front of it. Fortunately, we have a perfect model close at hand. We've already played with a small calculator that can easily be expanded. So, to begin, let's recover our adding machine.

The Adding Machine

Earlier, we worked with INPUT to add two numbers together. The program that accomplished this feat is to be found below and you should enter it into your 64 now.

```
NEW
 5 PRINT CHR$(147)
10 PRINT "TWO NUMBERS TO ADD"
20 INPUT "1ST NUMBER";A
30 INPUT "2ND NUMBER";B
40 C=A+B
50 PRINT A "+"B "="C
```

Take a moment to review this program, translating it as you do so. Then run the program, adding one number to another when prompted by the INPUT lines.

If at this point you are becoming both interested and impatient, it's likely you've made one or more errors in keyboarding the TWO NUMBERS TO ADD program. Take a few moments to investigate what you've done wrong and correct your oversights. And take a lesson from this experience. Be sure to run programs whenever you add a few lines to them so that you can check to see that your work is correct.

That said, it's now time to consider how this little adder might be expanded. The answer, obviously, is to allow the thing to subtract. Adding this capacity to the program ought to be a straightforward process. All you

have to do is translate the work already done. Such a translation is found below. Enter it into your 64's memory.

```
60 PRINT "TWO NUMBERS TO SUBTRACT"  
70 INPUT "1ST NUMBER";D  
80 INPUT "2ND NUMBER";E  
90 F=D-E  
100 PRINT D "-"E "="F
```

A cursory examination of this program fragment will tell you that it is virtually identical to the fragment involving addition. The only changes that have been made set up two new variables to hold the numbers to be subtracted and the switching of the plus sign to a minus sign on lines 90 and 100 so that subtraction occurs and is reported as such.

Test the two bits of program by typing RUN. When you do so the computer will once again search for the lowest line number in memory and execute it, clearing the screen. Thereafter it will proceed through the adder, requesting numbers to add. Then it will display the results of your addition. Once it has accomplished this task it will continue on to line 60 and begin subtracting things.

Obviously, this is a bit tedious in light of the fact we are already confident the adder works and only want to test the subtraction portion of the program. Luckily, BASIC allows you to run only portions of a program, avoiding those parts that you are not interested in. Type:

LIST

Then take a good look at the program in order to determine that the subtraction section starts on line 60. Then type:

RUN 60

This causes the computer to begin executing lines with line 60. Had you wished to start somewhere else, all you had to do was type the line number that should be executed first after the instruction RUN. Note that the screen is not cleared when you run the program this way. That's because the instruction to clear the screen is on line 5, a line you are circumventing.

After subtracting a couple of numbers you will once again see the READY prompt blinking before you. Under the circumstances this should indicate that the computer is prepared for you to write yet another addition to the existing program, perhaps this time making multiplication possible. Independent of the lines of programming you find below, try to add to the existing program so that two more numbers are multiplied.

Take some satisfaction in your efforts if your amendment to the program looks like this:

```
110 PRINT "TWO NUMBERS TO MULTIPLY"  
120 INPUT "1ST NUMBER";G  
130 INPUT "2ND NUMBER";H
```

```
140 I=G*H
150 PRINT G "*"H "="I
```

As with the earlier program bits we created, test this section to see that it operates correctly. Then, on to division.

```
160 PRINT "TWO NUMBERS TO DIVIDE"
170 INPUT "1ST NUMBER";J
180 INPUT "2ND NUMBER";K
190 L=J/K
200 PRINT J "/"K "="L
```

Immediately after typing this new stuff, test it.

Once you are confident that your entire program has been typed in correctly, try running the creature from the beginning. Then when you are asked to supply two numbers at each stage of the process, deliver the following: 7 for the first number and 0 for the second.

Adding 7 and 0 results in 7. Subtracting 0 from 7 results in 7. Multiplying anything times 0 results in 0. And dividing anything by 0 results in the computer response:

```
?DIVISION BY ZERO      ERROR IN 190
```

Division by zero produces a result which is theoretically infinite. As you know, an infinite amount is quite large, too large in fact to display adequately on the 64's screen. So, the mathematical oddity known as division by zero just doesn't wash. Even the best and the brightest of us aren't quite sure how much "nothing" is contained in the integer 7.

You should note, though, that BASIC tells you precisely where the impossible is being attempted: in line 190.

At this point, our little adder has grown into a calculator. But it lacks a certain practicality in that there are only rare occasions when you are interested in adding two numbers, subtracting two, multiplying two and then dividing two. So let's edit the thing so that the same two variables are used in each level of the program. This will short-circuit the program and speed up the delivery of the answer we're looking for.

To do so, take a look at the entire program. Type LIST. For your convenience, the entire program is shown below:

```
5 PRINT CHR$(147)
10 PRINT "TWO NUMBERS TO ADD"
20 INPUT "1ST NUMBER";A
30 INPUT "2ND NUMBER";B
40 C=A+B
50 PRINT A "+"B "="C
60 PRINT "TWO NUMBERS TO SUBTRACT"
70 INPUT "1ST NUMBER";D
80 INPUT "2ND NUMBER";E
```

```

90 F=D-E
100 PRINT D "-"E "="F
110 PRINT "TWO NUMBERS TO MULTIPLY"
120 INPUT "1ST NUMBER";G
130 INPUT "2ND NUMBER";H
140 I=G*H
150 PRINT G "*"H "="I
160 PRINT "TWO NUMBERS TO DIVIDE"
170 INPUT "1ST NUMBER";J
180 INPUT "2ND NUMBER";K
190 L=J/K
200 PRINT J "/"K "="L

```

Because we wish to work with only two numbers in all four of the mathematical processes at hand, it stands to reason that we need only two variables, A and B. We should have to enter them only once and the program should take it from there. Thus, all of the INPUT statements after line 30 are superfluous. As you scan the listed program, you'll see those lines that must be deleted. They are:

```

70 INPUT "1ST NUMBER";D
80 INPUT "2ND NUMBER";E
120 INPUT "1ST NUMBER";G
130 INPUT "2ND NUMBER";H
170 INPUT "1ST NUMBER";J
180 INPUT "2ND NUMBER";K

```

As you know, there is a simple way to delete these lines. Just type the line number in question and then hit RETURN. Do so now, deleting lines 70, 80, 120, 130, 170 and 180.

Now list the program and check to see that the lines have disappeared. Then run it.

At the outset everything looks great, except that tiresome TWO NUMBERS TO ADD is still in the program. Nevertheless, input 4 as your first response and 8 as your second.

What happens next isn't at all what we'd hoped for. Quick as a flash the following should appear:

```

4 + 8 = 12
TWO NUMBERS TO SUBTRACT
0 - 0 = 0
TWO NUMBERS TO MULTIPLY
0 * 0 = 0
TWO NUMBERS TO DIVIDE
?DIVISION BY ZERO   ERROR IN 190

```

Clearly, we haven't perfected this program. Type LIST to analyze what went wrong.

The errors lie in lines 90, 140 and 190. We've eliminated the variables to which these lines refer. So the computer is dredging their value from memory only to discover that it is zero. Thereafter, it is using this value, zero, in the mathematical operations we want performed. Since we're still interested in using the values in A and B throughout the program, we should use these variables on the lines in question. The correction appears below.

```
90 F=A-B
140 I=A*B
190 L=A/B
```

Now list the program to check your work. Then run it. Input 4 and 8.

The result this time appears quite preposterous. If you look carefully, however, you'll see that the answers are correct, it's just that the printed explanation of what's going on is delivering all those zeros. And it has probably occurred to you that our PRINT statements still contain the long-lost variables in question. List the program and peruse lines 100, 150 and 200. To correct them, eliminating any reference to unused variables, type:

```
100 PRINT A "-"B "="F
150 PRINT A "*"B "="I
200 PRINT A "/"B "="L
```

Once more, please, run the program. Use 4 as the first number and 8 as the second. Your screen should now whiz into view, revealing the following:

```
TWO NUMBERS TO ADD
1ST NUMBER? 4
2ND NUMBER? 8
4 + 8 = 12
TWO NUMBERS TO SUBTRACT
4 - 8 = -4
TWO NUMBERS TO MULTIPLY
4 * 8 = 32
TWO NUMBERS TO DIVIDE
4 / 8 = .5
```

Now everything is working as it should. However, there seems to be something of an excess of information on the screen. When we began, the lines which announced the mathematical process that was about to be executed were important. The screen display TWO NUMBERS TO MULTIPLY, for instance, told us what was going to happen to the numbers in question. Since we are working with only two numbers and all four mathematical operations are going to occur, these lines are no longer necessary. So, they should be deleted. Type:

```
10 PRINT "THE CALCULATOR"
60
110
160
```


Then run the program. When you're done playing with it, list the edited program that is in memory. It now appears as below:

```
5 PRINT CHR$(147)
10 PRINT "THE CALCULATOR"
20 INPUT "1ST NUMBER";A
30 INPUT "2ND NUMBER";B
40 C=A+B
50 PRINT A "+"B "="C
90 F=A-B
100 PRINT A "-"B "="F
140 I=A*B
150 PRINT A "*"B "="I
190 L=A/B
200 PRINT A "/"B "="L
```

Our program has been considerably shortened. It works beautifully, delivers all the answers correctly and, to be sure, tells us more than we really need to know. It is precisely what we set out to program: something that will perform multiple tasks. The question now is how we get it to do only that task we select.

IF

During the nineteenth century a mathematician working in France established a system of logic that delivered true or false as the answer to any properly posed question. His name was Boole and his thinking is now known as Boolean logic. Virtually all computer languages use some form of Boolean logic. This is the case in part because it reduces sets of circumstances to true-false, yes-no, on-off analytic results. Since computers run using huge numbers of carefully organized electrical switches, the on-off model of reasoning suits them perfectly.

The more advanced forms of Boole's algebra are impenetrable by normal minds. They are complicated, deadly to read and littered with those strange, distorted symbols that mathematicians use all the time. But the essence of his thinking is rather easy to understand.

Boole set up a pattern of reasoning that produces a conclusion or result. In a way, his logic is like a factory: you put in raw materials and a finished product comes out. The first part of his reasoning is known as an "argument." It establishes a kind of conditional circumstance, an "if" situation. One such argument is, "If you are telling the truth. . . ." Stated in a more algebraic fashion you might write, "If what you say = truth. . . ." This kind of argument has a resolution, an answer. And the answer is either yes or no.

It is important to note that Boolean arguments can have only two possible answers, true or false. Arguments with more than two potential conclusions don't function in this logic.

The second part of Boole's logic specifies the answer to be delivered, or the action to be taken, pending one of the two possible conclusions to the argument. This second section of the reasoning amounts to the "then" function, as in, "If you are telling the truth, then I will trust you." And since there is a second possibility, there has to be some form of "otherwise" function. "If you are telling the truth, then I will trust you, otherwise I won't."

Taken altogether, the argument and its two possible answers are known as a Boolean "statement." Boolean logic lets you construct all sorts of comparisons. You can write an argument that considers whether things are equal to one another, greater or less than one another and combinations thereof. Essentially, the system considers the question as you pose it and determines whether the circumstances at hand are true or false. Having made this distinction, the system then delivers the response you request if the situation is true or an alternate response if the situation is false.

Commodore BASIC includes a somewhat shortened form of Boolean logic. It has the power to do something in response to an argument. That is, if the conditions you pose are true, then 64 will respond. If they are not true, the computer's BASIC will do nothing at all.

This may seem silly, but it's not. In computer programming doing nothing at all results in the computer moving on to the next line of the program and doing whatever is there. Thus, the "otherwise" function is available to you.

Your possible confusion at this point is altogether understandable. However, the workings of an IF statement are simple to grasp once you've watched a few of them at work. So, type:

```
NEW
10 INPUT "5 OR 6";N
20 IF N=5 THEN PRINT "FIVE"
30 IF N=6 THEN PRINT "SIX"
```

Run this program and type 5 when the INPUT prompt appears on the screen. Then rerun it and type 6 when you see the question mark.

This small program is relatively easy to translate. In fact, the IF statement in BASIC is almost written in English. To begin, line 10 produces a prompt on the screen. It asks the keyboarder to input either the number 5 or 6. When either of these two numbers is typed on the keyboard, the computer takes that number and deposits it in the variable N.

Line 20 of this program poses the question, "What's in variable N?" The computer searches its memory and locates variable N. Once located it queries the value residing there. Then the IF statement takes over. The statement reads, "If the value of N is equal to 5 then print the word FIVE on the screen." Thus, when you input a 5 into N, you immediately see the word FIVE written on the screen. If you have typed anything other than a 5, line 20 does nothing at all. In any case, line 30 is then executed.

Please pause and consider this situation for a moment. Line 20 of this program runs every time you run the program. It causes something to happen on the screen if a certain situation is true. That is, if you've typed a 5. If you haven't, if you typed a 6 or a 12 or a 37 or any other number, line 20 will not print anything on the screen. The line makes a single, very specific comparison and responds. That's all it does. Nothing more.

Line 30 acts precisely the same way as line 20. The difference, of course, is that line 30 compares the value in variable N to the number 6. If it finds them to be equal, then it prints the word SIX on the screen. If the number in N isn't 6, then line 30 does nothing at all.

Run this program and respond to the prompt by typing the number 4. As you probably predicted, the computer responds to you by printing absolutely nothing on the screen. The entire program has run but lines 20 and 30 resulted in no action.

Among the more interesting aspects of computing in general and the IF statement in particular is the ability to determine the truth of many different sorts of statements. In the example above, we forced the computer to compare the value you entered at the keyboard with a known value, 6. But computers can compare words or strings of any kind just as accurately as they can compare numbers. To demonstrate, let's reverse the program we wrote earlier. Type:

```
10 INPUT "FIVE OR SIX";N$
20 IF N$="FIVE" THEN PRINT 5
30 IF N$="SIX" THEN PRINT 6
```

This program establishes a different kind of variable for later comparison. We've established a string variable, N\$, and then deposited a string in it, either FIVE or SIX. Run this program, responding to the prompt by typing either FIVE or SIX when prompted.

When you type FIVE, the program compares this string to the FIVE that is in the IF statement on line 20, finds N\$ equals FIVE and so prints a 5 on the screen.

The purpose of this second program is to demonstrate that the computer can compare strings as well as numbers. The IF statement is quite versatile that way. It doesn't care what you are asking to be compared, only that it have the two things available for comparison when it is called upon. Most important, any keyboard character or any ASCII or CHR\$ in the computer's memory is known to the 64. Since all these bits of information are known to the machine, it can compare them with anything you specify. For instance:

```
10 INPUT "1ST NUMBER";A
20 INPUT "+ FOR ADDITION";S$
30 INPUT "2ND NUMBER";B
40 IF S$="+" THEN C=A+B
50 PRINT C
```

Before you run this program, try to translate it. The key, by the way, lies in lines 20 and 40. Line 20 establishes a string variable, S\$, that accepts any key you hit on the computer. On line 40 this character is compared with the character that our program interprets to indicate addition. If the character typed in on line 20 is the same as the character that the program knows indicates addition, then addition takes place. Which is to say, the contents of variable A (the first number you input) and variable B (the second number you input) are added together.

The important point here involves the operation of the IF argument on line 40. The argument says, "If the string variable in S\$ is the same as the character +, then add A to B and deposit the result in C." Or, put more bluntly, if he wants addition, give it to him.

But the test for addition isn't a process of finding out if a plus sign has been input. Quite the contrary. The string variable in S\$ could be any word or character we wish to make it. The fact that we chose the plus sign is an arbitrary choice based on the way most calculators work. But we might have demanded some other string to make addition occur. For instance, make the following changes in this little adder:

```
20 INPUT "ADD FOR ADDITION";S$
40 IF S$="ADD" THEN C=A+B
```

Then run the program and type ADD when prompted. The point here is that, in the world of string variables, + is the same as ADD or any other word you might specify. Now, return the adder to its original form by typing:

```
20 INPUT "+ FOR ADDITION";S$
40 IF S$="+" THEN C=A+B
```

Run the program and respond as follows:

```
1ST NUMBER? 5
+ FOR ADDITION? +
2ND NUMBER? 3
8
```

Now run the program and type / when you see the prompt + FOR ADDITION. The computer now performs perfectly. That is, the screen should look like this:

```
1ST NUMBER? 5
+ FOR ADDITION? /
2ND NUMBER? 3
0
```

Because you typed / in response to the second prompt, the computer deposited / in S\$. When it eventually came to line 40 of the program, it compared the contents of S\$ to the character + and found that they weren't the same. Therefore it didn't add A to B. That is, the THEN portion of the expression on line 40 was not performed. Since A and B were not added

together and their sum deposited in C, C was left holding nothing. So, when line 50 rolled around and printed C, it printed a zero.

The Calculator

It's likely that you can anticipate our next improvement on the calculator we programmed earlier. The major flaw in it, if you recall, was that all four mathematical operations were performed and printed on the screen. And real calculators don't behave this way. They give only the answer you request. Modifications to the earlier calculator to make it perform more conveniently are found below:

```
5 PRINT CHR$(147)
10 PRINT "THE CALCULATOR"
20 INPUT "1ST NUMBER";A
30 INPUT "OPERATION";S$
40 INPUT "2ND NUMBER";B
50 IF S$="+" THEN C=A+B
60 IF S$="-" THEN C=A-B
70 IF S$="*" THEN C=A*B
80 IF S$="/" THEN C=A/B
90 PRINT "RESULT =";C
```

This program is only marginally more complex than the earlier calculator. It's short and to the point. But it contains a series of comparisons, each controlled by an IF statement.

A brief look at this program reveals that we establish a string variable on line 30 called S\$. The character that is input by anyone sitting at the keyboard in response to the INPUT prompt on this line will control the mathematical operations in the computer. In this particular calculator there are four possible operations: addition, subtraction, multiplication and division. Lines 20 and 40 of the program ask for, accept and label a variable for the two numerical values to be worked with.

But the real work of the program, the processing of the data that's been gathered, occurs on lines 50 through 80. Line 50 can be translated as follows: "If the symbol that's been deposited in the operations variable S\$ is a plus sign, then add the numerical variables A and B and deposit their sum in variable C." If you've typed a plus sign into S\$, this operation will duly occur. If you haven't done so, the computer will make the comparison between S\$ and + and find that they are not the same. It will then ignore that portion of line 50 which follows the THEN element of the statement, do nothing and move on to line 60.

If S\$ does not contain the character +, then nothing will happen on line 50. Thus, the computer will zip to line 60. It will then make a comparison between the contents of S\$ and the - specified in the IF argument on line 60.

And so the process continues, the computer comparing what you have asked for with what it is allowed to do on any given line. Reread the last

sentence. It is important. The argument portion of the IF statements in question decides whether you are asking for the operation that can be performed by the THEN portion of the line. In the event that the argument proves false, that you haven't asked for addition, the program ignores the rest of the line in question.

This behavior on the part of the computer is somewhat more important than it may seem. The program you see here establishes the accumulator variable we discussed earlier. It labels the accumulator "C." The function of this variable is to gather the results of any of four calculations. Obviously, we can't allow all of them to occur or the accumulator variable would become a profoundly confused, uncontrollably changing entity. That is, if C changes regardless of your intentions to change it, the variable will be out of your control.

For this reason, the program is written such that, unless the operation that has been requested matches the qualifying operation sign in the IF argument, nothing happens. Variable C is left alone.

The effect of this programming technique is to build a program that is capable of a number of actions. However, the design is such that only that action you have requested occurs. An alternative program design might cause all four operations to occur but then print only the result called for in S\$. Such a program is shown below:

```
5 PRINT CHR$(147)
10 PRINT "A DIFFERENT CALCULATOR"
20 INPUT "1ST NUMBER";A
30 INPUT "OPERATION";S$
40 INPUT "2ND NUMBER";B
50 C=A+B
60 D=A-B
70 E=A*B
80 F=A/B
90 IF S$="+" THEN PRINT "RESULT ="C
100 IF S$="-" THEN PRINT "RESULT ="D
110 IF S$="*" THEN PRINT "RESULT ="E
120 IF S$="/" THEN PRINT "RESULT ="F
```

Here, on lines 50 through 80, all four operations occur. But instead of depositing the results of each in one variable, four different variables are established. Thus, by the end of line 80 we have performed all the possible calculations on the two numbers we input. Thereafter, on lines 90 through 120, we check to see what operation is being called for in S\$. When the appropriate operation sign is located, the variable that contains the proper result is printed on the screen.

There are three disadvantages to this "different" calculator. First, and this may seem trivial to you at this point, it's slower. Since there is more work that has to be done in it, the thing takes slightly longer to do its work. You can't see the difference when you run the program, but it's there. And in

a much longer, more complicated program this sort of marginally slower performance can add up to a delay that can become downright frustrating.

Second, the program is longer. It takes more lines of programming to perform and sets up more variables. Thus, it uses more of the computer's memory. If you recall the first stuff that appears on your screen when you turn the 64 on, you'll remember that 38911 BASIC BYTES FREE is displayed on the screen. This is a description of the memory available to you when you begin programming. When it is used up you have to stop. There's no more room for additional information. And on any computer, one as small as the 64 or as large as a mainframe, memory is at a certain premium. The limitation of memory isn't a serious one these days because the cost of memory is plummeting. But still, if you don't have it you can't use it.

Finally, and most important, the "different" calculator doesn't know which variable the chosen result was deposited in. Since all four actions occurred, all four variables were set up. The four results were arrived at and are buried in the computer's memory. But only you and the screen ever know which variable matters. So if you wish to add a number to the latest result, you have to enter the result and the new number. The computer can never be made to do this for you because there is no red flag on the variable we care about. In the calculator as it now stands, this is irrelevant. The thing works only when you type in both numbers to be worked with. But soon we'll do away with this inconvenience. To do so, however, we have to accumulate a sort of running total somewhere. Hence the accumulator variable C.

As you are aware, your computer doesn't react kindly to a situation that involves division by zero. Just for fun, rerun our original calculator (the one with the accumulator, "THE CALCULATOR") and try dividing 7 by 0. You will be greeted with

?DIVISION BY ZERO ERROR IN 80

When the 64 happens onto a division by zero situation, not only does the error message appear on the screen, but your program is stopped in its tracks. It is possible to circumvent this disaster by building what is known as an "error trap" into your program. Think of this device as something akin to a mouse trap. It lies in wait for some unwanted creature to cross its path and snares the villain unmercifully. The advantage to you, or anyone else who uses your program, is that once the error is discovered and trapped, the 64 will continue to execute the remaining lines instead of crashing to a halt.

Trapping a DIVISION BY ZERO error involves an interesting problem. The error occurs only when two circumstances are working simultaneously. That is, you have to be dividing and you have to be dividing by zero. Once again, BASIC supplies us with a simple, straightforward device to handle the situation: AND.

Since division occurs on line 80 of our program, we should build our error trap some time before that event. And since the block of lines 50 through 90 makes a logical group, let's not break them up. Instead, we should add a line above this block that will stop division by zero. To do so we must either avoid

the zero or avoid the / operation. Consider this addition to the program:

```
45 IF S$="/" AND B=0 THEN S$="DIVISION BY ZERO"
```

Add this program line and run the calculator, dividing by zero. You'll immediately see that the program doesn't crash because we never get the chance to divide by zero. Line 45 now changes the division sign to the words DIVISION BY ZERO as soon as such a circumstance is created. Thus, as the computer moves through lines 50 to 90, it never discovers a "true" situation. So, nothing happens. And, therefore, the accumulator variable C is never increased from its starting point, zero.

The flaw in our remedy is that the program now delivers a bogus result. A number divided by zero isn't zero. So, we should write a programming line that will let anyone who has made this mistake be aware of his error. Such a device is known as an "error message" for obvious reasons. Here's one:

```
85 IF S$="DIVISION BY ZERO" THEN PRINT S$
```

Now when you try dividing by zero, you'll find that the error is both trapped and announced to you.

Other IFs

Thus far we've investigated arguments where things are equal to one another, as in "IF S\$="+" THEN do something." And we've looked at further qualifications of equality in the sense that two things must be equal simultaneously for an action to be taken. To accomplish this task we used AND. But there are other logical constructions that can be programmed in BASIC other than that of equality. For instance,

```
IF A>B
```

can be translated, "If A is greater than B . . ." and

```
IF A<B
```

means, "If A is less than B . . ." And you can group these sorts of comparisons together so that

```
IF A=>B
```

reads, "If A is equal to or greater than B . . ." Please note that, unlike the AND situation we dealt with earlier, the OR is unstated in the above argument. You can state the OR however:

```
IF A=B OR A=C THEN
```

translates into, "If either A equals B or A equals C then . . ."

Logical operations like those described above open a wide variety of worlds to the programmer. Not only can things be compared to see if they are identical, but they can be matched and observed in any number of ways. One application of this sort of comparative process goes on every day in magazine

offices all over the country. Magazines, as you can imagine, process thousands (sometimes millions) of names and addresses. In order to take full advantage of the post office's rate structure, magazine companies order their subscription lists in "zip code sequence." Which is to say they arrange the zip codes of their subscribers in numerical order to benefit from the discount the post office offers such mailers.

And computers do the work. They automatically examine the zip codes of those subscribers involved and reshuffle the names until they are in perfect order. To see how such a process occurs, let's write a small, but similar program.

```
NEW
10 PRINT CHR$(147)
20 PRINT "A PROGRAM THAT PUTS ANY"
30 PRINT "THREE NUMBERS IN ASCENDING ORDER"
40 INPUT "1ST NUMBER";A
50 INPUT "2ND NUMBER";B
60 INPUT "3RD NUMBER";C
70 IF A<B AND B<C THEN PRINT A;B;C
```

The left-pointing arrow, $<$, is the mathematical symbol for "is less than." On the 64 keyboard, it's obtained by hitting the shift key and the comma key.

Before typing in more lines of this program, let's review those already on the screen. Line 10, of course, automatically clears the screen and moves the cursor to the upper left-hand corner of the screen. Lines 20 and 30 can be thought of as a headline. They clue in the person at the keyboard as to what the program is out to accomplish and hint at the necessary participation on his part. In this particular case, the headline lets you know that the 64 will take three numbers and arrange them in numerical order. And by now you should be comfortable with the notion that you will be the one to let the computer know which three numbers will be considered. Lines 40 through 60 request the three numbers.

Once entered, the trio will be assigned the variables A, B and C in the order you've input them. This need not be in ascending order. You can offer numbers that wander all over the place.

Since we are dealing with only three numbers here, their potential disarray is limited. Thus, a series of tests can be established to rearrange them without a great deal of programming. The first test, on line 70, simply inquires as to whether you have offered the numbers in ascending order. In other words, it checks to see if the numbers are already ordered.

If you have entered the numbers 38, 96, 125, then the computer will assign the value of 38 to variable A, the value of 96 to variable B and the value of 125 to variable C. It will then decide that 38 is less than 96 and also that 96 is less than 125. This decision occurs on line 70. If the argument on line 70 is satisfied, that is, found to be true, the program will print the numbers: A, B, C, or 38, 96, 125.

Clearly, if the average mature adult is presented with a computer

program that promises to put a series of three numbers in order, he is unlikely to enter them into the program in order. Thus, we must test for various nonsequential arrangements. Below you'll find all those that can occur.

```
80 IF A<C AND C<B THEN PRINT A;C;B
90 IF B<C AND C<A THEN PRINT B;C;A
100 IF B<A AND A<C THEN PRINT B;A;C
110 IF C<A AND A<B THEN PRINT C;A;B
120 IF C<B AND B<A THEN PRINT C;B;A
```

If the logic of this program is lost on you, the following may help: This group of tests first determines how small A is relative to the other two numbers. These tests occur in the first part of lines 70 and 80. If A proves to be the smallest number (and this is discovered, one way or another, on either line 70 or 80), then it is printed first. If A is the smallest number, then either B is greater than C or C is greater than B. This test takes place in the second part of the arguments on lines 70 and 80. Clear? If not, don't worry. If you can decipher what's going here, then you can also figure out how 90 and 100 work. Therein B is tested for ultimate smallness. And in 110 and 120 C is so tested.

Having either satisfied yourself with the logic of the program or given up on the thing, run it. Respond as below:

```
A PROGRAM THAT PUTS ANY
THREE NUMBERS IN ASCENDING ORDER
1ST NUMBER? 45
2ND NUMBER? -43
3RD NUMBER? 29
```

The 64 should almost instantly respond by printing

```
-43 29 45
```

Then list the program. As you look through the various tests for the order of input, you will discover that the situation described on line 90 is the only one that applies to the particular numbers you input. Thus, all the other tests did nothing. Only this line printed a result.

Just for fun, run the program again and type:

```
1ST NUMBER? 4
2ND NUMBER? 7
3RD NUMBER? 4
```

Immediately, you'll see the READY prompt on the screen to indicate that the program has run. But your values have not been printed on the screen. This is because the program as written did not address the possibility of any values being equal. Thus, none of the tests is appropriate and so no result is produced.

CHAPTER 6

The IF statement as we have been using it gives the programmer very effective control over what a program does. Provided you design the thing carefully and offer yourself choices (using input), the IF statement can quickly discover what you want. It does so by testing your request against a series of standards that are established when you write the program.

However, the reacting part of the IF statement, the THEN... portion of it, is confined to a single line. And in any BASIC language there is a limit to the length of a line. So if you wish to execute a fairly complex program fragment using IF, you must do so on several lines, testing for the same thing over and over in the IF statement.

Obviously, writing a series of identical arguments in the IF portion of line after line is cumbersome. It's an awkward way of getting several lines to run based on a single choice. So the folks who brought you BASIC devised a way to control the flow of a program in a more substantial way. They concocted a method of short-circuiting the sequential order in which a program is executed. That is, a way of getting the program to stop moving from line number to line number in strict numerical order.

GOTO

The command which makes this possible is GOTO. It is precisely what it seems to be, an instruction to "go to" someplace specific. Try the following as an example:

```
NEW
10 PRINT "LINE 10"
20 PRINT "LINE 20"
30 GOTO 50
40 PRINT "LINE 40"
50 PRINT "LINE 50"
```

Now run the program. On the screen you will see:

```
LINE 10
LINE 20
LINE 50
```

The instruction on line 40 of the program was not executed because the GOTO command on line 30 forced the computer to skip it. The command instructs the computer to leap over lines, totally ignoring them, until the line number specified in the GOTO statement is reached. When the particular line called for is found, it is executed. And thereafter the computer continues on its normal road, executing line after line. Unlike the search and discover process of the READ/DATA statement, GOTO doesn't search for the line requested, execute it and then return to its original position. Instead, it goes where it is told and then picks up sequential execution from that place. Anything in between the GOTO command and its destination is never run.

GOTO can skip lines in both directions, causing some interesting results. Type this and see:

```
NEW
10 PRINT "HELLO"
20 GOTO 10
```

What you are seeing on your screen may be a bit deceptive. There is a column of HELLOs on the far left and the HELLO at the very bottom of the screen is flickering. In fact, the flickering greeting is a new HELLO being printed, over and over. You are actually watching a stream of HELLOs climb up your screen. Take a moment to scrutinize the little program that is creating so much activity. Line 20 simply tells the computer to go back to line 10 and execute it again. The 64 does so and, when it is done printing a new HELLO, moves on to line 20 which, of course, instructs the computer to go back to line 10 and execute it again. Forever, over and over, until you hit the RUN/STOP key on the lower left-hand side of your computer.

GOTO has more practical uses than producing an infinite number of HELLOs. Think back to our calculator. After you performed some calculation on it, the result was printed for you. And then the program ended. In order to perform more than one operation you have to rerun the program. But GOTO will allow you to recycle the program just before it ends, adding, subtracting, multiplying and dividing over and over.

Here's how:

NEW

```
5 PRINT CHR$(147)
10 PRINT "THE CALCULATOR"
20 INPUT "1ST NUMBER";A
30 INPUT "OPERATION";S$
40 INPUT "2ND NUMBER";B
45 IF S$="/" AND B=0 THEN S$="DIVISION BY ZERO"
50 IF S$="+" THEN C=A+B
60 IF S$="-" THEN C=A-B
70 IF S$="*" THEN C=A*B
80 IF S$="/" THEN C=A/B
85 IF S$="DIVISION BY ZERO" THEN PRINT S$
90 PRINT "RESULT =";C
```

Now add the following lines:

```
100 INPUT "TRY AGAIN Y/N";A$
110 IF A$="Y" THEN GOTO 5
120 END
```

Please note that we have introduced a new command here: END. In most modern BASICs it isn't required in a program. Commodore's BASIC included. But in some it is. Thus, to make the point that line 110 will not end the program if you have opted to continue, the command END is inserted here. That is, your Y/N choice on line 100 will direct the program to either line 110's action, GOTO 5, or line 120, END.

Run this new program, selecting the TRY AGAIN option a few times.

You are now in possession of a fairly workable calculator. You can add, subtract, multiply and divide any two numbers with it, over and over. But it still has a few problems, as you may have discovered.

For instance, run the program and add 4 and 4 together. The result, 8, appears on the screen. Now, select the TRY AGAIN mode. When asked for a 1ST NUMBER, type 4. Then a + and then, when asked for a 2ND NUMBER, enter nothing at all, just hit RETURN.

The total you are given, 8, is odd. Adding 4 and nothing should equal 4. But if you look at the program you'll see that the first addition executed, 4+4, set the values of variables A and B. Thus when you recycled the program, those two variables retained their value. They were not blanked to zero. So the "nothing" you input when you simply hit RETURN for the 2ND NUMBER, wasn't a zero. It was an instruction not to change the current value of B.

The same is true of the variable A\$. Run the program. Add a couple of numbers together and then elect to TRY AGAIN. Now add a couple of other numbers together. When asked if you want to TRY AGAIN, simply hit RETURN. Your response, not to change the value of A\$ but rather to leave it alone, has been interpreted by the computer as a Y. This occurs because that

string is being held in A\$ until you change it. So no answer at all defers to the last answer you gave.

This is one of the more interesting aspects of computer programs. They leave very little to the imagination and frown on creative responses. As you've no doubt deduced by now, programs are extremely structured. The user is supposed to respond in exactly the fashion that is preordained by the programmer. You can't answer a question such as TRY AGAIN Y/N? with a YES or a SURE, WHY NOT? and you shouldn't simply hit RETURN. When asked for an input, you have to give one. If you don't, chances are that something in memory will supply an answer for you.

A Better Trap

Now that we've learned something about GOTO statements, we can improve on our DIVISION BY ZERO error trap. This later version is a bit more elegant. Type:

```
86 IF S$="DIVISION BY ZERO" THEN 30
```

This new line redirects the program to start over at the 1ST NUMBER INPUT prompt if division by zero has been attempted. The line reads, "If division by zero has been attempted and so S\$ is now DIVISION BY ZERO (this change having occurred on line 45), then go to line 30." Note that the command GOTO does not appear on line 86. The reason for this is that it's not needed. Commodore BASIC is savvy enough to infer its use between THEN and 30. The only instance in which you must use the command GOTO explicitly is when it constitutes a line unto itself, e.g.:

```
10 PRINT "HELLO"  
20 GOTO 10
```

A Better Calculator

Before we leave the calculator for a while, let's see if we can't get it to behave more like a regular, hand-held calculator.

As was discussed before, the accumulator in our program, variable C, is potentially quite useful. In theory it can be made to keep a running total, a sum or result, that we can work with. This is the way calculators work, constantly allowing you to add, subtract, multiply or divide the result of your last calculation. But if you think about it for a moment, you'll recall that you constantly give a calculator one operation and one number. It assumes that the other number to be worked with is that total it has in hand. Even if that total is zero. Take a look at the first five lines of the calculator as it now stands:

```
5 PRINT CHR$(147)  
10 PRINT "THE CALCULATOR"  
20 INPUT "1ST NUMBER";A
```

```
30 INPUT "OPERATION";S$
40 INPUT "2ND NUMBER";B
```

The first variable in question, A, is the number that appears on the screen of a standard calculator the moment you turn it on: zero or the total you are working with. Let's put the accumulator on line 20 instead of requesting a new input:

```
20 PRINT "TOTAL =";C
```

Now, since we want to work with a running total, C, it has to be a working part of the operating section of the program. Note that this section currently exists as follows:

```
50 IF S$="+" THEN C=A+B
60 IF S$="-" THEN C=A-B
70 IF S$="*" THEN C=A*B
80 IF S$="/" THEN C=A/B
```

Because we've eliminated the variable A from our INPUT statements, we must eliminate it from the calculations. In its place we'll put the accumulator, the running total we want to work with. Edit the lines above to read:

```
50 IF S$="+" THEN C=C+B
60 IF S$="-" THEN C=C-B
70 IF S$="*" THEN C=C*B
80 IF S$="/" THEN C=C/B
```

Now run the program and make note of the problems you find in it.

The first, most obvious problem is that the program continues to print the RESULT = line below our chosen operation and number to be worked with. The running total appears on the second line of the screen, dutifully reporting that it is zero at the outset. But when we instruct the program to add 4 to this total, the result is printed below and we are asked if we wish to try again. Up to this point the screen looks odd:

```
THE CALCULATOR
TOTAL = 0
OPERATION? +
2ND NUMBER? 4
RESULT = 4
TRY AGAIN Y/N?
```

If you respond by typing Y, the screen then becomes:

```
THE CALCULATOR
TOTAL = 4
OPERATION?
```

The delay in updating the TOTAL = print occurs because we do not return to line 20 and print the revised value of C until line 110 sends us back to the

beginning of the program.

Confused? If not, you should apply to some elite group of deep and complex thinkers who specialize in convoluted and ever-changing logic constructs. However, if you feel somewhat perplexed, take heart. This is knotty stuff.

Our calculator isn't constantly informing the TOTAL = area of the screen as to the value of C. We've run across this kind of problem before. Unless you force the computer to make a change, it won't make it. Things in memory remain as they are until you make them change. And we're not forcing the variable C to update its contents until we demand that the person at the keyboard recycle the program by typing Y in response to the TRY AGAIN prompt.

To dissect the program for a moment, observe the odd activity that is going on with C. At the outset C equals zero. Then we add 4 to it. From an algebraic standpoint this is expressed on line 50 as

$$C=C+4$$

Now, from a purely logical standpoint, it is mathematically impossible for C to be equal to C+4. Still, the computer never balks at this action. Instead, it recognizes that a new value is about to be deposited in C. It then searches for that new value. Therefore, it pulls the current value of C out of its memory, adds 4 to it and then deposits the total in the variable C. Simple. Difficult. Confusing.

But, having done so, the computer has to be instructed to inform the user of his action. Somehow, we have to print the new value of C on the screen so that the mathematician at the keyboard knows that this bit of sleight of hand has taken place. And it is this bit of reporting that isn't taking place.

Let's look at the program and see why. Type LIST and you will be presented with the following:

```
5 PRINT CHR$(147)
10 PRINT "THE CALCULATOR"
20 PRINT "TOTAL =";C
30 INPUT "OPERATION";S$
40 INPUT "2ND NUMBER";B
45 IF S$="/" AND B=0 THEN S$="DIVISION BY ZERO"
50 IF S$="+" THEN C=C+B
60 IF S$="-" THEN C=C-B
70 IF S$="*" THEN C=C*B
80 IF S$="/" THEN C=C/B
85 IF S$="DIVISION BY ZERO" THEN PRINT S$
86 IF S$="DIVISION BY ZERO" THEN 30
90 PRINT "RESULT =";C
100 INPUT "TRY AGAIN Y/N";A$
110 IF A$="Y" THEN GOTO 5
120 END
```


The flaw in this program rests in the option to TRY AGAIN. This input pause stops the workings of the computer before it has updated the screen as to the new value of C. Which is to say, line 90 reports on the new value of C, but line 20 has to wait until line 100 has been satisfied.

The problem seems simple, but isn't. If we are to make the TRY AGAIN function of our current calculator automatic, that is, happen without our requesting it, then we have to build some sort of "stop" into the program or we'll never be able to halt the thing. So let's first program a "stop."

```
35 IF S$="END" THEN 120
```

This new line indicates to the computer that "If the operation sign input is END, then the program should skip immediately to line 120, which ends the functioning of the program."

Having programmed an exit of some sort, we can now make the continuation of the program automatic, provided we don't request to END. Type:

```
90 GOTO 5
```

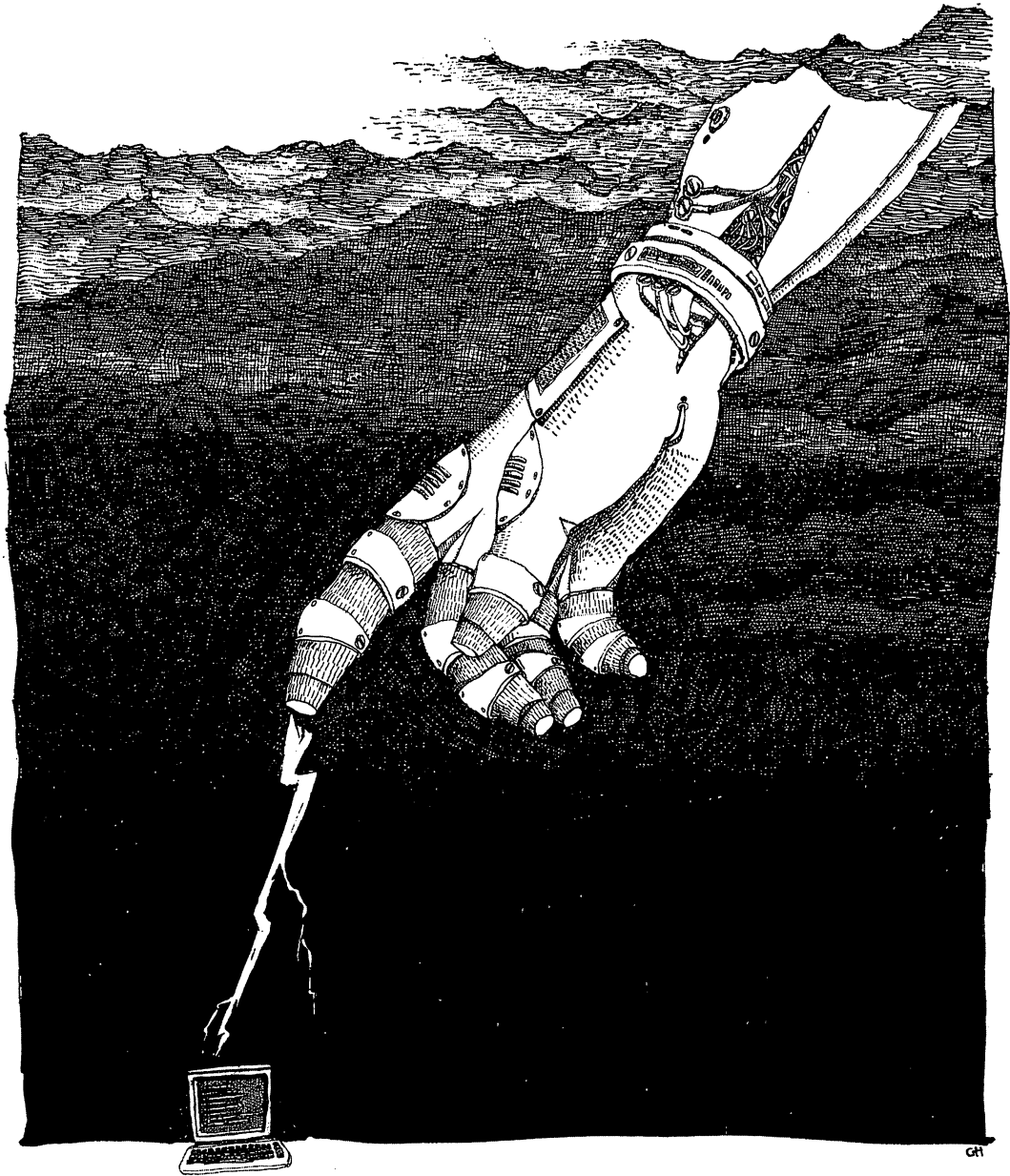
Now the program will perform its duties, update the TOTAL = line on the screen and wait for a new operation and number to be input.

You should notice that lines 100 and 110 are now irrelevant. So type the following to send them into the electronic abyss known as "erased":

```
100
```

```
110
```

Now run the program. You will quickly find that the program runs uncontrollably. To stop it, hold the RUN/STOP key down and hit the RESTORE key. Then reread this chapter until it either makes some sense to you or you are too bewildered to continue. And, by the way, take heart. You're understanding more than you are failing to grasp.



"NEW" zaps the "Old" directly into computer heaven.

CHAPTER 7

In the chapter just completed we used a simple programming technique to keep track of our calculator's running total. We established an accumulator variable and constantly deposited new results in it. The most interesting aspect of this repository proved to be its ability to add its current value with a new value. This phenomenon was accomplished in spite of the fact that a variable erases its old contents when new contents are added to it.

If you recall, we tried adding something to a variable some time ago. But unlike the bucket that first accepts one cup of water and then accepts two more, a variable disposes of the first cup before accepting the next two. Getting the variable to retain its original contents while adding more to it involved a strange-looking line of BASIC:

$$B=B+2$$

Here, the variable B is increased by two. The process that the computer goes through in performing this feat is an important one to understand. So, here is a sort of description.

The computer happens upon a line such as $B=B+2$ with a certain amount of trepidation. Were it not to do so, the process would fail. If BASIC responded to the instruction by first establishing a variable, B, and then filling it with a value, the contents of the old variable would be lost before it was needed. Thus, BASIC reacts to the situation by holding the idea of a new variable in its memory until the contents of that variable has been worked out. Only when this second action is complete is the variable established and the value of it recorded. Put differently, BASIC mutters something like this to

itself: "I've got this variable in memory, called B, that is about to be redefined. And I notice that the new definition includes the old definition. So, I'll take a memo as to the value of the 'old' B, calculate the 'new' B and then redefine the 'new' B based on my findings. Then and only then will I toss the memo."

This idea of keeping memos for future use goes on inside your 64 all the time. This is possible because the computer can establish a memory location that keeps track of any particular bit of information that may, at some point, be needed. For instance, we've already seen how the computer can print HELLO on the screen over and over.

```
NEW
10 PRINT "HELLO"
20 GOTO 10
```

If you run this little program again, remember that you can turn it off by hitting the RUN/STOP key.

Suppose, however, that you aren't interested in an infinite number of HELLOs. Imagine needing only five of them. Typing five consecutive PRINT lines would be tedious. And if you needed a thousand or so HELLOs, the process would be positively horrible. So, you have to think of a way to accumulate some sort of ongoing sum of the HELLOs that have been printed. One way to do so is shown below:

```
10 PRINT "HELLO"
20 H=H+1
30 IF H<5 THEN 10
```

Running this program will produce five tidy HELLOs on your screen in no time. This happens because the accumulator we set up, the variable H, controls the record-keeping for the program. That is, it counts how many times the program recycles.

On line 10 the program prints HELLO. On line 20 the computer establishes a variable, H, and deposits the "old" H (which is equal to zero) plus one in it. Thus H is equal to one and at this point we have a single HELLO on the screen. Then, on line 30 the program checks to see how many times we've executed the program. It inquires as to the value of H, discovers a one, compares this number with the five and finds that we haven't reached the limit allowed. Therefore, the program redirects the computer to line 10. (Note that the GOTO is implied but not necessary here.)

After line 10 has been executed five times, the variable H will have a value of five. It is then time to stop. So the program compares five with five, finds that H is not less than five and terminates the program by not recycling it.

Similar programs can be written to display the "counter" in variable H. For instance:

```
10 PRINT V
20 V=V+1
30 IF V<21 THEN 10
```

This program prints the numbers zero to twenty, one after another. Instead of printing HELLO we are simply printing the value of the counter inside the program that is keeping track of how many times the thing has recycled.

If you wish to print only the even numbers between zero and twenty you can do so easily. Simply make the counter variable increase by two each time. Change line 20:

```
20 V=V+2
```

and run the program. And to print the odd numbers you simply have to start with one, not zero, by adding this line:

```
5 V=1
```

Run this and you'll see.

FOR . . . NEXT

Among the wonderful features of the BASIC language is a simple, straightforward way of requesting a counter like the one we are programming here. It is the FOR . . . NEXT loop.

This BASIC command is a sort of merry-go-round that comes complete with a highly skilled rider. The rider catches a brass ring every time the ride goes around and puts each in a stack. When the stack reaches a certain height, the merry-go-round stops, the rider gets off and moves on to other chores. To count from one to twenty using the FOR . . . NEXT command, try this program:

```
NEW
10 FOR V=1 TO 20
20 PRINT V
30 NEXT V
```

Now for the translation. Line 10 of this program announces to the computer that it had better take notes. That is, the FOR establishes a situation in which we intend to count how many times we perform some action. The line then specifies that what we're going to count is the value of the variable V and that we're going to do so from the value one to the value twenty. The computer then scribbles a note as to the value of V.

Line 20 prints the value of V on the screen. And line 30 is a specific sort of GOTO instruction. It sends the computer back to the FOR on line 10. Here, the computer checks its memo pile and finds that the memo on top states that V has a value of one. The value is then increased by one and the memo is compared with the "ceiling" memo, which sets twenty as the highest

value of V allowable. Since V is now equal to two, the ceiling hasn't been surpassed and the computer continues.

For the sake of amusement and your curiosity, let's pause a moment and find out how long it takes your 64 to count to ten thousand.

```
10 FOR V=1 TO 10000
20 IF V=10000 THEN PRINT V
30 NEXT V
```

Run this program and keep track of how many seconds it takes for 10000 to be printed on the screen. Our running required exactly one minute and nineteen seconds.

Also for the sake of interest you should know that your computer doesn't print things on the screen as fast as you think it does. Rewrite the last program as follows:

```
10 FOR V=1 TO 10000
20 PRINT V
30 NEXT V
```

Running this program took us seven minutes and twenty-nine seconds. As you can quickly deduce, the only difference in the two programs is that the latter prints the count on the screen. Every count, all ten thousand of them. The difference in the total amount of time required to run the programs is then caused by the computer waiting for the screen to do its job.

FOR . . . NEXT loops aren't independent entities. They can be programmed into larger programs. Once the counter has completed its work, the line immediately after the NEXT function executes. For instance:

```
10 FOR V=1 TO 10
20 PRINT V
30 NEXT V
40 PRINT "ALL DONE"
50 PRINT "ON TO SOMETHING NEW"
```

Running this program indicates how the FOR . . . NEXT loop recycles ten times and then the program proceeds to lines 40 and 50.

STEP

Earlier, when we wished to print only the even numbers on the screen between zero and twenty, we added two to the variable that had been established to count the cycling of our program. The FOR . . . NEXT loop manages the same action in a more convenient way. It employs an add-on expression called STEP. Type NEW, then try this:

```
10 FOR V=0 TO 10 STEP 2
20 PRINT V
30 NEXT V
```

The STEP function indicates to the computer the amount by which it should increase the value of V. Running this program, therefore, produces a printout on the screen of all the even numbers, zero through ten.

STEP allows you to stride forward in great bounds. You could, if for some bizarre reason you so desired, count from one to one million in increments of one hundred thousand:

```
10 FOR V=0 TO 100000 STEP 100000
20 PRINT V
30 NEXT V
```

And you can mince along in tiny steps by using fractions:

```
10 FOR V=1 TO 2 STEP .1
20 PRINT V
30 NEXT V
```

Better yet, you can walk backward:

```
10 FOR V=20 TO 1 STEP -1
20 PRINT V
30 NEXT V
```

Note that BASIC assumes forward-counting procedures in increments of one. That is, if you don't specify a STEP, the computer will assume you wish to count from a lower number to a higher number, one increment at a time. But if you wish to count backward by one, you must state explicitly the STEP in question. Also, you have to arrange the FOR portion of the statement backward as well, the higher number first and the lower second.

Finally, BASIC is comfortable with negative numbers in this context:

```
10 FOR V=-5 TO 5 STEP .5
20 PRINT V
30 NEXT V
```

Here we begin with a negative five starting point and progress in increments of one-half to positive five. Check your screen to make sure you can see the counter at work.

At this point you may be wondering what practical purpose FOR . . . NEXT loops can serve. Suffice it to say that computers are ideally suited to performing remarkably boring, repetitive tasks that benefit from great speed and utter accuracy. As a result they can be enlisted to create necessary but mundane products.

For instance, we're certain that you've been turning the pages of this book at an alarming rate in your effort to become familiar with your computer. And no doubt you've paid scant attention to the page numbers, except on those rare occasions when you must stop and mark your place. But the fact of the matter is that all the page numbers and book titles you see here were typeset from an elementary computer program.

If you'll notice, all the left-hand page folios (as they are called) have the page number on the far left and the words COMMODORE 64 next to them. On the right-hand pages appear the words FOR THE INQUISITIVE ADULT followed by the page number. What's more, and this you may not have realized in spite of having read thousands of books, all the left-hand pages of a book are even-numbered and all the right-hand pages odd.

Now, there are a lot of pages in this book and it would be passably dull to type the folios for it by hand. So we wrote a little program to do the work. The secret to it lies in being able to distinguish between odd and even numbers. But there is a tried-and-true method of doing so. It's a gimmick that you'll surely find amusing.

Integers

BASIC includes a function to which you've already been introduced. It is the odd notation of a variable using the percent sign. Labeling a variable in this way lops off the number at its decimal point so that 9.25 becomes simply 9. To refresh your memory, type:

```
NEW
N%=9.99
PRINT N%
```

The result, 9, is achieved by ignoring the fraction after the decimal point. The percent sign signals this.

This same effect can be produced in BASIC by requesting the "integer" of a number. In BASIC one does so simply by typing INT. Try this:

```
PRINT INT(9.9)
```

Translated, this line means: "Print the whole number portion of 9.9." The INTEGER command, or function, is useful for a wide variety of things, one of which you'll discover in the game we will program shortly. It is also indispensable in determining odd and even numbers.

As we all know, an even number is one that is a multiple of two. That is, if you divide it by two you get an even, whole number. If the number is an odd number, its division by two yields a number and a half. As in $3/2=1.5$. This elementary fact, when coupled with the INTEGER function, separates right-hand pages from left-hand pages. Here is the program:

```
10 PRINT CHR$(147)
20 FOR F=1 TO 10
30 IF F/2=INT(F/2) THEN 60
40 PRINT "FOR THE INQUISITIVE ADULT";F
50 GOTO 70
60 PRINT F;"COMMODORE 64"
70 NEXT F
```


Line 10 of this program clears the screen and moves the cursor to the upper left-hand corner.

Line 20 starts a FOR . . . NEXT loop that, normally, would be limited by the total number of pages in the book. For the sake of this exercise we are setting only ten folios.

Line 30 decides whether we are dealing with an odd or even number. It simply says, "If the folio in question (F on this count) is divided by two and compared with the integer of the folio divided by two and if that comparison shows that the two are equal, then the folio is an even number and we should go to line 60 where even-numbered folios are printed." To demonstrate how this works, go through the program from its beginning. The first value F will have is 1, since the FOR . . . NEXT loop starts with one. On line 30, 1 is divided in half, yielding .5, and this amount is compared with the integer of .5, which is 0. Since .5 does not equal 0, the number 1 must be an odd number. Therefore, we don't go to line 60 but instead move to line 40 where odd-numbered, right-hand pages are printed.

After printing this folio we move to line 50 automatically and are therein sent to line 70, the closing part of the loop.

This sets the folio variable, F, at 2. Now 2 divided by 2 is exactly 1. And the integer of 1 is 1, so the condition set on line 30 is met and we go to line 60. There a left-hand, even-numbered folio is printed on the screen. Once this work is complete we return to line 20 for yet another value of F.

If you'd like to isolate the odd/even part of this program and watch it work, type:

```
NEW
10 FOR F=1 TO 10
20 IF F/2=INT(F/2) THEN PRINT F
30 NEXT F
```

This program decides whether a number is even or not and prints it on the screen if it is. Odd numbers are ignored.

Random Numbers

Not long ago we mentioned that we were going to program a game. The particular challenge in question involves either logic or pure chance, depending on how you play it. In other words, if you approach the thing methodically, with a plan of attack, you will always win. If you don't, you almost certainly will be defeated.

Intrigued? Well, the sport involves guessing a randomly selected number between one and one thousand. You have ten guesses. The interesting part of the game is that the 64 is going to pick the number and tell you if you have guessed it. And the machine will select a purely random number every time.

This generating of random numbers is a built-in feature of the BASIC language. It has a variety of uses but we're going to concentrate on its simple ability to manufacture numbers in a completely unpredictable way. Type:

PRINT RND(0)

Instantly an ungainly looking number appears on your screen. It is impossible for your authors to discuss the specific number you see because the 64 has selected it at random, not quite out of thin air, but unpredictably. Type the same instruction again and you'll get a new number. To get ten of them in a hurry try this:

```
NEW
10 FOR N=1 TO 10
20 PRINT RND(0)
30 NEXT N
```

One characteristic of randomly generated numbers is that they are all fractions that are less than one and greater than zero. This last remark may alarm you in light of the ten numbers that now appear on your screen. One of them may look something like this:

5.619746E -08

This particular figure doesn't seem to meet the requirement that a random number be comprised of a value between zero and one. It does, however. The E -08 is a form of scientific notation. It is used to express very long fractions that are either extremely unwieldy to look at or won't fit in the standard screen space allotted to numbers. Your 64 doesn't have the capability of printing all the digits in this number on the screen, so it has reported on the number by displaying it with an exponential value of negative eight. And since you probably don't know what that means, it's simple. Move the decimal point eight places to the left. Thus:

5.619746E -08 equals .0000005619746

This much understood, you probably now want to know how an eminently systematic machine like a computer generates a number at random. The secret lies in a type of counter inside the 64. The moment you turn the computer on, an element of it begins counting at a very rapid rate. Like some giant roulette wheel, these numbers circle endlessly. When you command the 64 to print one of them, the computer retrieves the particular number that happens to be available to it at the time. Since there are a huge number of values being produced and since the timing of your request bears no relation to the rate of the random number generation, the particular figure delivered is unpredictable.

Now type:

```
PRINT RND
```

and hit RETURN. You are informed you have made a syntactical error. The reason for this is simple. The BASIC command for a random number consists not only of RND, but also of a number that must follow in parentheses. So PRINT RND(0) yields a true random number. Instructing the 64 to PRINT

RND(2) or RND(-3) yields a different sort of number, generally but not universally unpredictable. That is, putting any number other than zero in the parentheses delivers an ersatz random number that we won't bother to explain here.

The Game

The program you are about to enter into your computer is rather long. As a result, you will almost certainly find some of it confusing at first. However, it uses only those commands we've already discussed. And after you've entered it into your computer, we'll discuss how it works. For now, just type patiently, figuring out what you can as you do so.

Note that typing line 25 of this program produces the following on the screen:

```
25 PRINT "THE COMMODORE 64 HAS SELECTED A NUMBER"
```

This has occurred because the line contains more than forty characters and the 64's screen is limited to forty characters per line. The machine's BASIC is not so limited, though. Thus, in spite of its ungainly appearance, the line is still one programming line to the computer. It just takes two lines on the screen to display. Please note that you must not hit RETURN after typing "SELECTED" on this line. Let the computer skip to the next line on the screen. Hit RETURN when the entire programming line is complete.

NEW

```
5 PRINT CHR$(147)
10 X=INT(1000*RND(0)+1)
15 INPUT "YOUR NAME";N$
20 PRINT "RULES:"
25 PRINT "THE COMMODORE 64 HAS SELECTED A NUMBER"
30 PRINT "BETWEEN ONE AND ONE THOUSAND. YOUR"
35 PRINT "JOB IS TO GUESS WHAT THE NUMBER IS."
40 PRINT "YOU WILL BE GIVEN TEN GUESSES. AND"
45 PRINT "THE COMPUTER OFFERS CLUES AS YOU DO."
50 PRINT "GOOD LUCK."
55 FOR G=1 TO 10
60 PRINT "GUESS NUMBER";G
65 INPUT N
70 IF N<X THEN READ R$:PRINT "TOO LOW ";R$
75 IF N>X THEN READ R$:PRINT "TOO HIGH ";R$
80 IF N=X THEN PRINT "BINGO ";N$:GOTO 110
85 NEXT G
90 PRINT "SORRY, THE NUMBER WAS";X
95 DATA TRY HARDER, YOU CAN DO IT, COME ON, THINK, YIPE
100 DATA DOOMED, KEEP TRYING, OOPS, WRONG, NOPE
110 END
```

No doubt this program strikes you as complicated and somewhat bewildering. In fact, it is fairly simple once you break it into parts and analyze them. So, we'll approach the program in that fashion.

Line 10 of the game creates the number that you are trying to guess. It does so in an interesting way, generating a number between one and one thousand that is an integer (has no decimal places). The process is as follows: The computer establishes a variable to contain the number once it is generated and labels that variable X. The line then describes the number. First, it is to be an integer, so the command INT() surrounds the remainder of the line. Inside the parentheses you find three elements. The central one is RND(0), which generates a random number. This number, as you know, will be between zero and one. So, since we want a number between one and one thousand, we must multiply the random number that is generated by 1000. For instance, assume the computer delivers the number .747331281. Multiplying this number by 1000 turns it into 747.331281. Turning this number into an integer makes it 747. Which is fine, a perfectly acceptable number.

However, it's conceivable that the random number 6.42133178E -04 will be chosen. And that number has a value of .000642133178, which, when multiplied by 1000 becomes .642133178. Unfortunately, the integer of this figure is zero. And for the purposes of this game we need numbers between one and one thousand. Thus, inside the integer-creating parentheses we add one to the number created, turning it into 1.642133178, which has an integer value of one.

Another possibility is that the computer will select the number .999743462. Once that's multiplied by 1000, the result will be 999.743462. Adding one to this figure results in 1000.743462, which has an integer value of 1000. Another acceptable number.

Line 15 of the program gathers your name into the computer's memory and stores it in N\$ so that we can be personal about our congratulations should they be necessary.

Lines 20 through 50 simply explain what the game is about.

Line 55 establishes a FOR . . . NEXT loop that will count the number of guesses the player makes, hence the limit of ten on this line. The loop is closed, by the way, on line 85.

This program calls for the player to make ten guesses in an effort to discover the random number that the 64 has in its memory. Thus, on line 60 we print the words GUESS NUMBER and then the number that is stored in the variable G (for guess) that is serving as a counter. Since this line of code operates inside the FOR . . . NEXT loop, it will execute ten times, once every pass through the loop. On line 65 we ask for, accept and store the current guess. Since it will be a number, the variable N is used.

At this point in the program all the information necessary for its operation has been produced. The computer has created a number and the player has been asked to guess what it is. The guess has been received. Now we must compare the two. Lines 70 through 80 perform this task. But each is

in a form that isn't completely familiar to you at this point, so we'll investigate them slowly.

Line 70 compares the guessed number (N) with the computer's random number (X). Note that X is generated outside the FOR . . . NEXT loop and so will be created only once. After all, it would be unfair to change the number after every guess. At any rate, the two are compared to determine if the guess is greater than the number being sought. If it is, then this line is instructed to READ from DATA. But it does so in a way we haven't discussed.

To illustrate what's going on here we're going to write a small program. However, we are not going to type NEW before doing so or write over those lines already in memory. That would ruin a considerable amount of work. So pay attention to the line numbers here.

```
1 FOR I=1 TO 10
2 READ Z
3 PRINT Z
4 DATA 1,2,3,4,5,6,7,8,9,10
5 NEXT I
6 END
```

Before running this program, take a look at it. Clearly, we're going to do something ten times. The FOR . . . NEXT loop that is begun on line 1 establishes this fact. It sets a counting index (hence the "I" most often used in this sort of program) that will operate from 1 to 10.

That which we are about to do ten times is READ from a DATA line and then label the data found there Z. Once read, line 3 prints the data on the screen. And, in DATA we've stored ten bits of information. Line 5 returns the program to line 1 for yet another pass. This described, type:

RUN

On the screen you immediately find the numbers 1 through 10 in a nice, neat column. The only mystery behind this particular performance is the incongruity between the READ and DATA statements. There is only one READ variable and there are ten bits of DATA. Previously, we established a corresponding number of variables in the READ command and the DATA statement. As in (don't type this into your computer):

```
10 READ A,B,C
20 DATA 1,2,3
```

But this time we've limited the READ command to a single variable and included ten bits of data on the DATA line. There seems to be a shortfall in the variable department here. But this doesn't prove to be a problem when you run the program. This is because the computer once again takes notes. When a single variable is established in a READ command and there is more than one piece of information in a DATA statement, then BASIC makes note of those bits of data that it has read previously. As it rereads the data, it

overlooks those bits that it has already used, progressing through the offered information one bit at a time. Thus, a single variable can read several bits of data on each pass of the FOR . . . NEXT loop.

Now, get rid of this little program. Type:

```
1  
2  
3  
4  
5  
6
```

Think of the READ variable as a pencil. And think of the DATA statement as a grocery list. With each execution of the READ command, the variable picks up an item on the grocery list. At the same time the item is checked off, eliminating it from those things yet to be acquired. The next time the pencil goes to work it searches for a bit of data on the list that hasn't been checked off. Finding such a piece of information, it accepts it and then checks off the data, indicating that it has been "used up." And so on until all the data is exhausted.

This is precisely what is happening in our game program. On lines 70 and 75 the program compares the value of your guess, N, with the number created by the computer. If your guess is either greater than or less than the number, then the computer reads data from either line 95 or 100. On these two lines we've written ten bits of encouragement and comment. With each of the ten cycles of the FOR . . . NEXT loop, either line 70 or 75 reads these comments, consecutively, into the variable R\$.

Note that BASIC is indifferent to the multiple use of variable R\$. It takes notes, that is, checks off its grocery list, on the DATA lines. Thus, this variable progresses through the data in an orderly way regardless of whether line 70 or line 75 is doing the work. Note also that we have two lines of data here, lines 95 and 100. We could have written all this information on a single line if we had wished. Or, we could have put one comment on each of ten consecutive data lines. BASIC doesn't care. It regards a block of data as just that, a single chunk of information to be read.

To refresh your memory, line 70 of the program reads:

```
70 IF N<X THEN READ R$:PRINT "TOO LOW ";R$
```

So far we've considered how the guessed number is compared with the computer's number and how this line then reads from DATA. The next element of the line is a colon, not a semicolon. It is used to separate two commands that are used on one line of programming. The semicolon also used on this line, just before the printing of R\$, is a punctuation mark used to separate different kinds of elements in a PRINT statement (or INPUT statement if the input is doing some printing on the screen). Make an effort to remember this simple but crucial distinction. Semicolons and colons do very different things and can never be interchanged without generating a SYNTAX

ERROR message.

Lines 70 through 80 compare the guessed number with the computer's number. Logically, your guess must be either less than the computer's number (line 70), greater than the computer's number (line 75) or exactly correct (line 80). In the case of wrong answers the message TOO HIGH and a comment or TOO LOW and a comment are printed on the screen. Then the computer reaches line 85 and the cycle begins again. In the case of a correct answer we must end the game. Thus, line 80 not only prints BINGO and your name, but then sends the computer to line 110, which ends the program.

In the event you make ten guesses, all of which are wrong, the counter in the FOR . . . NEXT loop exits the program from the loop on line 85. Thus, line 90 is performed at the end of the game if, and only if, you fail.

Please note that this program contains a conditional loop. That is, the FOR . . . NEXT loop is set up to perform all the actions inside it ten times. However, there is an IF . . . THEN test within the loop that short-circuits the cycling of the program, sending its operation out of the loop. Which is to say, line 80 keeps line 85 from executing under certain circumstances.

Undoubtedly, you've now played with the game a while. And, it's equally likely that you have been frustrated in your efforts to defeat the computer. There are, after all, a thousand possible numbers and you've only ten guesses. But there is a method of playing this game that will allow you to win with some regularity. The secret is to behave like a computer, searching for the number in question in a manner that eliminates the greatest number of possible choices with each guess. To do this, you must reduce your potentially correct answers by one half on each move. That is, aim for the middle every time. Begin by guessing 500 at the outset, since this is the exact center of your choices. If, for instance, this number is too high, you then know that the correct answer is between 1 and 499. So halve the field again by guessing 250. And so on. Try it. You'll be surprised how often you discover the answer before you run out of guesses.

Time

Since we've completed our review of a fairly complicated program, having dined on some meaty stuff, perhaps an after-dinner mint is in order. Type:

```
PRINT TI$
```

On the screen you will see a number like 002654. This is the time your computer has been running since you last turned it on. Of course, your number will probably be different. And it is always changing. The above example indicates that our machine has been running for zero hours (00), twenty-six minutes (26) and fifty-four seconds (54).

If you'd like to know precisely what time it is when you're working at your computer, you may easily set the 64's clock to the correct time. However, you have to remember that the computer's timepiece is a

twenty-four-hour clock, so 2:00 p.m. is the fourteenth hour. So 2:30 p.m. is 143000, or fourteen hours, thirty minutes and zero seconds. To set the clock to two-thirty, you type:

```
TI$="143000"
```

Since TI\$ is a string variable, you must surround your time with quotes. After setting the clock, type:

```
PRINT TI$
```

for an up-to-the-second report on the time.

Now, it's interesting that the 64 has a clock. But since it turns off whenever you turn off your computer, it doesn't appear to be terribly useful. But it is.

Assume for the moment you'd like to know how long, exactly, something takes to execute on your computer. You can use the clock to time what happens and compare the duration of various operations. Let's time the multiplication of two numbers. Since this will require mere nanoseconds, let's time a thousand such multiplications so that we can develop a meaningful period.

```
NEW
10 PRINT TI$
20 FOR I=1 TO 1000
30 A=6
40 B=7
50 C=A*B
60 NEXT I
70 PRINT TI$
```

Running this program will cause a number, the TI\$ on line 10, to appear on the screen immediately. Then, after a few seconds, another number, the TI\$ on line 70, will appear. Our running resulted in:

```
143220
143228
```

Thus, eight seconds passed as the computer executed the FOR . . . NEXT loop one thousand times and multiplied six times seven as it did so. Now type:

```
30 A=23456789
40 B=98765432
```

Then run the program again. As you can see, the computer takes longer to perform very large multiplications than smaller ones.

CHAPTER 8

You have probably deduced that a FOR . . . NEXT loop reiterates everything enclosed within it as many times as the index or counter requires. That is,

```
10 FOR I=1 TO 10
20 PRINT "HELLO"
30 NEXT I
```

produces ten HELLOs on your screen. And

```
5 PRINT "GOODBYE"
10 FOR I=1 TO 10
20 PRINT "HELLO"
30 NEXT I
```

will print one GOODBYE and ten HELLOs because the PRINT statement on line 5 is not held inside the loop.

Think of loops as parenthetical remarks. The opening and closing brackets set limits to these remarks. They contain and control them. What's more, as in more complex levels of algebra, you can use more than one pair of parentheses at a time. For instance, one might write the following formula:

$$X=2(xy^2(B-1))$$

in order to assure that the process of subtracting one from B occurs prior to and independent of any other calculation. Similarly, you can establish loops within loops. Doing so creates what is known as a "nested loop."

Nested Loops

Nested loops reside, one inside another, like a child's nesting building blocks. But they don't behave exactly like the parentheses of a formula. Observe:

```
NEW
10 PRINT CHR$(147)
20 FOR A=1 TO 12
30 FOR B=1 TO 12
40 C=A*B
50 PRINT A"*"B"="C
60 NEXT B
70 NEXT A
```

Note that the first FOR statement refers to variable A and that the last NEXT statement refers to variable A. Inside this loop we've nested another one, the FOR and NEXT loop that refers to variable B.

Run the program and you will be presented with a simple multiplication table, the sort you were told to memorize in the second grade. This table is produced in a simple way. On line 20 the variable A is assigned a value of 1. On line 30 variable B is given the value of one. On line 40 the two variables are multiplied and their result deposited in variable C. Then, as before, we print an explanation of this process on the screen and

```
1 * 1 = 1
```

appears there. But the computer finds itself in an odd position when line 60 is encountered. There, the program recycles to line 30, changing the value of B to 2. Thus, line 20 has been avoided and the variable A remains 1. With A equal to 1 and B equal to 2, the screen then displays:

```
1 * 2 = 2
```

This process continues, the loop controlled by variable B recycling twelve times until the screen contains:

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
1 * 7 = 7
1 * 8 = 8
1 * 9 = 9
1 * 10 = 10
1 * 11 = 11
1 * 12 = 12
```



Dreaded Loop
LOOPIUS OMINOUS
Looping Corp. 1. Nested Loop 2.

Beware the dreaded Loop and its even more threatening cousin, the Nested Loop.

At this point the loop controlled by B exits on line 60 since all twelve reiterations of the process are complete. So the computer moves on to line 70, where it is instructed to proceed to the next value of the counter A. Thus, A is increased from a value of 1 to a value of 2. This accomplished, the program moves to line 30, where it once again encounters the FOR . . . NEXT loop involving variable B.

This inner loop, of course, has already been executed once. So the computer resets the value of B at 1 and begins anew. This produces:

```
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
```

In this ever-repetitive way the nested loops in this program produce a multiplication table that goes from $1 * 1 = 1$ through all the possibilities to $12 * 12 = 144$.

Rerun the program, slowing down the print that appears on the screen by pressing the CTRL key in the left-hand corner of your keyboard.

It is very important to be sure you nest loops properly. Unless each fits neatly and completely inside another, the computer will rebel. For instance, type:

```
60 NEXT A
70 NEXT B
```

and then run the program.

```
1 * 1 = 1
2 * 1 = 2
3 * 1 = 3
4 * 1 = 4
5 * 1 = 5
6 * 1 = 6
7 * 1 = 7
8 * 1 = 8
9 * 1 = 9
10 * 1 = 10
11 * 1 = 11
12 * 1 = 12
? NEXT WITHOUT FOR      ERROR IN 70
```

appears on the screen. List the program and take a look at it.

As you can see, the program now executes all twelve of the values of A, holding B constant as it does so. This is because the first NEXT statement it encounters, on line 60, sends the computer back to line 20. When this loop is completed and the computer moves on to line 70, it finds another NEXT that refers to another variable, B. However, this variable's FOR line is inside the

other loop. It is therefore unavailable. It is as though each loop possesses an impenetrable shell that can be breached only by the loop's own counter. The NEXT statement of one loop can't penetrate the perimeter of another loop. Thus, line 70 contains an instruction that the computer can't fulfill. It can't get to variable B.

So, as far as BASIC is concerned, the NEXT on line 60 exists entirely alone. There is no matching FOR command.

Now try:

```
60 NEXT Q
70
```

The new line 60 establishes a NEXT statement for a variable that doesn't exist in the program. Typing 70 simply deletes that line from the program. Run this variation.

On your screen you should see the following:

```
1 * 1 = 1
? NEXT WITHOUT FOR      ERROR IN 60
```

Since there is no counterpart to variable Q, your computer rejected this new line out-of-hand.

It is interesting to note that your computer will assume you know what you are doing when you nest loops. That is, it will proceed according to its own rules unless you instruct it to do otherwise. Type:

```
60 NEXT
```

Running the program in this configuration produces a screen that indicates we've run through one loop. The only question is, which one? If you look at the screen and list the program, you'll see that the innermost loop has been executed. When VIC reached line 60 and found a NEXT that directed it nowhere in particular, the computer then cycled to the nearest possible FOR statement. Thus, the loop described by variable B was performed twelve times. Now type:

```
70 NEXT
```

and run the program.

As you can see, everything works perfectly. BASIC takes note of the fact one NEXT is employed in the B loop and one is used to close the A loop. Its busy note-taker has scribbled a memo and kept perfect track of the two loops involved, even though you didn't indicate which was which.

Nested loops can be made to perform a variety of tasks. And, of course, you can program control of those chores into your own hands. Enter the following program:

```

NEW
5 PRINT CHR$(147)
10 INPUT "START";A
20 INPUT "STOP";B
30 FOR X=A TO B
40 FOR Y=A TO B
50 C=X*Y
60 PRINT X"*"Y"="C
70 NEXT Y
80 NEXT X

```

This program is quite similar to the multiplication table we just looked at, with one exception. You are now in control of the range of numbers to be multiplied one with the other. Run the program and plug in these numbers:

```

START? 156
STOP? 170

```

Your screen will scroll the table before you, beginning with $156 * 156 = 24336$ and finishing with $170 * 170 = 28900$.

The overall design of this program should be understandable to you. Instead of demanding that the computer produce multiplication tables for values 1 to 12, we've left the starting and stopping values accessible through the use of INPUT statements. Read through the program until you feel comfortable with the way lines 30 and 40 establish loops that begin and end (that is count) for a period that is set with the INPUT statements.

If you've run this program a few times, you may have uncovered several problems with it. For instance, enter the answers as below:

```

START? 170
STOP? 156

```

On your screen will appear

```

170 * 170 = 28900

```

and nothing more. As we discussed earlier, the FOR statement sets up the beginning and ending points of a variable that "counts." BASIC assumes that you count from a smaller to a larger number unless you specify counting backward by using STEP - 1 or some other negative increment. So when you input a value for A greater than for B in the program, it is stopped in its tracks after multiplying the first two values in the two nested loops. No instruction was written for the 64 to count backward if the numbers required it.

But take heart. There are some simple revisions to the existing program that will solve the problem. The first of these solutions involves an IF . . . THEN statement. The logic of this amendment to the program holds that if the variable A is greater than the variable B, then their values ought to be exchanged. The most appropriate time to make this switch is immediately after the INPUT lines. Try:

```
25 IF A>B THEN A=B:B=A
```

Note that we've used the colon to separate two individual commands on the same line of programming. Now run the program and again answer:

```
START? 170  
STOP? 156
```

Your computer will then print

```
156 * 156 = 24336
```

and nothing more. Clearly, something's wrong. The machine seems to have started to work as we had hoped but is now unable to continue. Consider the IF statement for a moment. Since our original input established A as 170 and B as 156, the conditions set in the IF statement were fulfilled, A was greater than B. So, the 64 then took the value in B and deposited it in A. Once that was completed, A equaled 156. And since we'd done nothing to B, B also equaled 156. So when the second command on line 25 was executed, B=A, the value of A, now 156, was redeposited in B. Thus, all the values in the program equaled 156. Confused?

Think of this as a shell game. Values are being moved from one variable to another. But, as you know, depositing a value in a variable erases the value that was previously there. Hence, the disappearance of the value in A, 170.

To solve this dilemma we can rely on the technique used in our calculator. Establish an alternative variable to hold numbers temporarily while we move them around. Build a little accumulator variable that you can put a number in so it won't be lost. Try this:

```
25 IF A>B THEN Z=A:A=B:B=Z
```

This line should work better. Translated, it reads, "If the value of A is greater than the value of B, then let variable Z hold the value in A temporarily. Then change the value in A to the value in B. And, finally, take the value in Z and put it in B." The switch is made. Rerun the program to check that everything works as it should, regardless of the order in which you input numbers.

As a matter of interest, you might have solved the problem in another way. Consider this:

```
8 D=1  
30 FOR X=A TO B STEP D  
40 FOR Y=A TO B STEP D
```

List the program and glance over the lines you've just added to be sure your revisions are included.

So far, the change you've made is superfluous. BASIC will assume that you STEP by increments of one if you don't specify an alternate increment. By appointing a value of 1 to D, you've simply confirmed this inclination. So, in essence, we're taking time to tell the computer something that it already knows to do. However, we can take this variable, D, and work with it. Type:

25 IF A>B THEN D=-1

This change in line 25 is simple to translate. It states, "If the value of variable A is greater than the value of variable B, then variable D should be negative 1." This changes the STEP in lines 30 and 40 so that, if you introduce numbers backward, BASIC knows to count backward as a result. Of course, your multiplication table will be backward too. Try it and see.

CHAPTER 9

Thus far you have been introduced to a variety of BASIC commands that allow you to control the operation of a program using different sets of data or inputs. This has been accomplished by using the INPUT command to vary the information the computer is working with, the IF . . . THEN statement to establish conditional patterns in the program and GOTO to redirect the program's operations.

Needless to say, the number of programs that can be created using only these commands is enormous. Their logical power, inherent speed and flexibility are remarkable. But it will come as no surprise to you to learn that there are many other BASIC programming tools. And one of them proves invaluable when designing programs that repeat operations.

GOSUB

Often, the computer is required to perform an operation several times and in different areas of the program. Processes such as rounding off numbers to the nearest cent or checking the nature of an input to see that it is appropriate can be needed many, many times in a single program. And writing these processes into every location that might require them is time-consuming and tedious. Thus, BASIC allows you to create a programming fragment that behaves like a miniature program within a program. This module is known as a "subroutine" and performs the same action over and over, whenever it is asked to do so.

A subroutine is created using the BASIC command GOSUB, which is a

modified form of the GOTO command. GOSUB sends the computer to an isolated part of the program where something is done. When the action has been completed, the subroutine "returns" the computer to the line of programming that immediately follows the GOSUB command. Type:

```
NEW
10 PRINT "A PAUSE"
20 GOSUB 100
30 PRINT "IS EASY"
40 GOSUB 100
50 PRINT "TO PROGRAM"
60 GOSUB 100
70 PRINT "USING GOSUB"
80 END
100 FOR I=1 TO 1000
110 NEXT I
120 RETURN
```

You should notice several somewhat different elements to this program. First of all, let's consider the final lines of the program, 100 through 120. In this area we have written a subroutine that, essentially, accomplishes nothing other than count. Line 100 sets up a FOR . . . NEXT loop that counts variable I from one to one thousand. Line 110 closes the loop. Since we know that it takes the computer time to perform any action, requesting a thousand of them should slow the machine down somewhat. And, of course, this is exactly what happens. But the trick to the subroutine in question is that it is employed three times in this program, in spite of the fact it is written only once.

Line 10 of the program prints A PAUSE on the screen. Line 20 contains the command to GOSUB, or "go to the subroutine," and specifies the line on which that subroutine begins: line 100. Thus, the computer is directed to execute line 100. But before doing so, it takes yet another memo to itself, noting that it has been sent to the subroutine from line 20. This annotation having been made in the computer's memory, lines 100 and 110 are executed. As you know, these two lines form a loop and so the computer cycles between them a thousand times. This produces a slight pause while the computer seems to be doing nothing at all. Still, after a few seconds, all one thousand counts of the variable I have been made and line 120 is executed.

RETURN is the closing command in a subroutine. It instructs the computer to refer to its GOSUB memo and return to the line that immediately follows the last executed GOSUB command. That is, the computer remembers which of many GOSUB commands was last executed and returns to this spot.

This is very much like a loop. And a bit like GOTO. But there is a crucial difference. A loop, even when nested in another loop, is a single entity. A variable is set up and it is counted from its beginning value to its final value. While the NEXT portion of the programming device recalls the exact location

of the FOR portion of the loop, a loop can work only with one FOR and a matching NEXT line of programming. Similarly, the GOTO command is perfectly capable of interrupting the flow of a program, sending the computer to another area in order to execute something. But in order for the 64 to return to its original line of code, another GOTO is required. And this one must have a specific line number associated with it. (There is an exception to this situation which we'll get to shortly.)

The power of a subroutine lies in the computer's ability to recall which GOSUB it last performed. The result of this recollection, in our pausing program, is for the computer to delay for a moment, then get to line 120. Here the computer is sent to line 30 because this is the line which immediately follows the last GOSUB encountered.

On line 30 IS EASY appears on the screen. And then, on line 40, the computer is once again sent to line 100. And, once again, the computer notes that it is being dispatched from line 40 and so knows to return to line 50 when a RETURN command is encountered.

The pause subroutine is used three times in this program. And you should note, it is at the very end of the program. But not at its END. Which is to say, the subroutine occupies the highest line numbers in the program. But, in the case of this program, we END on line 80. This instruction forces the program to stop.

In the past, we've dispensed with this superfluous command. We've simply let the computer run out of lines to execute in order to end the program. But here we must use the command. Type:

80

and hit RETURN. You've just eliminated the END line of code from this program. Now run it.

?RETURN WITHOUT GOSUB ERROR IN 120

appears on your screen. This is the result of a programming error created by line 120 of the program. Because the computer now executes lines 100 through 120 as its final action, it comes upon the RETURN command in line 120 without having been sent to it by a GOSUB command. That is, when line 120 is last encountered, on its fourth execution, there is no memo in the computer's memory informing the RETURN as to the particular GOSUB that should be paired with it. This happens, of course, because no GOSUB was employed this time around. So the computer points out that you have a RETURN in your program without a matching GOSUB. To avoid this you must END the program before the final execution of line 120.

Now type:

120

and hit RETURN. You have just removed the RETURN command from the program. Run it and

A PAUSE

appears on your screen. And nothing more. The computer has executed line 10 and line 20 has then sent it to line 100. There the computer counted to one thousand and, lacking any instruction to RETURN, stopped.

One note: pauses are sometimes programmed into a computer so that the machine will give the impression of considering what it should do next. This illusion is often built into games, such as chess or checker games played with the computer. The purpose? To avoid intimidating and annoying the human player who is not accustomed to nanosecond-long response times.

ON . . . GOTO and ON . . . GOSUB

Earlier, the point was made that the computer can perform a sort of loop using two GOTO commands. An example follows:

```
NEW
10 PRINT "ONE PAUSE"
20 GOTO 50
30 PRINT "FROM GOTO"
40 END
50 FOR I=1 TO 1000
60 NEXT I
70 GOTO 30
```

This program prints two lines on the screen, pausing between the appearances to count to one thousand. The second GOTO, the one on line 70, acts as a RETURN of sorts here. But since there is only one line number following the GOTO command, it can be used to recycle the computer only to a specified program location. But both GOTO and GOSUB have a useful accessory command known as ON.

Like a quarterback hunched over the center to take the snap, ON counts, "ON ONE, ON TWO, ON THREE . . ." Try this program:

```
NEW
10 INPUT "A NUMBER";A
20 PRINT "A =";A
30 ON A GOTO 100,200,300,400
100 PRINT "LINE 100":END
200 PRINT "LINE 200":END
300 PRINT "LINE 300":END
400 PRINT "LINE 400":END
```

Run this program and respond to the INPUT prompt by typing:

```
A NUMBER? 3
```

The computer will then print:

```
A = 3
```

LINE 300

This occurred because line 30 directed the computer to GOTO line 300. It did so by analyzing the contents of variable A and using the value of A to select the particular GOTO line called for on line 30. That is, because variable A was 3, the GOTO command on line 30 picked the third line specified for execution.

The variable A in this case is known as the "index variable" in an ON . . . GOTO or ON . . . GOSUB line of programming. The index number has to be a whole number greater than zero and has to be an integer. No decimal places allowed. And, of course, the number must be lower than the total number of lines specified after the GOTO command. The index number selects the line to "go to" by counting those in the list. It's as simple as that. But the construction can become quite complex if the index number is the result of some calculation in the program. One simple example follows:

```
NEW
10 FOR I=1 TO 4
20 ON I GOTO 100,200,300,400
30 NEXT I
100 PRINT "AN INDEX":GOTO 30
200 PRINT "CAN COUNT":GOTO 30
300 PRINT "BUT IT'S":GOTO 30
400 PRINT "CONFUSING TO FOLLOW"
```

Here, the counter set up on line 10 counts the value of variable I from 1 to 4. When the value of I is 1, on the first pass of the cycle, the index, I, on line 20 picks line 100 to GOTO. When line 100 has printed AN INDEX on the screen, it then is sent to line 30 and thus back to line 10 where the value of I is increased to 2. This causes line 20 to send the computer to line 200 because line 200 is the second line in the list. This process continues until all four counts of I are complete and so all four of the lines specified in the ON . . . GOTO line have been selected in turn.

Please note that all of the same rules apply to the ON . . . GOSUB command.

A Utility

Subroutines can be built into a program to perform all sorts of useful functions. One such use is to make numbers conform to a given format. For instance, if you were producing a financial model of some kind that was to generate a number or a series of numbers as the result of a calculation, you might want those numbers to appear in the form of dollars and cents. Of course, some calculations might result in the number being more awkward, such as 6.8647. As you know, the 64 would respond to this number by printing it in its complete form:

6.8647

In order to turn this figure into six dollars and eighty-six cents,

6.86

you have to round it off to two decimal places. What's more, the normal convention in rounding is to round up any unwanted portion of a number if that portion is .5 or more. So 6.8647 rounds to 6.86 and 6.8657 rounds to 6.87.

There is a classic subroutine to handle this problem. By way of illustration, we'll look at it piecemeal. Type:

```
NEW
5 PRINT CHR$(147)
10 INPUT "1ST NUMBER";A
20 INPUT "2ND NUMBER";B
30 INPUT "3RD NUMBER";C
200 PRINT A
210 PRINT B
220 PRINT C
230 END
```

Run the program as it stands so far, inputting numbers with several decimal places, such as:

```
2.3865
5.8765
9.6111
```

You'll notice that the computer ingests your inputs and then gives them back to you exactly as you typed them in.

Now, the subroutine. To construct it we are going to assume that each number to be rounded will be put temporarily into an accumulator variable. And we're going to label the accumulator X.

The first thing to do is recognize that a number that is turned into an integer has all of its decimal places lopped off. But we wish to retain two such places. So, the strategy is to multiply the number by 100, then turn it into an integer and finally divide it by 100. Like this:

```
310 X=X*100
320 X=INT(X)
330 X=X/100
```

Now, to test this approach, type:

```
300 INPUT "X";X
RUN 300
```

When you RUN 300 the computer begins executing the program at that line and you are confronted with:

```
X ?
```

so type:

```
2.3865
```

and hit RETURN. The computer then proceeds to line 310 where 2.3865 is multiplied by 100, becoming 238.65. On line 320 this number is turned into an integer, 238. And on line 330 it is divided by 100, becoming 2.38.

We can confirm that this has occurred by typing:

```
PRINT X
```

We have successfully reduced the number in question to one having only two decimal places. However, 2.3865 should be rounded up to 2.39. To remedy this problem we should make our number bigger at the outset so that the rounding subroutine will work properly. Type:

```
300 X=X+.005
290 INPUT "X";X
```

Now run the program, starting on line 290 and again use

```
2.3865
```

as your input. You should type:

```
PRINT X
```

and be greeted with:

```
2.39
```

List the program starting at line 290 and do the math that is prescribed there on the back of an envelope. Your results should look something like this:

```
2.3865 + .005 = 2.3915
2.3915 * 100 = 239.15
239.15 turned into an integer = 239
239 / 100 = 2.39
```

Our subroutine now works, successfully reducing a number to two decimal places and rounding up when called for. The question is, how do we get the values A, B and C into and out of the subroutine? First, blank the line we wrote to test the subroutine. Type:

```
290
```

and hit RETURN. Next, we must move the values in our three variables into the accumulator, one at a time, so that the subroutine can work. This is handled very simply, duplicating the value in hand before we work with it:

```
40 X=A
50 GOSUB 300
60 A=X
```

The above shell game ought to be clear to you. On line 40 we've duplicated the input value of A in another variable, X. On line 50 we're going to send the computer to the rounding subroutine where X will be manipulated. Once this chore is complete we'll copy the new value of X back into the variable A as instructed on line 60. All we need to do now is perform the same action on variables B and C.

```
70 X=B
80 GOSUB 300
90 B=X
100 X=C
110 GOSUB 300
120 C=X
```

And, finally, we must turn our rounding subroutine into just that by giving it a RETURN command.

```
340 RETURN
```

Now, run the program and input as follows:

```
1ST NUMBER? 2.3865
2ND NUMBER? 5.8765
3RD NUMBER? 9.6111
```

and you should be delivered the following:

```
2.39
5.88
9.61
```

List the program and study it.

The power of a subroutine lies in your ability to manipulate virtually any aspect of a program with a single bit of code. You don't have to take the time and computer memory required by duplicating lines over and over. As long as you can convert individual and different variables into an accumulator variable, then the computer will work with a virtually infinite number of words or numbers in the same way, over and over.

However, you may have discovered that our subroutine for turning numbers into dollar and cents expressions isn't exactly perfect. Run the program once again and respond as follows:

```
1ST NUMBER? 2.3865
2ND NUMBER? 123.997
3RD NUMBER? 4
```

The screen should appear as follows:

```
2.39
124
4
```


While we've successfully eliminated superfluous decimal points and correctly rounded the numbers in question, the screen fails to present us with the sort of columnar financial statement one might prefer. A more familiar display of the information in question would be:

```
2.39
124.00
4.00
```

BASIC's control over the printing of numbers has remained absolute. Our rounding subroutine has changed the nature, in fact the value, of the numbers in our three variables. But we haven't taken control of the way these numbers are handled when printed on the screen.

Once again, BASIC is manhandling numbers by following its own rules. And the only way to exit from this dilemma is to stop working with numbers.

If you recall, all of the characters and keyboard actions are coded. The ASCII and CHR\$ codes in the back of the manual that came with your 64 list everything that can happen at the keyboard so that you can program a keyboard action to occur rather than having to physically type a key. The first line of many of our programs, PRINT CHR\$(147), is just such a device. It is put at the beginning of a program so that you don't have to hit the key in question yourself every time the program runs. To investigate how the organization of the ASCII and CHR\$ codes may be useful to us, type (and be careful to use the line numbers specified so as not to damage the program now in memory):

```
1000 FOR I=48 TO 57
1010 PRINT CHR$(I);" ";
1020 NEXT I
```

Before running this program, note that line 1000 defines the value of I as beginning at 48 and continuing to 57. Also, line 1010 instructs the computer to print the CHR\$ with each of the values of I. Finally, the first semicolon on line 1010 distinguishes the printing of the CHR\$ from the space between the two quotation marks. The second semicolon, the one that ends line 1010, keeps the computer on one line of the screen so that the printing to be done will appear on a single line rather than in a column. Run the program by typing:

```
RUN 1000
```

and

```
0 1 2 3 4 5 6 7 8 9
```

appears on the screen. This is because these are the CHR\$ 48 through 57. Now, dispose of these test lines. Type:

```
1000
```

and hit RETURN.

1010

and hit RETURN.

1020

and hit RETURN.

Though it may not have occurred to you, the numbers we have just printed on the screen are strings, not numbers. You know this because they have been dredged from their permanent locations in memory using a variable, CHR\$, that has a dollar sign at its end. To prove that these are strings, not numbers, try adding with them. Type:

```
PRINT CHR$(49)+CHR$(50)
```

This means: PRINT 1+2. Hit RETURN and you will see:

12

which is the two characters, not the numbers, combined.

STR\$

Interestingly, you too can turn numbers into strings. That is, you can take a value and convert it to a string. Type:

```
PRINT STR$(4)
```

and, at once, you see:

4

The 4 you typed was a number but the 4 that appeared on the screen as a result has been changed into a string. Once again, to demonstrate, type:

```
A$=STR$(4)
B$=STR$(8)
PRINT A$+B$
```

and you are greeted with:

4 8

which is a string made up of the two strings A\$ and B\$.

Turning numbers into words may not seem very useful to you at first. However, should you one day wish to create a program to balance your checkbook or compute your mortgage payments, the likelihood is you will want the results to appear in real-money form. In order to manipulate the way numbers show up on the screen, you should turn them into strings so that they can be moved around, added to and otherwise dressed up.

In the case of our rounding subroutine, this isn't very difficult to do. To

refresh your memory, the thing currently looks like this:

```
300 X=X+.005
310 X=X*100
320 X=INT(X)
330 X=X/100
340 RETURN
```

By the end of line 330 we'll have rounded the number to our liking. Then we must turn it into a string so that we can work with it. Type the following:

```
340 X$=STR$(X)
```

This line of code replaces the RETURN in our original subroutine with a line which translates as follows: "Establish a string variable called X\$ and then turn the value of X into a string so that it can be deposited in a string variable." Now, the rounded form of X is a string and is in variable X\$.

Next, we should display numbers on the screen in monetary form despite the fact they have no decimal places. We can manage this without a great deal of trouble because the value of X still resides in the numeric variable X. We haven't destroyed either the variable or its contents. So we can inquire as to the numeric form of X, testing the number in it to see what it is. The purpose here is to discover if the number has any significant decimal places. That is, if it has any numbers to the right of the decimal point which aren't zeros. If the number has this form, with just zeros to the right of the decimal point, it won't look like money when printed on the screen. That is, the number will appear as

9

instead of the more traditional

9.00

To test for the utter lack of significant fractions in a number, type the following:

```
350 IF INT(X)=X THEN X$=X$+".00":RETURN
```

This line of code reads, "If the integer of X, that is, X with its decimal places lopped off, is the same as the original X, then we know that the original X had no decimal places of significance (they were all zeros). So change the string called X\$ to have two decimal places tacked on to it in the form .00 so that the new X\$ will look like money." The RETURN is added to the end of this line so that, if we have discovered the problem with the number and have corrected it, no further work with the number will occur.

At this point, numbers which exit from our rounding routine that have two decimal places, such as 9.95, are in acceptable financial form. And those which are delivered that have no fractional part, such as 9, have been converted to 9.00. Running the program won't print the correct result on the screen yet, because X is being printed there. Soon, we'll remedy this flaw. All

that remains is to reconstruct numbers that have one decimal place, such as 9.1.

A Glitch

On the vast majority of microcomputers, this process is simple. To perform the trick merely requires multiplying the number by ten and then comparing the integer value of the number to the number itself. That is, 9.1 multiplied by ten is 91 and is therefore equal to its integer. Thus, the following will fix such a number:

```
360 IF INT(X*10)=X*10 THEN X$=X$+"0"
```

Sad to say, however, this perfectly legitimate line of code won't run on a Commodore 64. When contacted, the company that manufactures the computer had no explanation. It just won't work.

So, ever ingenious, your plodding authors have created a different way to reconstruct numbers with only one decimal place. But doing so requires an oddity or two. Here goes.

The curious problem confronting the programmer at this point is to deal with numbers as both strings and numbers. That is, to move back and forth between the two. As you know, we can turn a number into a string by using the STR\$ function. Similarly, BASIC provides the ability to turn a string into a number. The process is just as straightforward. You just ask for the value (VAL) of the string. Since only numbers have value, the value of a letter, or any set of keystrokes that doesn't begin with a number, will have a value of zero. To investigate just what VAL does, type (being careful to use our line numbers):

```
1000 INPUT "A NUMBER";A$  
1010 PRINT VAL(A$)
```

Note that we've kept this little program a good distance from the work we've been doing. Now run this program by typing:

```
RUN 1000
```

which starts the computer operating at line 1000. You are greeted with:

```
A NUMBER?
```

Respond to this with the keystroke:

```
4
```

You are then delivered the value of 4:

```
4
```

Rerun the program a few times to determine the value of the following:

```
99
187654
JOHN
R2D2
C3PO
1 FOR THE MONEY
```

You should receive the following responses from your computer:

```
A NUMBER? 99
99
A NUMBER? 187654
187654
A NUMBER? JOHN
0
A NUMBER? R2D2
0
A NUMBER? C3PO
0
A NUMBER? 1 FOR THE MONEY
1
```

It's possible that you have deduced the functioning of VAL. This command evaluates strings. If it finds that the entire string being considered is made up of numbers, it turns the string into a corresponding, matching value. If, however, the first keystroke in the string is not a number, then the string has a value of zero. If the first keystroke is a number, it is made into a value and the remainder is ignored.

Better yet, there's a very convenient exception. Run the program again and discover the value of:

```
-45
+45
```

These two keystrokes (the plus and minus signs), unlike the others we tested, don't short-circuit VAL. They are acceptable when used with numbers.

At the outset we intended to find a way to avoid problems inputting numbers and inadvertently offering letters. The solution, it seems, is to request numbers but accept them in the form of strings. Observe:

```
1000 INPUT "A NUMBER";A$
1010 A=VAL(A$)
1020 PRINT A
1030 PRINT A+5
```

VAL obviously offers the programmer the capacity to move back and forth between numbers and letters. And since BASIC looks kindly on the manhandling of letters, this offers a very real advantage to someone trying to work around a flaw in his computer.

But we still need a vehicle for dismembering a word so that its parts can be analyzed. BASIC provides this ability with three different commands: LEFT\$, RIGHT\$ and MID\$. This is neither the time nor the place for a description of all these. However, RIGHT\$ is about to come to our rescue. To examine its function, type:

```
PRINT RIGHT$("TODAY",3)
```

and hit RETURN.

```
DAY
```

appears on your screen. This is because the computer has been asked to print the rightmost characters in the string, starting at the far right and proceeding for three characters. That is, the last three letters in the word have been peeled off.

RIGHT\$ format is easy to work with. The command always precedes the statement and then, in parentheses, the string that is to be manipulated is given. A comma follows and then the number of characters to be pulled from the string is stated. That's all.

Having now added VAL and RIGHT\$ to our quiver, we can attack the problem of adding a zero to a number with only one decimal place. But before doing so, eliminate the four lines of code we created in investigating VAL. Type:

```
1000
```

and hit RETURN. Then type:

```
1010
```

and hit RETURN. Then:

```
1020
```

and hit RETURN. And,

```
1030
```

and hit RETURN. Presto, the lines have vanished.

Now we must take a look at a number the form of which has but one place to the right of the decimal, such as 9.1 or 145.8. Interestingly, such figures have a curious trait. If you consider the two characters to the far right of the number, you always are dealing with a decimal point and a number. Thus, unlike 9.11, which has the two far right characters equal to eleven, 9.1 can be sliced up to produce a fraction, .1, which is something less than one. At the same time, numbers like 900, which also have their two right-hand characters equal to less than one, won't get by the RETURN on line 350 because they are equal to their integers. Confused? Perhaps the code will help you to see the process:

```
360 IF VAL(RIGHT$(X$,2))<1 THEN X$=X$+"0"
```

This line translates as follows: "If the value of the two most right-hand characters in the string X\$ is equal to less than one, then there must be a decimal place, and only one decimal place, so add a zero to the end of this string." To be sure, we've gone far afield to solve this problem, but it is now fixed. Type:

```
370 RETURN
```

You should at this point pause to take note of the process involved. Faced with an unfair and annoying characteristic, the programmer has a choice. Quit or cheat. In all cases it is both more practical and more entertaining to cheat.

The final fix to be made in our program simply involves its PRINT statements. Obviously, since we've gone to such lengths to alter the nature of our strings in X\$, we should print the strings, not the numbers. So type:

```
60 A$=X$
90 B$=X$
120 C$=X$
```

and then change the PRINT statements to read:

```
200 PRINT A$
210 PRINT B$
220 PRINT C$
```

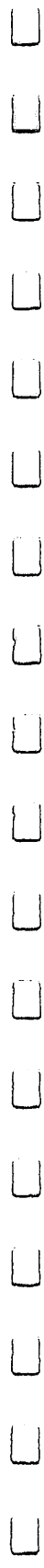
Now run the program, responding as follows:

```
1ST NUMBER? 4.9831
2ND NUMBER? 100
3RD NUMBER? 25.1
```

and you should be delivered:

```
4.98
100.00
25.10
```

You have successfully rounded numbers, converted them to strings, altered the strings to conform to normal representation of dollars and cents, and printed them on the screen. Pretty impressive. But you still haven't made the screen look like a column of financial figures. The decimal points aren't lining up. But this task requires a number of other commands that you haven't been introduced to. So, on to another chapter.



CHAPTER 10

Our most recent experiments have involved manipulating words and numbers before they appear on the screen so that they materialize there in a form we find acceptable. This endeavor isn't significant insofar as the actual crunching of numbers and words is concerned. Your results will appear before you in their accurate form, regardless. However, managing their format can sometimes be crucial, at least as that format affects how they are received and understood by others. So, since we are working with the look of the screen, let's continue to do so for a bit.

When last we considered the problem of displaying dollars and cents amounts, the problem had been nearly solved. We had successfully reduced all unwieldy numbers to two decimal places, rounding appropriately. And we had added one or two zeros to inappropriately formatted numbers. The subroutine that accomplished this is an independent utility that works on any program wherein such chores are required. Our only difficulty was in aligning the decimal points of the results when they appear on the screen.

There are two perfectly acceptable and independently interesting ways of accomplishing this end. First, we'll examine the simplest method.

TAB

Most typewriters include a key labeled "Tab" that moves the carriage across the paper before you to a predetermined location. It simply skips a programmed number of spaces before printing anything. BASIC contains such a device. It moves the cursor instantaneously to a position that has been

programmed in advance. To illustrate, type:

```
PRINT TAB(11);"HELLO"
```

As you can see, the HELLO appears rather more to the right of the screen than might otherwise be expected. This is the case because the TAB(11) in the PRINT statement acts as a spacing device. Translated, the command means, "Hit the space bar eleven times before printing HELLO." As you can probably guess, the (11) determines the number of spaces to be skipped before anything else is printed.

Note that the TAB command is in a PRINT statement. It is meaningless anywhere else. And also note that it appears before the thing to be printed. This occurs because the computer, ever literal, first prints the tabbed spaces and then prints the message you request. What's more, TAB can separate two or more things to be printed:

```
PRINT "W";TAB(1);"X";TAB(3);"Y";TAB(6);"Z"
```

This line results in ever-increasing spaces between the W,X,Y and Z. Your screen should look something like this:

```
W X Y Z
```

Thus, TAB is the single most direct method of controlling where the cursor begins printing input on your screen. And so it offers us a method of arranging things in ordered ways when they are reported on the screen.

TAB moves the printing action of the screen to the right where, depending on the length of a number or word, that word will be printed. And so it's not the complete solution to our decimal point-aligning problem. That difficulty involves numbers that are large (that is, have several digits to the left of the decimal point) and numbers that are small (that is, have few numbers to the left of the decimal point). Clearly, what we need to know now is how long a number is before we decide where to print it. Once again, BASIC comes to the rescue.

LEN

The BASIC statement LEN stands for "length." The LEN command inquires as to the number of characters in a string or number. The result of the inquiry is a whole, positive number that tells you how long something is. A few examples will help:

```
A$="ROBERT"  
PRINT LEN(A$)
```

You should now see

```
6
```

on your screen. This is so because a measurement of the length of ROBERT finds that there are six characters in the string. Now type:

```
A$="COMMODORE 64"  
PRINT LEN(A$)
```

and you should discover a

12

on your screen. This may seem incorrect. However, BASIC measures length by taking spaces into full consideration. Thus, the space between COMMODORE and 64 is considered a character and is therefore measured. BASIC also measures the length of numbers:

```
A$=STR$(40)  
PRINT LEN(A$)
```

You will note that the computer responds with a

3

Once again, there seems to be a discrepancy. But the number 40 in BASIC contains the unprinted but nevertheless real identification as to the sign of the number, i.e., positive. And this positive sign is taken into consideration when the length of the number is gauged. This is easier to understand if you try the following:

```
A$=STR$(-30)  
PRINT LEN(A$)
```

Again, you are told that the length of A is

3

but at least this time you can see the character, the minus sign, that is causing the measurement to be increased.

Combining the ability to measure the length of a string with the capacity to tab across the screen offers us the opportunity to manage the location of anything's appearance on the screen. All we must do is decide how far we should TAB before beginning to PRINT a word. And since the purpose here is to align all the decimal points and cents amounts, the clue comes in the entire length of the thing to be printed.

Mathematically, if you want things to align on their far right-hand side, you should tab less for long strings than for short ones. The longer the string, the less you should tab. Confusing? Think of it this way: we wish the last digit or character in our string to be in a certain position, regardless of the length of the whole string. So, the challenge comes in figuring out how many spaces to TAB. The solution, oddly enough, is quite simple. Since we are dealing in 40 characters per line of 64's screen, we should subtract the length of the string from a tab action of 40. Try this:

```
10 A$="35.76"  
20 B$="1234567.00"
```

```
30 PRINT TAB(40-LEN(A$));A$
40 PRINT TAB(40-LEN(B$));B$
```

Run this program to see how the two strings are produced on the screen. You should see the following:

```
35.76
1234567.00
```

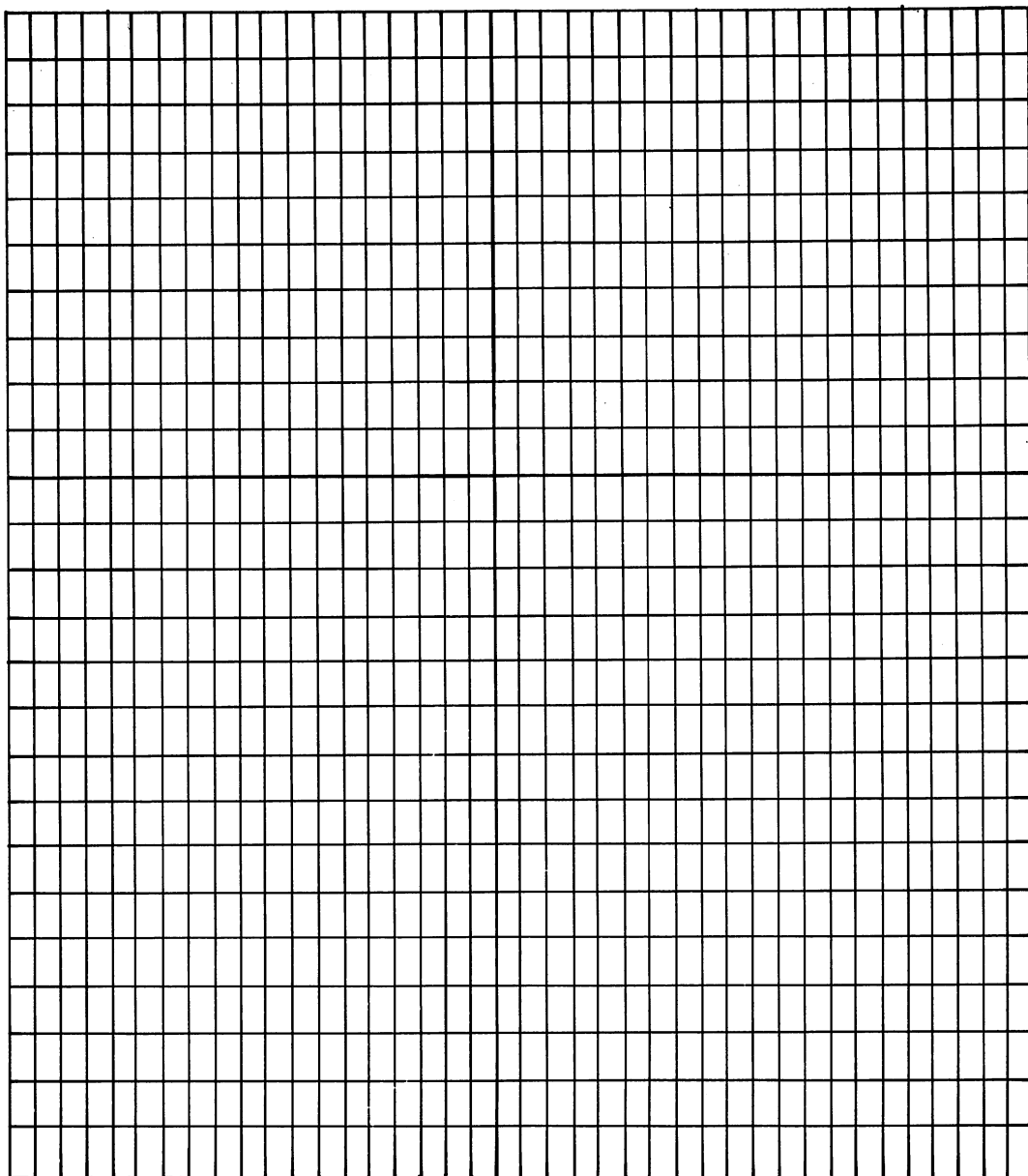
In effect, the computer here tabs across the screen to the right, bringing it to the far right character location on a line, the 40th spot. It then un-tabs, or backspaces, the length of the string it is dealing with. This is accomplished by the `TAB(40-LEN(A$))` portion of line 30. The computer tabs to the right and then moves back. Of course, the actual action of the program is to first subtract the length of `A$` from 40 and then to tab the resulting amount. But the effect is the same. In either case the last character printed on the screen on each line is the last character in the string.

If you are wondering about the plus and minus signs associated with the numbers we are measuring with `LEN`, don't worry. BASIC is thoroughly consistent here. It carries a number's unseen positive sign wherever it may go and so the length of a positive number and the length of a negative number, whether they be strings or numbers, are measured in the same way. Observe:

```
A=10
B=-10
A$=STR$(A)
B$=STR$(B)
PRINT LEN(A$)
PRINT LEN(B$)
```

The result in both cases is three because the plus in numerical values is carried over into string variables when you are using the conversion command `STR$`. The plus in 10 and the minus in -10 are both quite real and so are both taken into account.

If the foregoing has been slightly bewildering to you, pause to reflect on the real geography of the 64's screen. Insofar as BASIC is concerned, the screen is divided into quadrants, each of which will hold one character. There are forty vertical columns on the screen and twenty-five horizontal rows. So the number of positions in which you can make something appear can be gridded to look something like this.



With this map in mind, reread the last few pages. Remember that the length of something is measured relative to the number of quadrants it occupies and that the use of TAB counts one space, starting at the left, for each TAB requested.

The aligning feature we have just developed can be incorporated in the program we wrote earlier to round numbers and present them in a dollars and cents format. The end of that program need only be changed to print the resulting numbers on the screen in the manner we've just worked out. So that you can see the whole sequence, the program, altered to align the numbers input, is reproduced below. Please note that the aligning alteration appears on lines 200 through 220.

```
5 PRINT CHR$(147)
10 INPUT "1ST NUMBER";A
20 INPUT "2ND NUMBER";B
30 INPUT "3RD NUMBER";C
40 X=A
50 GOSUB 300
60 A$=X$
70 X=B
80 GOSUB 300
90 B$=X$
100 X=C
110 GOSUB 300
120 C$=X$
200 PRINT TAB(40-LEN(A$));A$
210 PRINT TAB(40-LEN(B$));B$
220 PRINT TAB(40-LEN(C$));C$
230 END
300 X=X+.005
310 X=X*100
320 X=INT(X)
330 X=X/100
340 X$=STR$(X)
350 IF INT(X)=X THEN X$=X$+".00":RETURN
360 IF VAL(RIGHT$(X$,2))<1 THEN X$=X$+"0"
370 RETURN
```

Another Approach

There is another, somewhat more versatile way of aligning numbers on a screen. It involves a program fragment that can be included in our number-management subroutine. It is more complex than the code we wrote to TAB across the screen. However, this method permanently alters the number in question so that it will forever behave in a financial way when making an

appearance on the screen. This feat is accomplished by tacking spaces on to the far left side of the numbers we are working with. Since blank spaces have computer substance, they must be printed on the screen. So if the proper number of them is added to every number, all such numbers will align.

To illustrate, consider that the two figures

```
34.95
200.00
```

fail to align on a screen because the 34.95 is printed beginning with its 3. Were we to insert a blank space to the left of the three, the space would be printed and the two numbers would align. Thus, the trick is to figure out how many blank spaces to add to each number, accumulate them in some sort of variable and then tack them on to the number in question. The loop that accomplishes this has its ceiling fixed by the length of each number. Observe:

```
380 IF LEN(X$)=40 THEN RETURN
390 SP$=""
400 FOR I=1 TO 40-LEN(X$)
410 SP$=SP$+" "
420 NEXT I
430 X$=SP$+X$
440 RETURN
```

Line 380 of this program part states that if the number being worked with is 40 characters long, we will accept it as it is. That is, 40 characters is the total width of the column we wish to work with. So, no blank spaces need be added to such a number. Line 390 establishes a string variable into which we are going to put blank spaces. Hence the variable name SP\$. Because this is a subroutine that will be used over and over, we have to reset the contents of this variable to nothing. Were we not to do so, the spaces that had been accumulated in SP\$ during the last operation of the subroutine would be carried over into this operation. So, on line 390 we define the contents of SP\$ as a void, known as a "null string." There is nothing whatever between the beginning and ending quotation marks on the line.

Line 400 counts the number of spaces we should add to each number. Since line 380 will short-circuit this program if the number needs no spaces at all, the counter, I, is set to begin at 1. We know we need one blank space added at this point or the number wouldn't have gotten this far. The purpose of the loop here is to make the new number a total of 40 characters long. So we loop as many times as needed by subtracting the length of the original number from 40.

Line 410 accumulates a blank space with every cycle of the loop and strings them together in SP\$.

Line 420 closes the loop. And line 430 redefines the original number, X\$, as that number with the accumulated number of blank spaces tacked on to it.

If you insert this into your subroutine for rounding numbers, you should change lines 200 through 220. The TABs there are no longer necessary.

Thus, the lines should read:

```
200 PRINT A$  
210 PRINT B$  
220 PRINT C$
```

The simplicity of these PRINT statements is precisely the advantage of this method of manipulating numbers. When you've altered the number in A\$ once, it remains altered permanently. If you need to print it more than once, it is easy to do so. On the other hand, moving the number in A\$ to a position different from the one originally established for it becomes more difficult. You have to lop off some of the blank spaces.

To complete the changes to this program we must include this approach in our original subroutine. So, eliminate line 370. Type:

```
370
```

Also, the RETURN on line 350 has to be changed so that the program will always include our addition. Type:

```
350 IF INT(X)=X THEN X$=X$+ ".00":GOTO 380
```

Now, run the program and see how it works.

CHAPTER 11

In the chapter just concluded we developed methods of controlling the nature and appearance of the numbers your computer prints on the screen. As you might imagine, this is fertile ground which this book can only begin to explore. However, you now possess the rudiments and, with some imagination, can try expanding on your skills.

This might, therefore, be an appropriate time to look at how you can manage the other end of the pipeline: the inputs a person enters into the computer. In virtually all of the programs we have written so far, you have been the programmer as well as the keyboarder when the program ran. Because you know what the computer is looking for, you have little trouble typing appropriate responses. However, among the many commonly issued complaints concerning computers is their stalwart refusal to assist the uninitiated.

Imagine, if you will, having created a program for your next-door neighbor's 64 so that he could analyze how he spent his money during the year. It's a nifty little program that allows him to enter the amount of each check, the date on which he wrote it and the purpose of the payment.

At about midnight of the evening he undertakes the labor of entering all his data, he is closing in on the end of the work. Nearly 250 checks have been entered and he has only a half-dozen or so to go. Then, in his weariness, he accidentally types Q instead of 1 and hits RETURN.

Both you and the 64 know that he has offered a string to the computer when it wants a number. The BASIC language immediately issues the following error message:

?REDO FROM START

Of course, the computer is requesting him to redo only the last input prompt and is expecting him to begin from the first character in that data. Hence, the START. But your friend and neighbor doesn't know that. So he responds to the query on the screen by typing

NO

since he has no intention of entering all 250 checks again. Naturally, BASIC thinks NO is as much a string as Q, so it once again states

?REDO FROM START

Frustrated, and certain you are a rank amateur programmer, your friend turns off the computer, sending all his work into the never-never land known as zapped.

It stands to reason, in light of this odious possibility, that you and the computer ought to conspire to check inputs before they are turned over to BASIC. The language isn't very helpful if you don't speak it, so more explanatory messages and easier-to-understand assistance should be available. Fortunately, BASIC is beautifully equipped to allow the programmer just such luxuries.

As you know, all of the characters and keyboard actions are coded in the ASCII and CHR\$ codes. Only a few pages back we used this list to manipulate numbers. Of course, you know that the characters and their code numbers are published in the booklet that came with your computer. But you can also dredge them from memory if you wish.

The program below prints a specific group of ASCII symbols on the screen. It is limited to those that have nongraphic uses and it avoids several other categories of symbols that would cause confusion on the screen as things were printed. Put differently, we're only printing a selection of characters, the group we want to deal with here.

Notice also that there is a nested loop in this program that briefly delays the printing on the screen so that you can watch the characters appear one at a time.

```
NEW
5 PRINT CHR$(147)
10 FOR I=33 TO 95
20 FOR D=1 TO 200
30 NEXT D
40 PRINT CHR$(I);" ";
50 NEXT I
```

Line 40 contains two instructions. The first is to print all the CHR\$ between 33 and 95 and the second is to skip one space (there is one space between the quotation marks) between each.

Running this program will produce a listing of all the symbols you

normally use when sitting at a typewriter. Included are all the digits, 0 to 9 (CHR\$ codes 48 through 57), and all the letters of the alphabet (CHR\$ codes 65 through 90). Also note that the period (or decimal point) is included. It is CHR\$ 46.

BASIC stores these keystrokes in memory in the form of strings for a good reason. A string variable will accept anything you can type on a computer. Letters, numbers, multiplication signs, graphic symbols, the instruction to clear the screen, anything. But a numeric variable will accept only numbers. Thus, storing the symbols as strings makes virtually any work you want done with them possible. Except numerical operations. Only numbers can perform this feat.

The problem confronting the programmer is, therefore, discovering a way to accept numbers as strings but still perform mathematical operations with them. If this can be accomplished, we will avoid BASIC's error messages during input because a string variable will accept anything.

As you know, BASIC solves this problem by allowing you to convert a string value into a numeric value. The process is utterly straightforward. You just ask for the value (VAL) of the string. Since only numbers have value, the value of a letter or any set of keystrokes that doesn't begin with a number will have a value of zero.

However, a program should not accept a letter when a number is called for simply by inputting it into the computer and then converting it to a value. Avoiding BASIC's mismatch error message in this way might result in a mistakenly typed letter being used in a computation as a value. This is because VAL turns erroneously typed words into a perfectly acceptable number, zero, that will work perfectly in most computations. Thus, we might never be aware of a typing error. So, we must check the input once we've accepted it to see that it's a number. We can accomplish this end by accepting characters one at a time and evaluating each before proceeding to the next.

GET

The GET command is very much like the INPUT command, except it doesn't pause to wait for your input and doesn't require the person at the keyboard to hit the return key. That is, GET ingests material constantly, immediately putting into an assigned variable whatever the keyboard is sending.

One of the difficulties of GET accepting keyboard input constantly is that it will accept nothing at all. Which is to say, if you aren't typing for a nanosecond, the command will ingest the nothing you've typed and blank whatever is in the variable. Since even the most experienced writer can't type at nanosecond intervals, GET poses a problem. But it's one that is traditionally managed by constructing a loop. Try this:

```
NEW
5 PRINT CHR$(147)
10 PRINT "TYPE A NUMBER ";
```

```
20 GET A$
30 IF A$="" THEN 20
80 PRINT A$
```

This program operates using the null string mentioned earlier. Line 10 prints the request for a number on the screen, along with one blank space after NUMBER. Line 20 then sets up a GET statement that is requesting a string to put in A\$. Line 30 states that if A\$ contains nothing (which it will contain until you hit a character), then the program should recycle to line 20.

Running this program will present you with the request for a number. But you will quickly notice that there is no blinking cursor on the screen. This is always the case with GET. Think of it this way. By the time the screen can produce a cursor for you to see, the GET statement has already ingested the nothing coming from the keyboard and is in the process of recycling. At any rate, if you want a cursor in GET, you have to create it yourself.

Also note that there is a semicolon ending line 10. This punctuation mark, as you know, keeps the program from skipping to the next line. Thus, the computer is waiting to GET something one position beyond the blank space in "TYPE A NUMBER " and will wait there until you strike a key.

Run the program. First you see:

```
TYPE A NUMBER
```

and nothing more. When you hit 7 you see:

```
TYPE A NUMBER 7
READY
```

You exited the program after the GET statement accepted only one digit. Clearly, this command is difficult.

But the impetuosity of GET has its uses. Because you need not hit RETURN to enter a number into GET's assigned variable, the program will continue running while you type. Which is to say that one keystroke at a time can be absorbed by the computer, analyzed in some way and then printed on the screen. But before we check each keystroke, we need to store each somewhere. Once again we need a kind of accumulator to hold characters until we are done with an entire input. Add this accumulator to the program:

```
70 A2$=A2$+A$
```

and change line 80 to conclude with a semicolon:

```
80 PRINT A$;
```

Now send the computer back to GET a new character after it has printed the last one received:

```
90 GOTO 20
```

When you run this program you are presented with:

```
TYPE A NUMBER
```

When you hit a digit you see:

```
TYPE A NUMBER 7
```

If you hit another you are offered:

```
TYPE A NUMBER 75
```

and so on and on as you offer the GET statement more characters and it continually deposits them in the accumulator, A2\$.

Now, we can type in a longer number if we wish, checking each digit as we go. The check, or error trap as it is called, compares the contents of A\$ with the CHR\$ code numbers to see what it is that has been typed into A\$. By dealing with keystrokes one at a time and making this comparison, we can avoid typing errors. First, hit the RUN/STOP key. Then add:

```
60 IF A$<CHR$(48) OR A$>CHR$(57) THEN 20
```

This line immediately follows the line of programming that recycles the GET statement until something is typed. Thus, as soon as you hit a key, GET puts that character in string variable A\$. This keystroke is then compared to the CHR\$ code numbers. If the code number for the keystroke is less than 48 or if the code number is greater than 57, the program knows that some key other than a digit has been pressed. Line 60 therefore sends the computer back for another try. It doesn't allow the new bogus keystroke to be added to the accumulator and it doesn't allow it to be printed on the screen. As it now stands, this program won't put anything other than a number into A\$, so it never gets to A2\$. The whole program, so far, is below.

```
5 PRINT CHR$(147)
10 PRINT "TYPE A NUMBER ";
20 GET A$
30 IF A$="" THEN 20
60 IF A$<CHR$(48) OR A$>CHR$(57) THEN 20
70 A2$=A2$+A$
80 PRINT A$;
90 GOTO 20
```

Run the program and try typing some numbers and letters. You'll find that the letters aren't accepted. You'll also find that there is no way to stop the program. A2\$ keeps growing. Uncontrollably. To stop the program, hit the RUN/STOP key.

What we need now is an exit, some way of concluding our entries into the accumulator. Again, the CHR\$ code numbers help. Enter these lines:

```
40 IF A$=CHR$(13) THEN GOTO 100
100 END
```

CHR\$(13) is the return key. So as soon as you hit it, the program will skip from line 31 to 60 where it will end. The return key won't be printed on the screen (of course) and won't be added to A2\$.

Now run the program and hit RETURN after a few keystrokes. You should now be greeted by the READY prompt since the RETURN sends you to END. Next, try a number with a decimal in it. Type:

```
RUN
TYPE A NUMBER 32.95
```

The period, as you know, is CHR\$(46), a code number not accepted by our program. Naturally, we should fix this flaw. But not in the manner of simply adding it to the list of acceptable keystrokes. Since we are trying to guarantee that only real, honest-to-goodness numbers can be typed, we should make sure that only one decimal point is accepted into A2\$. After all, numbers can have only one decimal point. We'll use variable D to count the number of decimal points that have been typed:

```
45 IF A$=CHR$(46) AND D=1 THEN 20
50 IF A$=CHR$(46) THEN D=1: GOTO 70
```

These lines perform as follows: Line 45 checks the keystroke that is in A\$ to discover if it is a decimal point. If it is and the value of D is equal to one, then the computer recycles to the GET statement again, not registering the decimal point. Thus, if something has raised our counter from its beginning value of zero to one, no more decimal points will be accepted. If, however, D equals zero, as it will when we first run the program, then line 50 will execute. There, if the character being analyzed is a decimal point, the decimal point counter is raised to the value of one. Line 50 jumps the program over our normal no-letters-or-other-symbols-allowed error trap on line 60. It does so by sending the computer to line 70. On line 70, the decimal point is deposited in A2\$, our accumulator.

At this point in our analysis of this program, it is only fitting that you be passably perplexed. A variety of checks, programming maneuvers and other chicanery has been going on. However, the gist of the program should be understandable to you. But, by way of review, it seems fitting to cover the more arcane aspects of the new material we've just dealt with.

First, GET. This is a strange command primarily because it initiates an immediate action. Whenever the computer is running a program and encounters the instruction GET, it considers it much as it might the PRINT command. As quickly as possible, the computer does what it is told. When you use GET, therefore, the computer goes immediately to the keyboard for its input rather than waiting patiently as INPUT waits. If the keyboard has nothing to offer, GET absorbs the nothing as though it were a real bit of data.

Thus, GET asks for information in a voracious fashion. And so, programmers traditionally create a loop to force the GET command to go get more if it hasn't managed to get anything yet. They do this by building a conditional statement that says, "Okay, GET, you went and retrieved nothing. Try again." This loop looks like this:

```
20 GET A$  
30 IF A$="" THEN GOTO 20
```

The important point is that GET is like a dog in search of a bone. Unless you tell it to keep looking, it stops.

Second, you can consider keystrokes one at a time and compare them with the established CHR\$ codes in the 64's permanent memory. Because certain kinds of characters exist in certain ranges of the code numbers, you can admit only those that pass muster and reject the rest. This rejection has no particular bearing on the information you've taken in because, in mere nanoseconds, GET will replace the unacceptable with either nothing at all or a new keystroke.

Finally, you can absorb numbers in the form of strings. Doing so keeps the computer's BASIC language from controlling your program because almost anything can be taken into a string variable. Once there, of course, you can analyze it to your heart's content, discarding any string you don't like and absorbing anything you consider acceptable.

Of course, once you've accepted a string made up of digits and one decimal point, you have to turn it back into a number so that you can add, subtract, multiply, divide, etc., with it. But doing so is easy. You just take the string's value. Since you've already kept letters and other aberrant symbols from becoming part of the string, you can be confident that it is a number. So you complete the program we've been working on by typing:

```
75 A=VAL(A2$)
```

and in a flash your string is a number residing in a numerical variable. To observe this fact, run the program. Input a number, trying several letters and a couple of decimal points. Then hit RETURN. When the READY prompt appears, type:

```
PRINT A
```

and your number will appear, its invisible plus sign moving it to the right. Then type:

```
PRINT A2$
```

and you'll find the number (a string) is flush to the left of the screen.

The Beginning and the End

As a program structure that you can use wherever it seems appropriate, we've now constructed a fairly workable form of input and output control. We've trapped the kind of mistakes that can cause all sorts of trouble and we've given numbers a form that looks good on the screen.

However, the methods we've concocted are somewhat unwieldy, at least in certain circumstances. And they are hard to imagine as part of a much

larger program, one that is designed to do real work. What needs to be done is a sort of standardization of the processes in question so that we can turn them into subroutines that can be affixed to nearly any practical numerical program. But to do so requires introducing a few simple concepts.

Starting Over

When we establish a counter or an accumulator variable in a program, we know at the outset that the value of that variable is equal to zero. When the program first runs, everything is equal to zero until we establish otherwise. This is the case because all variables equal zero, or contain nothing, to begin with. Thus, we can safely count or accumulate in such a variable assuming that nothing is there.

When such a variable is put inside a loop or used in a subroutine, however, we can no longer make this assumption. The first time the subroutine is executed, the variable will equal zero. But the subsequent uses of the subroutine will pick up any value that prior uses have deposited in the variable. Thus we have to "reset" the variable to zero whenever we enter the subroutine.

(We're about to disappear everything in your computer's memory. If you are pleased with the work there, fear not. We'll sum up, offering a better format, shortly.)

Assume for the moment that we wish to create a program that will limit the length of a typed word to four letters. Such limits are commonly set in programs that accept lists of names or items in an inventory. In fact, code numbers and even the length of raw figures to be calculated are sometimes limited by the design of a program. At any rate, here we'll work with a string of four letters and no more.

We can accomplish this end by establishing a variable in the program that will keep track of every keystroke that the word is made of. When there are four such keystrokes, the computer will then stop accepting them. The program is below.

```
NEW
 5 PRINT CHR$(147)
10 PRINT: PRINT "TYPE A WORD ";
20 GET A$
30 IF A$="" THEN 20
40 PRINT A$;
50 L=L+1
60 IF L=4 THEN 10
70 GOTO 20
```

This program appears relatively straightforward. The GET function on lines 20 and 30 behave in exactly the same way as our earlier GET statements. One character at a time is ingested and deposited in A\$. The rub lies on lines 50 and 60. Here we've established a variable that is to count the

number of characters that have been input. With each cycle through the GET statement, the limiting variable L will increase by a count of one. So when four characters have been typed (when L=4), then the computer is sent to line 10 where a new word is begun.

Running this program will result in the following:

```
TYPE A WORD FOUR
TYPE A WORD FOURFOURFOUR
```

This less than ideal result is caused by the failure of variable L to perform as intended. On the first cycle through the program the word FOUR is produced, L counts the characters in it and, when four have been accepted, the program recycles. This prints TYPE A WORD on the screen once again. However, as you start typing again, variable L grows from a value of four to five and then six and so on. Since the limiting factor, the IF statement on line 60, reroutes the program when L equals four, the program recycles only once. So to continue measuring the words as they are fed to the computer, we have to reset the value of L every time we start a new word. Type:

```
15 L=0
```

and rerun the program. Now you will see that the thing functions as it should. While we are building individual words using the GOTO 20 on line 70, the value of L grows with each character. When the four-letter word is complete, we recycle on line 60 with a GOTO 10. Immediately thereafter, on line 15, we clear the value of L and start counting all over again.

This simple program serves to illustrate the importance of clearing any sort of variable that may grow with every cycle of a subroutine. It's simple enough to do. Just set the value of the variable at zero from the beginning of the subroutine. The difficulty comes not in the programming technique, but in remembering to employ it.

This same sort of difficulty can arise in an accumulator. For instance, let's accumulate the four-letter words we're building in a variable called A2\$, just as before. Type:

```
45 A2$=A2$+A$
60 IF L=4 THEN PRINT " ";A2$:GOTO 10
```

These two lines build the word in a single variable and then print it on the screen when it is complete. Since we've fixed the counting variable, L, it should be four letters long every time. Running the program will cause the following mess to appear on the screen:

```
TYPE A WORD FOUR  FOUR
TYPE A WORD FOUR  FOURFOUR
TYPE A WORD FOUR  FOURFOURFOUR
```

You are typing four-letter words and they are being terminated by the counting variable. But the accumulator variable, A2\$, is constantly being increased. Just as the counting variable, L, has to be reset for every word, so

too must the accumulator. This is accomplished by

```
17 A2$= ""
```

Notes

When you write a more or less substantial program, you invariably include in it some convoluted thinking. More often than not your imagination fabricates variables with names that aren't always obvious. More important, the twists and turns of loops, subroutines and GOTOs can become Byzantine after a while. Days, even hours, later you can list such a program only to find you can't figure out what you were trying to do or how you thought you were going to do it. The program won't make any sense because the logic of it isn't clear and you haven't left any clues behind.

Of course, you could take notes on every program you write in some sort of loose-leaf binder. Then you could comment on your work, describing the lines of code you've written. But this isn't necessary. BASIC lets you take notes in the program. You can type in comments of any kind and the computer will list them when you list the other lines of code. To do so you use what is known as a REM statement.

The REM statement is very versatile. You type it on a line in your program, giving it a line number like any other programming statement. After the remark, REM, you can make any notes you like, typing even reserved words in the BASIC language. Thus, you embed hints as to your thinking in the program. An example follows:

```
NEW
10 REM A WAY TO NOTE
20 PRINT "A REM PROGRAM"
30 PRINT "WITH REM IN IT"
```

Run this program. On the screen you see:

```
A REM PROGRAM
WITH REM IN IT
```

Now list the program. Line 10 is still there, telling you what the program was for. But it was never involved in the operation of the program. It didn't actually do anything.

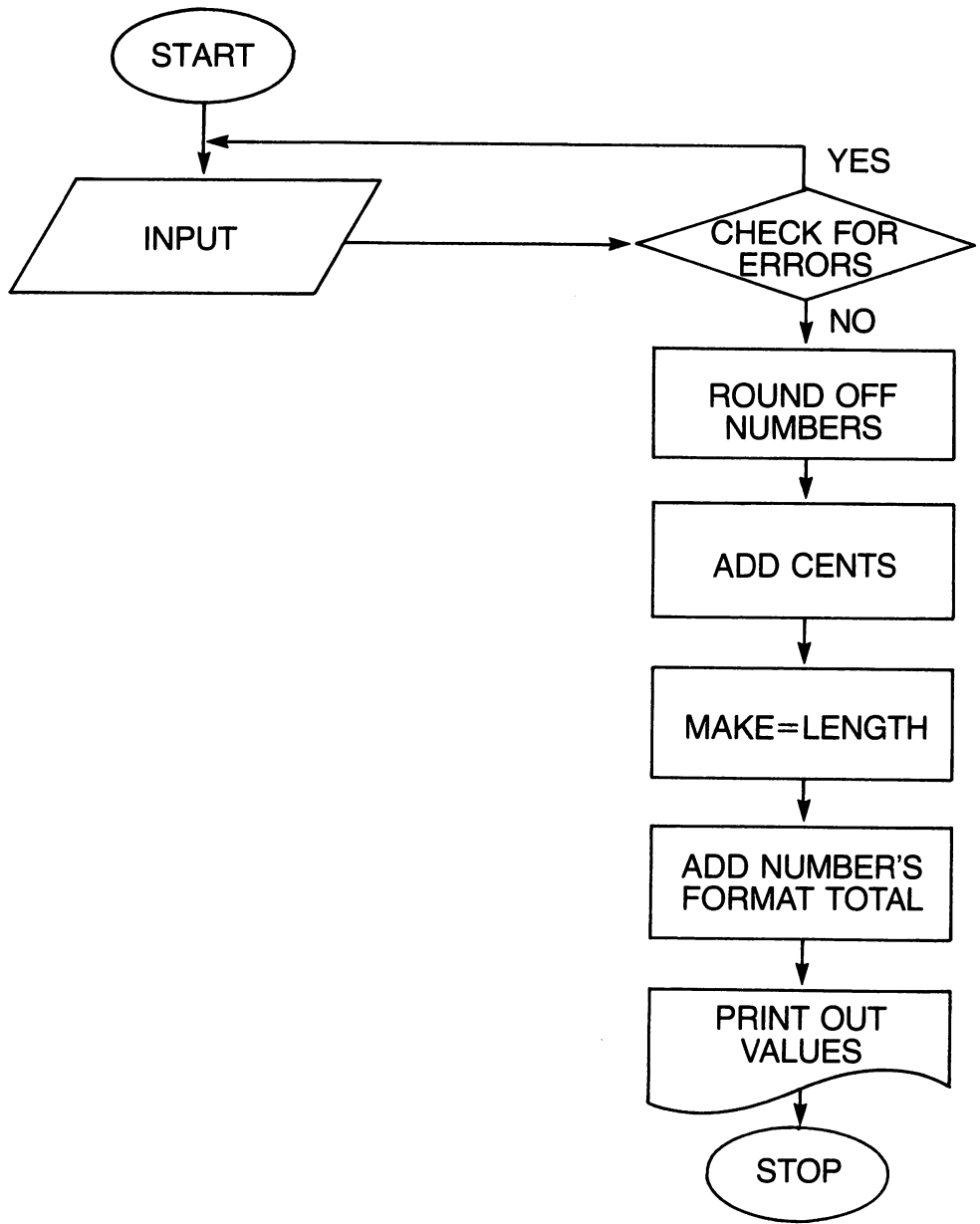
Put such statements in all your programs. Label sections and describe actions so that when you come back to the program you'll know what you were doing and how you were doing it.

Little Boxes

Throughout this book you've been introduced to the idea of programming structure in a rather oblique way. We've built programs bit by bit, adding features here and there and manipulating words and numbers in many ways. And in every case we've followed a kind of unspoken rule that is inherent in all computer programming. That rule states that there are three parts to any program: the information portion, the processing portion and the report portion. All computer programs have to be given information in some way. Then they work with that information. And finally, if the work is to be of any use, they report on their work.

Many programmers draw diagrams of their program's logic before actually writing the lines of code that will operate in the computer. They do so in order to organize their thinking and to check their logic before going to work. Thus, before we develop a fairly sophisticated program to manage numbers, we ought to draw a simple diagram of how the program in question is going to work. In case you are wondering, such sketches are called "flow charts" because they map the "flow" of work through a program the way a riverbed charts the spring rains.

Below you'll find a flow chart of a program that will take three numbers and add them together. No serious piece of work, to be sure. The real purpose of the program is to demonstrate a set of subroutines that will manage our input of data and organize the output. The three subroutines check keyboarding for errors, round numbers off to dollars and cents and then align the decimal points of the numbers so that they appear on the screen in a traditional way.



As you can see, this chart is relatively simple. It maps the order in which we will perform various activities and isolate one process from another. If nothing else, it helps to clear your head before we begin. Also, you'll notice that there are a few REM statements inserted into this program so that the chart and our notes in the program can be tied together. Well, here goes.

```
NEW
  5 REM INPUT
 10 PRINT CHR$(147)
 20 PRINT "THREE NUMBERS TO ADD"
 30 PRINT:PRINT"1ST NUMBER ";
 40 GOSUB 1000
 50 N1$=A2$:N1=X
 60 PRINT:PRINT"2ND NUMBER ";
 70 GOSUB 1000
 80 N2$=A2$:N2=X
 90 PRINT:PRINT"3RD NUMBER ";
100 GOSUB 1000
110 N3$=A2$:N3=X

200 REM ADD THE NUMBERS
210 T=N1+N2+N3

300 REM FORMAT THE TOTAL
310 A2$=STR$(T)
320 GOSUB 2000
330 T$=A2$

400 REM PRINT RESULT
410 PRINT:PRINT
420 PRINT N1$
430 PRINT N2$
440 PRINT N3$
450 D$=""
460 FOR I=1 TO LEN(T$)
470 D$=D$+"-"
480 NEXT I
490 A2$=D$
500 GOSUB 2500
510 D$=A2$
520 PRINT D$
530 PRINT T$
540 END

1000 REM GET THE DIGITS
1010 A2$=""
```

```

1020 D=0
1030 GET A$
1040 IF A$="" THEN 1030
1050 IF A$=CHR$(13) THEN 1500
1060 IF A$=CHR$(46) AND D=1 THEN 1030
1070 IF A$=CHR$(46) THEN D=1: GOTO 1090
1080 IF A$<CHR$(48) OR A$>CHR$(57) THEN 1030
1090 A2$=A2$+A$
1100 PRINT A$;
1200 GOTO 1030

1500 REM ROUND OFF
1510 X=VAL(A2$)
1520 X=(INT((X+.005)*100))/100
1530 A2$=STR$(X)

2000 REM ADD CENTS
2010 IF INT(X)=X THEN A2$=A2$+".00":GOTO 2500
2020 IF VAL(RIGHT$(A2$,2))<1 THEN A2$=A2$+"0"

2500 REM FORMAT TO ALIGN
2510 SP$=""
2520 IF LEN(A2$)=40 THEN RETURN
2530 FOR I=1 TO 40-LEN(A2$)
2540 SP$=SP$+" "
2550 NEXT I
2560 A2$=SP$+A2$
2570 RETURN

```

As you review this program, notice that no single part of it is overwhelming. Each chunk of this code is a separate process that manipulates either numbers or strings in a specific way. What's more, no element of it involves any skills that haven't already been discussed. We've simply combined a considerable number of logical, but not complex, BASIC commands to perform a series of actions.

Walk through the program slowly, making sure you follow the computer as it is directed by the program. Take notes. Write in the margins. Refer to earlier chapters. Have a good time.

One Quick Note

As you can see from the program just completed, the GET statement allows all sorts of character-by-character analysis. The immediacy of the computer's actions when under the control of a GET statement allows very detailed and rapid management of anything that happens at the keyboard. But you may have wondered how complex the analysis of such activity can

become. After all, if the computer "disappears" to process a keystroke, analyzing whether it is acceptable or not, what will happen to a pressed key if the GET command isn't available to pick it up?

The answer lies in the keyboard. It contains a very small amount of memory. Just enough, in fact, to store ten keystrokes. The keyboard "buffer," as it is called, holds these strokes until the computer's main memory is available to absorb them. A short program will demonstrate:

```
NEW
  5 PRINT CHR$(147)
 10 PRINT "TYPE LETTERS ";
 20 GET A$
 30 IF A$="" THEN 20
 40 PRINT A$;
 50 FOR I=1 TO 500
 60 NEXT I
 70 GOTO 20
```

When you run this program you are greeted with the request to TYPE LETTERS. If you do so very fast, hitting one key over and over, you'll find that its character appears on the screen slowly. This is because the pause loop on lines 50 and 60 occupies the computer for a second between the instances of the GET statement on line 20 being available to read the keyboard. Thus, while the computer is counting to 500, the keyboard stores the keystrokes for you. When line 20 once again becomes active, the keyboard delivers the next character typed on it to the program. To break out of the program, hit the RUN/STOP key.



CHAPTER 12

The model program just completed demonstrates the computer's ability to control the information it takes from the keyboard and deposits in memory. It also transforms numbers into the form we desire, in this case, dollars and cents. All things considered, it is a fairly attractive piece of work. And you ought to feel a certain twinge of satisfaction if you have managed to understand how it operates.

However, in spite of the work that went into it and regardless of the cleverness buried in some of its parts, the thing doesn't do much. It is designed to add three numbers together. That's all the program is capable of accomplishing. From earlier programs you know that we could modify it to be a calculator that would add, subtract, multiply, divide and raise things to certain powers. But at all times such a program would be dealing with two or three numbers and a single arithmetic process.

Clearly, what's needed is a method of increasing the total number of variables that the computer can consider at any one time. And this must be accomplished in a relatively efficient way.

For this reason, those who program computers are often obsessed with the compactness of their programs. To be sure, tighter, more economical programs tend to run faster. But they also require fewer lines of code and take up less space in a computer's memory. Therefore, leaner programming affords you more powerful programs without the expense of additional memory.

Naturally, all this is by way of introducing yet another BASIC statement. It is by far the most abstract of the concepts we've addressed in this book.

Yet once mastered, it allows you to perform remarkable feats with far less chance of error and unbelievable economy of programming. The bottom line here is that one line of a program can be made to do the work of literally hundreds of lines.

Let's begin by considering the act of establishing locations in memory that are to contain words. Let's say we want to store the first, middle and last names of someone. Each of these strings should be placed in distinct memory locations so that we can deal with them separately when necessary. We could easily program a solution to this problem as follows:

```
NEW
5 PRINT CHR$(147)
10 INPUT "FIRST NAME";A$
20 INPUT "MIDDLE NAME";B$
30 INPUT "LAST NAME";C$
```

Running this program fragment presents you with prompts that request the three strings, one at a time. After you have typed each, the three variables, A\$, B\$ and C\$, contain the names. Simple and to the point. But imagine if you intended to record the first, middle and last names of all the members of your extended family. In such a case you might need twenty or thirty first name variables and an equal number of middle and last name variables. Or as many as sixty string variables.

After considering this dilemma for a while, you might well conclude that each person should receive a specific letter of the alphabet to key his three variables. So your grandfather might be assigned variables A1\$, A2\$ and A3\$, your grandmother B1\$, B2\$ and B3\$, and the rest of your family would follow accordingly.

While there's a certain method to this madness, it's clear that such a program is going to contain a lot of code. Every single input will require a line. Still, you're headed in the right direction.

DIM

The BASIC statement DIM means "dimension." It is used when the programmer wishes to establish a series of variables to contain a group of related data. Improving on the idea of grouping each member of your family's name in a pattern of alphanumeric variables such as A1\$, A2\$ and A3\$, DIM allows you to establish such patterns with relative ease.

When a variable is given dimension using DIM, computer folks refer to the resulting pattern of variables as an "array." This word hails from the military and describes the orderly arrangement of troops as they prepare for battle. This is precisely the sort of parade that DIM produces: a simple, two-dimensional series of patterned variables. Observe:

```
NEW
5 PRINT CHR$(147)
```

```

10 DIM A$(3)
20 INPUT "FIRST NAME"; A$(1)
30 INPUT "MIDDLE NAME"; A$(2)
40 INPUT "LAST NAME"; A$(3)

```

The statement on line 10 tells the computer that it soon will be establishing three different locations in memory that belong to the "group" or "array" of variables known as A\$. Thereafter, on lines 20 through 40, those variables are named and strings are deposited in them.

Think of A\$ as a staircase. The particular one in question has three steps, each numbered. And in an array of this sort the numbers of each step appear in parentheses after the array variable's name.

Of course, using DIM hasn't shortened our program a wit yet. But since you are now familiar with the process of counting with the computer, you can probably imagine that there is a simple way to produce loads of variables using DIM. All we have to do is get the computer to do the work of numbering them. And what better tool for this work than a FOR . . . NEXT loop?

```

NEW
5 PRINT CHR$(147)
10 DIM A$(3)
20 FOR I=1 TO 3
30 INPUT "NAME";A$(I)
40 NEXT I

```

Running this program will present you with

```

NAME ? JACK
NAME ? DAVID
NAME ? MARTIN

```

after you've filled in the three names. In effect, you have established the three variables one at a time, setting each up with a single cycle of the FOR . . . NEXT loop. As the counter in line 20, labeled I, progresses from 1 to 3, it deposits its current count in the newly established array, A\$(I). This happens on line 30 as each successive variable is named. Needless to say, line 10 is essential here. It warns the computer that the DIM form of variable names is about to be employed and that there will ultimately be a certain number of them used. In this case, three.

You can add to the program just written so that it will print the contents of its array on the screen. To do so, use a technique that mirrors the method of input. Simply set up another loop to haul the contents from each variable in the array and then print it.

```

50 FOR P=1 TO 3
60 PRINT A$(P);" ";
70 NEXT P

```

Now when you run the program your screen will look like this:

NAME ? JACK
NAME ? DAVID
NAME ? MARTIN
JACK DAVID MARTIN

List the program and read it carefully. Follow the two loops as they cycle through the computer and consider the values in I and P as they do so. Then, try again. It may take you some time to become completely familiar with the structure of combining an array with a FOR . . . NEXT loop as demonstrated here. Just remember that you must first give the array a dimension that should correspond to the full count of the loop.

In theory, at least, arrays are terrifically powerful tools. They allow you to set up huge numbers of variables that are given logical sequence by their ascending numbers. And each is flawlessly established by the computer so that you can't make typing errors as you go. Best of all, they are quick and easy to dimension so that you don't have to spend hours programming variables into your computer.

But the key to constructing arrays, or at least their major advantage over just about any other method of setting up a pattern of related variables, is the sheer orderliness of the things. A two-dimensional array is very much like the military's finest drill team. Each step from variable to variable is exactly the same. Each stride has precisely the same reach. And the drill sergeant, the DIM statement, knows absolutely where everyone is and how many there are in his group.

Thus, an array is the ideal way of categorizing a series of numbers or strings. As long as all the data in question shares some trait, then an array can organize and file all the data. No single bit of information is stored with another. Each is given its own variable. But the array as a whole makes working with the group much easier. You know in an instant that names stored in a particular array have something in common. And you can both store and recall the names with ease principally because you know exactly where they are. So think of each array as establishing some sort of category, a classification of data into a complete set that is stored in like manner for the sake of accuracy, ease and accessibility.

This system of classification may not make sense to you as yet. But you'll soon see how DIM allows you to execute an organizational plan for your computer's memory.

Since we're trying to convince the 64 to ingest all the names of all our immediate relatives, perhaps the program to do so should be enlarged. At the same time, it might be pleasant if the program asked for a first, middle and last name. However, as soon as this concept is introduced, the nature of the

array in question is changed. Where once the program established a single row of variables, each A\$ with a different number after it, now we are dealing with three categories of variables. And, worse, there may be twenty or so names to put in each category. One solution to the problem might be to set up three different arrays:

```
NEW
5 PRINT CHR$(147)
10 DIM A$(20)
20 DIM B$(20)
30 DIM C$(20)
40 FOR I=1 TO 20
50 INPUT "FIRST NAME";A$(I)
60 INPUT "MIDDLE NAME";B$(I)
70 INPUT "LAST NAME";C$(I)
80 NEXT I
90 FOR I=1 TO 20
100 PRINT A$(I);" ";B$(I);" ";C$(I)
110 NEXT I
```

This program establishes three lists of names. All of the first names it is to absorb go into the array called A\$. All the middle names go into B\$ and all the last names into C\$. At the same time, the program identifies each person involved by numbering the variables in each of the arrays. That is, all three parts of a person's name are entered inside the loop described on line 40. Thus, the value of I is the same for all three parts of any one individual's name.

At the end of the program, when we wish to print all the names in question, we do so by calling them up based on their proper order (A\$ then B\$ then C\$) and use the I value (or the index number of each person) to keep everything organized.

Pause for a moment to consider the landscape of the locations in memory that we are creating. Imagine a column of variables known as A\$, each one labeled as to its column and the number or position it occupies in that column. This is a two-dimensional array. It's like a single-file line of marching soldiers, each with his dog tag to identify him.

Now, right next to this column of variables, picture another column known as B\$ and then a third one known as C\$. Each column has twenty soldiers. Finally, in a fit of whimsy, paint first names on the helmets of all the soldiers in column A\$, middle names on the helmets of soldiers in B\$ and last names on the helmets of soldiers in C\$. The resulting group of arrays looks like this:

A\$(1) Bob	B\$(1) William	C\$(1) Jones
A\$(2) Alice	B\$(2) Locke	C\$(2) MaGee
A\$(3) Ted	B\$(3) Edmond	C\$(3) Richards
A\$(4) John	B\$(4) Paul	C\$(4) Getty
A\$(5) Peter	B\$(5) Wadsworth	C\$(5) Stat
A\$(6) George	B\$(6) Allan	C\$(6) Carson
A\$(7) Ken	B\$(7) Geoff	C\$(7) Hicks
A\$(8) Susan	B\$(8) June	C\$(8) Brown
A\$(9) Merle	B\$(9) 'The Pearl'	C\$(9) Smith
A\$(10) Betty	B\$(10) Gravely	C\$(10) Lewis
A\$(11) Liz	B\$(11) Burton	C\$(11) Morrow
A\$(12) Cheryl	B\$(12) Lea	C\$(12) Vitali
A\$(13) Kristin	B\$(13) Jennifer	C\$(13) Wilson
A\$(14) Kevin	B\$(14) Samuel	C\$(14) Kale
A\$(15) Ralph	B\$(15) David	C\$(15) Longfellow
A\$(16) William	B\$(16) Jones	C\$(16) Kent
A\$(17) Carl	B\$(17) Bud	C\$(17) Bookbinder
A\$(18) Richard	B\$(18) Lewis	C\$(18) Stevenson
A\$(19) Ann	B\$(19) Marie	C\$(19) Kelley
A\$(20) Marie	B\$(20) Jessica	C\$(20) Johnson

Run the program, input names to your heart's content, and then watch the printing on the screen. As you can see, the computer keeps everyone in

his place, categorized with absolute precision.

The program that has thus far been developed works perfectly. If, for some reason, we need only the first and last names of our relatives, we simply amend the program to print only from array A\$ and array C\$.

But assume for the moment you would like to create a computer program that stored more than names. It's possible that phone numbers, birth dates, sex, addresses, all sorts of information might be useful. A list of birth dates, for instance, might be programmed to warn the unwary relative (you) of an impending birthday that ought not to go unnoticed. And the addresses might make the annual Christmas card writing marathon much easier.

This idea, while a good one, seems to call for a very large number of arrays. Each category of information requires its own. Worse, there are so many of them that there is a threat the program will develop the sort of confusion that arrays were supposed to eliminate. We're dealing with a considerable number of categories and a potentially large number of facts in each category. Yet again, we need an orderly system. But instead it appears that we are concocting something that grows in a capricious, haphazard way.

Happily, arrays once more come to the rescue. They do so by allowing you to construct two-dimensional arrays. That is, you can establish a single array and assign each individual variable in it two categories. In the case of our earlier program that stored the first, middle and last names of relatives in three arrays, we can now use a single array for the chore. All we have to do is add a second, qualifying bit to each variable. To explain, we'll begin by writing the program.

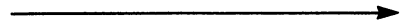
```
NEW
5 PRINT CHR$(147)
10 DIM A$(3,20)
20 FOR I=1 TO 20
30 INPUT "FIRST NAME";A$(1,I)
40 INPUT "MIDDLE NAME";A$(2,I)
50 INPUT "LAST NAME";A$(3,I)
60 NEXT I
70 FOR I=1 TO 20
80 FOR N=1 TO 3
90 PRINT A$(N,I);" ";
100 NEXT N
110 PRINT
120 NEXT I
```

Line 10 of this program establishes an array that is two-dimensional. But it is precisely the same, from a working standpoint, as the three linked arrays we produced before. Instead of three arrays, A\$, B\$ and C\$, each of which described twenty numbered variables, we now have a single array that has three major categories and twenty variables in each category. Hence,

A\$(1,I)

means array A\$, first category, example number I (where the variable I stands for numbers between one and twenty). And

A\$(2,I)

means array A\$, but the second category, example number I. In this way lines 30 through 50, in conjunction with the FOR . . . NEXT loop on lines 20 and 60, allow you to absorb twenty first names, twenty middle names and twenty last names. Each name is described as to its "category" or "kind" and is given a number that ties it to a certain individual. Pictured in the fashion of our earlier arrays, this one looks like this 

Lines 70 through 120 in this program create a print on the screen of all the names that have been input. They perform this feat using two nested loops. Since this is a bit tricky in and of itself, proceed slowly here. But note that the organization of the array that is established at the outset allows us a very, very economical way of recovering the names that have been typed into memory. The loops are simple and straightforward because the data we've stored has been tucked away in memory in an orderly way.

As you may recall, nested loops work from the inside out. That is, the loop in the middle, the one in the nest, recycles through its full count before the outer loop recycles a second time. Thus, the inner loop described by lines 80 and 100 involves the major categories of names: first, middle and last. This is because we wish to print each of these categories for every individual on the list.

The outer loop, on lines 70 and 120, counts the identifying number for each person on the list. This number stays constant as the program cycles through the categories, printing your aunt's full name. When the outer loop changes its count, the program is dealing with your cousin.

Line 90 does the actual printing. Here we first call on A\$(N,I), which will be A\$(1,1) at the outset. This is the first name of the first person on our list. After this moniker has been displayed on the screen, we call on the NEXT N, which now is 2, and so print A\$(2,1), or the middle name of the first person on the list. Finally, A\$(3,1) is printed. This, of course, is the last name of the first person on the list.

Note that the program also uses semicolons to keep the printing on the same line and it adds a space after each of the three parts of everyone's name. After cycling through the full three categories, the computer exits this loop on line 100 and goes to line 110. Here it skips a line to get ready for the next name. And the next name, as you may have concluded, is picked out of our list by ratcheting the counter, I, up one on line 120.

Inhale deeply, refer to the chart of our array, reread the program and then study the explanation of it for a bit. Take your time to contemplate thoroughly what is happening here. There's no hurry and this is a significant building block insofar as the material to follow is concerned.

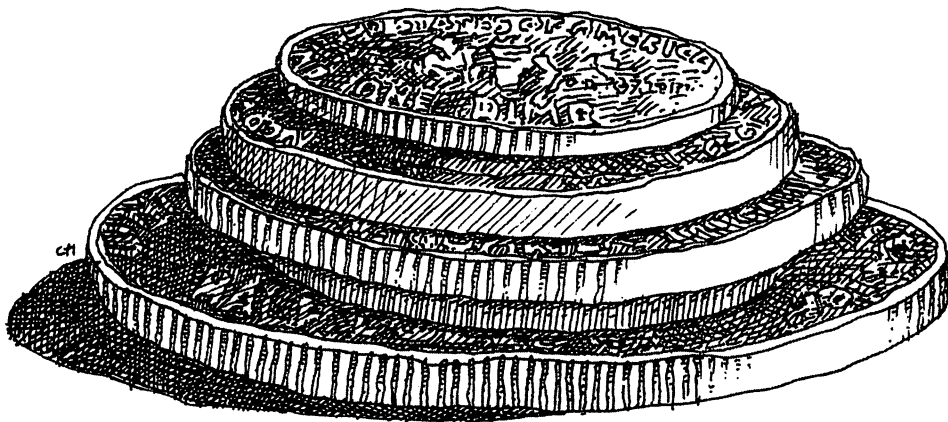
A\$(1,1) Bob	A\$(2,1) William	A\$(3,1) Jones
A\$(1,2) Alice	A\$(2,2) Locke	A\$(3,2) MaGee
A\$(1,3) Ted	A\$(2,3) Edmond	A\$(3,3) Richards
A\$(1,4) John	A\$(2,4) Paul	A\$(3,4) Getty
A\$(1,5) Peter	A\$(2,5) Wadsworth	A\$(3,5) Stat
A\$(1,6) George	A\$(2,6) Allan	A\$(3,6) Carson
A\$(1,7) Ken	A\$(2,7) Geoff	A\$(3,7) Hicks
A\$(1,8) Susan	A\$(2,8) June	A\$(3,8) Brown
A\$(1,9) Merle	A\$(2,9) 'The Pearl'	A\$(3,9) Smith
A\$(1,10) Betty	A\$(2,10) Gravely	A\$(3,10) Lewis
A\$(1,11) Liz	A\$(2,11) Burton	A\$(3,11) Morrow
A\$(1,12) Cheryl	A\$(2,12) Lea	A\$(3,12) Vitali
A\$(1,13) Kristin	A\$(2,13) Jennifer	A\$(3,13) Wilson
A\$(1,14) Kevin	A\$(2,14) Samuel	A\$(3,14) Kale
A\$(1,15) Ralph	A\$(2,15) David	A\$(3,15) Longfellow
A\$(1,16) William	A\$(2,16) Jones	A\$(3,16) Kent
A\$(1,17) Carl	A\$(2,17) Bud	A\$(3,17) Bookbinder
A\$(1,18) Richard	A\$(2,18) Lewis	A\$(3,18) Stevenson
A\$(1,19) Ann	A\$(2,19) Marie	A\$(3,19) Kelley
A\$(1,20) Marie	A\$(2,20) Jessica	A\$(3,20) Johnson

Arrays clearly offer an efficient programming tool for storing large amounts of information. They let us pigeonhole data in a sensible way and they don't require cumbersome programming techniques or huge amounts of computer memory. But having stored information, we need to investigate ways in which arrays can be manipulated. After all, such is the purpose of the machine.

For the time being, leave the fledgling electronic address book behind. Instead, let's try to work with some numbers.

The human mind operates in delightfully mysterious ways. It registers information using a wide variety of visual skills, past experience and intuitive leaps. As a result, many of the things we do every day occur unconsciously, depending on years of experience and little in the way of carefully planned procedure. Computers, on the other hand, are incapable of this behavior. They must proceed along strictly logical paths, making objective comparisons and responding in methodical ways. Thus, the computer programmer's most interesting challenge is to break a relatively obvious action down until it is completely understood. Then and only then can he teach a computer how to do even the simplest things.

For instance, consider the act of taking a pile of coins and arranging them so that the largest is on the bottom and the smallest on top. To begin, the pile looks like this:



In a glance you can tell that the coins are almost as they should be. All you have to do is move up the coin that is next to the bottom until it is next to the top. In another instance, the same visual and logical action takes place: sort

the following numbers so that the smallest is at the top and the largest at the bottom and their values decline in order:

6
100
3
5678
10987

No trick to that. You respond: "3, 6, 100, 5678, 10987." But whether you know it or not, you have sorted these numbers by first considering them as objects. That is, you looked first at their relative bulk. Discovering that the two bottommost numbers were the largest and in order, you then addressed the top three numbers, fixing them by considering their value as well as their bulk.

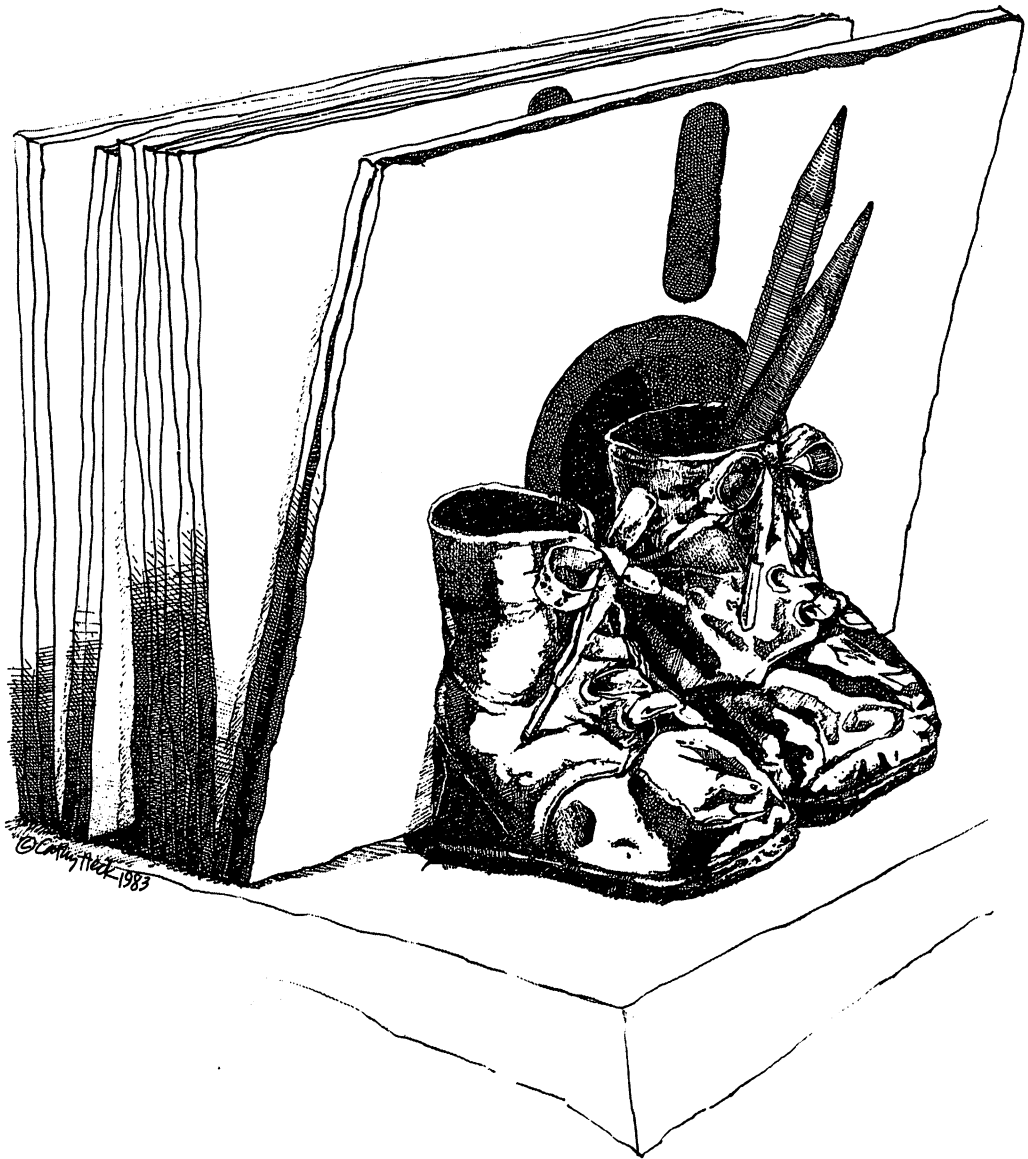
Computers, alas, can't see. So unless you program some kind of "visual" check into your computer, it won't avail itself of this aid. Therefore, computers have to be told to compare everything to everything else in a very quantitative and exacting way before even the simplest form of sorting or reordering can be done.

It should come as no surprise to you that computers are asked to do this kind of work all the time. They sort zip codes, put names in alphabetical order, perform remarkable financial calculations based on the relative proportions of various assets and otherwise compare things constantly. How they do so, however, is not necessarily simple. Among five numbers there is a minimum of ten unique relationships, one number to another. Sorting them involves the comparison and analysis of each relationship.

One classic method of sorting numbers is to employ this one-on-one comparison activity from the bottom up. That is, you compare the number that has its location on the bottom of the list with the number just above it. If the number on the bottom is smaller than the one immediately above it, you move the smaller number up and the larger one down. The idea thereafter is to follow each number as it moves upward, each to its ultimate, highest point. Then the next number is compared to all those above it until it too rises as far as it can. When all the numbers have been elevated on the list to their highest value, they are in order.

This method of sorting is called a "bubble sort" because numbers rise in a list based on their value. Small numbers, like bubbles, rise. Bubble sorting is a painstaking way of putting things in order. Many, many comparisons are made in moving only a few numbers around. But the individual action of switching the position on a list of two compared numbers is easy to understand. If one number is greater than another, it belongs lower on the list. Therefore, they should exchange positions.

What follows is a short program to put five numbers in order. This is accomplished here in reverse of the bubble sort. That is, comparisons begin at the top of the list and move larger numbers down rather than smaller ones up. The distinction is made in this book because the program is easier to



You have to learn to walk before you can RUN.

understand. Working from the bottom up requires FOR . . . NEXT loops that count backwards (STEP -1) and this seems an unnecessary problem at this stage.

To begin, of course, we have to ask whoever is sitting at the computer to give us five numbers. So, let's do so:

```
NEW
5 PRINT CHR$(147)
10 REM TAKE THE NUMBERS
20 PRINT "FIVE NUMBERS TO SORT"
30 DIM N(5)
40 FOR I=1 TO 5
50 INPUT N(I)
60 NEXT I
```

At this point the program has absorbed the numbers in question and has deposited them in an array called N that has five discrete variables: N(1), N(2), N(3), N(4) and N(5).

This accomplished, the strategy is to compare every position on the list with every other number in the list until the computer gets the very lowest valued number to the top (i.e., in variable A(1)). Put differently, the program is going to check every number on the list against the number at the top. If the number at the top of the list is larger than the number we're checking, the smaller number will replace the larger. In this way, when all four comparisons have been made, the smallest number will be at the top. Thereafter, we'll go to the next position on the list, the second spot, and compare the number in it with all the numbers beneath it on the list.

The program that performs this tedium is relatively short because we know where all the numbers are: in the array. And we know what the variable names are. So we need only set up a pair of nested loops to make the comparisons for us over and over. Here they are:

```
70 REM COMPARE POSITION
80 FOR I=1 TO 4
90 ACCUM=N(I)
100 REM OTHER POSITION
110 FOR P=I+1 TO 5
120 IF N(P)<N(I) THEN N(I)=N(P):N(P)=
    ACCUM:ACCUM=N(I)
130 NEXT P
140 NEXT I
```

To begin, line 80 states that the process is going to do something to each of the upper positions on the list. We're working only with the top four because, when each of these has been compared to those below it, the work will be done. After all, the last position on the list has none beneath it, so no comparisons have to be made. The crucial concept here is that the program is going to use the index number in the variable to position each variable in the

array. Thus, N(1) is the topmost position on the list. It will be compared to all the others first. Thereafter, the computer will compare each successive position with all those beneath it.

On line 90 we establish an accumulator variable, labeled ACCUM here for your convenience, to hold the value that may be moved somewhere. That is, we put the value that is in the position being considered into an accumulator.

On line 110 the positions being compared to the original position are defined. Since we begin with position number one, the counter is set to begin at position I+1 or the position A(2). This counter will loop through all the positions, offering each in turn for comparison with whatever resides in position A(1).

Line 120 isn't as intimidating as it looks. Here the program compares N(P), the compared position, with N(I), the comparing position. If the compared position is less than the number in the comparing position, A(1) the first time, then we switch the two numbers. This is done by redefining N(I) to equal N(P). Then we dredge the original value of N(I) out of the accumulator and deposit it in N(P). The switch having been made, we redefine the accumulator so that it now holds the new number that is in the position in question.

Line 130 recycles the program to compare the original position, A(1), with the next to be considered, A(3). When the first position has been compared to all the others by the loop bounded on lines 110 and 130, then line 140 counts a new value for I, thus comparing the second position to all those below it.

To be sure, you are scratching your head. But walk through the program with a pad and pencil. Pick five numbers, label them as they are labeled in the array and then see what happens as the two loops cycle and recycle. If before you do so you wish for some assurance that the process works, simply add the following to print the results of the program and try running it.

```
150 FOR I=1 TO 5
160 PRINT N(I)
170 NEXT I
```

The program in question demonstrates two important characteristics of programs. First, they operate with remarkable speed and so can perform a task in a relatively awkward way without sacrificing too much time. And, second, they must be strictly logical in order to run. Regardless of the manner in which you might solve a problem, the computer must address it in small, absolutely individual steps.

To show you the steps made by the computer in sorting five numbers, we've picked five and worked through the rearranging process step by step. The table on the facing page demonstrates the activity of line 120 in the program. It compares numbers to determine which is greater and then switches the values when the smaller number occupies a lower position on the list.

		<i>Compare</i>	<i>React</i>	<i>Compare</i>	<i>React</i>	<i>Compare</i>	<i>React</i>	<i>Compare</i>	<i>React</i>
A(1)	23	23	23	6	6	6	6	3	
A(2)	89	89	89	89	89	89	89	89	
A(3)	6	6	6	23	23	23	23	23	
A(4)	26	26	26	26	26	26	26	26	
A(5)	3	3	3	3	3	3	3	3	6

First count of variable I completed here. Lowest number in variable A(1).

A(1)	3	3	3	3	3	3
A(2)	89	23	23	23	23	6
A(3)	23	89	89	89	89	89
A(4)	26	26	26	26	26	26
A(5)	6	6	6	6	6	23

Second count of variable I completed here. Variable A(2) corrected.

A(1)	3	3	3	3
A(2)	6	6	6	6
A(3)	89	26	26	23
A(4)	26	89	89	89
A(5)	23	23	23	26

Third count of variable I completed here. Variable A(3) adjusted.

A(1)	3	3
A(2)	6	6
A(3)	23	23
A(4)	89	26
A(5)	26	89

Fourth and final count of variable I completed. Variables A(4) and A(5) corrected.

Bigger Is Better

Comparing five numbers isn't a particularly awesome task. But among the beauties of programming is the ability to expand on a job well done until it becomes truly useful. What's more, the use of arrays makes this enlargement of a program an extremely simple task. This is the case because an array can be any size that your computer's memory can absorb. Even in the case of a small computer like the 64, this amounts to a considerable quantity of information.

Arrays are enlarged by writing a program that doesn't limit them. That is, we establish the proportions of the array using variables and then query the keyboarder as to how much information he thinks he has.

To amend our FIVE NUMBERS TO SORT program so that it will sort many more numbers, you need only type the following changes:

```
20 INPUT "HOW MANY NUMBERS ARE THERE";X
30 DIM N(X)
40 FOR I=1 TO X
80 FOR I=1 TO X-1
110 FOR P=I+1 TO X
150 FOR I=1 TO X
```

The above alteration in the program is quite straightforward. Essentially, all you've done is substitute the variable X for all the values of five in the program. Since the total number of numbers to be sorted controls the number of comparisons and exchanges that are made, using a variable allows you to increase the capacity of the program. All you have to know is the total number of numbers involved at the outset.

List the program and retrace how it works. Then run the thing. Use negative numbers, figures with decimals and try using the same number three or four times.

At this juncture we can turn our concentration back to the address book program we worked on earlier. It should be fairly obvious that no electronic storage facility for names and addresses, much less numbers such as zip codes, phone numbers and birth dates, should be a mere record. The program ought to be able to manipulate the data it contains and offer useful selections.

To manage this we need to be able to work with strings in the same way we have just worked with numbers. The computer should be programmed to make string comparisons and draw conclusions from those comparisons. One such activity is quite obvious: putting words in alphabetical order.

From our earlier exercises with the ASCII and CHR\$ codes, you can probably guess some of the techniques in arranging words alphabetically with a computer. In fact, the program that performs this minor miracle contains many of the approaches used in putting numbers in order. This is the case because the ASCII codes have been assigned code numbers in order through

the alphabet. Making comparisons of the ASCII value of a letter with another letter's ASCII value reduces the problem to a mathematical exercise for the computer. And computers, as you know by now, thrive on such efforts.

The problem with words, however, is that their value isn't particularly determined by their length. The expression, "Now that was a twenty-five cent word," is extremely figurative. And has no useful value in putting things in order, at least not alphabetical order. So, to begin, we'll put some letters in order. The program is presented here in its entirety because it is virtually identical to the one we concocted earlier. Only a few of the lines in it are different.

```
NEW
  5 PRINT CHR$(147)
 10 REM TAKE THE LETTERS
 20 PRINT "FIVE LETTERS TO SORT"
 30 DIM L$(5)
 40 FOR I=1 TO 5
 50 INPUT L$(I)
 60 NEXT I
 70 REM COMPARE POSITION
 80 FOR I=1 TO 4
 90 ACCUM$=L$(I)
100 REM OTHER POSITION
110 FOR P=I+1 TO 5
120 IF ASC(L$(P))<ASC(L$(I)) THEN L$(I)=
    L$(P):L$(P)=ACCUM$:ACCUM$=L$(I)
130 NEXT P
140 NEXT I
150 FOR I=1 TO 5
160 PRINT L$(I)
170 NEXT I
```

In terms of naming variables, the only alteration in this program from the other we dealt with is to call the letters to be ordered L\$ (for letters) rather than N (for numbers). And, of course, the accumulator has been renamed ACCUM\$ so that it becomes a string variable, capable of accepting strings.

The only other change of any substance is to compare the ASCII code value of the letters being rearranged. This is accomplished by converting the letter, a string, into a number. This transformation is carried off in the same way we checked for bogus typing in our program that managed keyboard inputs. As you can see on line 120, we compare the ASCII value of L\$(P) to the ASCII value of L\$(I). It's as simple as that.

After having run the program a few times you may wonder why words present such a problem for the computer. To find out, type:

```
PRINT ASC("TROUBLE")
```

and you'll find that the value of the word is

84

Now type:

```
PRINT ASC("TABLE")
```

and once again you discover that the value is

84

This apparent myopia on the part of the 64 results from the function of the ASC in the PRINT statement. ASC delivers the value of the first letter or character inside the parentheses. It ignores all the others that follow. So the program puts letters in alphabetical order and puts words in alphabetical order so long as each starts with a different letter.

Solving this limitation requires a chapter unto itself.

CHAPTER 13

The authors of this book have gone to considerable lengths to ensure that this concluding chapter would be the thirteenth. For we were determined at the outset to present you with a concrete omen concerning that which ends your studies here. So far, we have investigated BASIC commands in short programs and program fragments. And we have built small or relatively small programs to watch the computer perform discrete tasks. This, of course, was by design. It's considerably easier to understand a new idea if it isn't surrounded by a throng of only slightly familiar ones. So we've proceeded in small steps, hoping you could master each in its turn.

However, you have come to the point in this text wherein you will learn how to assemble in a single program the better part of everything taught thus far. That is, we're going to try to put all the parts together into something useful. The particular task at hand isn't important. We've simply selected one that is typical of the work you can coax from your 64. In design and structure, in organization and planning, it is very much like most of the programming you will ever do. Happily, when it is complete, you'll have at least one of two things: you will possess a firm grasp of programming in BASIC and/or you will have a decent address book.

Of course, what follows is potentially confusing (i.e., you are certain to be regularly bewildered). But don't let the size of the program intimidate you. It is written in altogether functional chunks, each clearly separated from the others. In trying to understand what's going on, think of only one portion of the program at a time. Consider it a miniprogram unto itself. Trace what goes into a section in terms of data and watch what is done with that data. Each

manipulation is logical, each BASIC command simple and to the point. It's their interaction that becomes complex. So take it all one relationship at a time until, eventually, you see how the tens of thousands of actions this program performs are really very simple. The wonder is that the thing is so fast and accurate, not that it is all that clever.

To begin, back up. The program to follow can do a number of different things. We've investigated this versatility in our calculator program. To select certain kinds of arithmetic operations, the keyboarder was given a series of options. He could hit any of the following keys, +, -, *, /, to direct the computer to do a number of different things. Because we were dealing with an exercise at the time, the program never bothered to explain these operations. You know them and so do most other people. But at this stage, the program is going to offer the person at the keyboard options that are not obvious. So, it must explain them. In so doing we follow the traditional, though not universal, method of offering options. It is a method derived from the culinary world. It's called a menu.

Menus

A menu, in a restaurant, is a list of dishes that one can request from the kitchen. Such menus were originally produced so that a diner would know what foods, among the endless number of possible foods, were specifically available in a particular establishment. Thus, menus serve to tell a hungry person what he can eat and also, by implication, what he can't eat.

Computer menus work the same way. They present you with a list of things that the program can do and allow you to select one. An extremely simple menu appears below:

```
NEW
5 PRINT CHR$(147)
10 PRINT "1. PRINTS JOHN"
20 PRINT "2. PRINTS PAUL"
30 PRINT "3. PRINTS GEORGE"
40 PRINT "4. PRINTS RINGO"
50 INPUT "PLEASE SELECT";A
60 IF A=1 THEN PRINT "JOHN"
70 IF A=2 THEN PRINT "PAUL"
80 IF A=3 THEN PRINT "GEORGE"
90 IF A=4 THEN PRINT "RINGO"
```

Running this program will cause the following to appear on the screen:

```
1. PRINTS JOHN
2. PRINTS PAUL
3. PRINTS GEORGE
4. PRINTS RINGO
PLEASE SELECT ?
```

If you type 1, 2, 3 or 4, then one of the four names is printed on the screen. If you type any other number, none of the conditions on lines 60 through 90 will have been met and nothing will be printed.

This structure is the simplest form of menu. However, it does present the possibility that the keyboarder can make choices about what he wants the computer to do. Instead of line 60 reading

```
60 IF A=1 THEN PRINT "JOHN"
```

we can replace the prescribed action with another, more complex action. A particularly versatile choice is to direct the computer using GOTO. In this way, we can construct a relatively complex group of actions in a program and allow the person sitting in front of your 64 to select from among any of them. Thus, the conditional statements in a menu often look like this:

```
60 IF A=1 THEN 1000
```

Which reads, "If option 1 of the menu has been chosen, then go to the area of the program that begins on line 1000." Heaven knows, there is a wide variety of things that can be done commencing on line 1000. And there are always lines 2000, 3000, etc., for an equally staggering number of actions.

The program we are about to construct will open with a menu. The potential processes described there will list all the things the program can do to the names and addresses in an address book. Better yet, the program will include mini-menus. At the end of any specific function we'll offer a simpler menu that allows the keyboarder yet another choice.

For instance, after having added a name to our list of friends and relatives we may 1) wish to add another name, or 2) want to return to the "main menu" that starts the program. The mini-menu will then offer this option and will contain further GOTO commands to direct the computer exactly where it has been asked to go.

In doing so we will build what is known as a "conditional, branching" program. One that, like a tree, branches in several directions. Unlike a tree, however, ours will offer the option to go back to the root of the matter, the main menu, and start over.

Working With Words

As was discovered in our program that put five words in alphabetical order, strings are inherently more difficult to work with than numbers. BASIC understands, at a glance, the full value of the number 54721. But the ASCII value of the word GENUINE is determined only by the letter G. The ASC command simply takes the first character in a string and converts it to its code number.

Thus, we have to find a way to inquire more deeply into the words that make up an address file. Not surprisingly, BASIC allows us to prod almost any string and inquire as to parts of it. Once again, there are some strangely useful commands that permit this and which, in a deceptively simple way,

offer the programmer tremendous power.

The first such command, and the one we will use here, is LEFT\$. This command looks at a specific number of characters at the far left side of a string. Which means LEFT\$ will contemplate only a certain number of the first letters in a word. This ability to slice the beginning off a word is extremely useful. Unlike the ASC command, which takes the ASCII value of only the first letter in a string, LEFT\$ can look at and compare more than one letter. And unlike a direct comparison of the entire contents of two strings, as in

```
IF A$=B$ THEN . . .
```

all the characters in a variable need not be considered. The programmer can elect only a limited number of characters to be compared. The advantage here may escape you at the moment, but it will shortly become clear.

LEFT\$, like the RIGHT\$ discussed earlier, operates quite straightforwardly. All the program must do is specify the string in question and offer the number of characters to be extracted from it. Type:

```
PRINT LEFT$("GOODBYE",4)
```

and hit RETURN.

```
GOOD
```

appears on your screen. The computer has been requested to print the leftmost characters in the string, starting at the beginning and proceeding for four characters. Hence the 4 after the GOODBYE.

The format of LEFT\$ is convenient enough, once you get used to it. LEFT\$ always precedes the statement. Then in parentheses, the string (or the name of the variable in which the string resides) is given. A comma follows and then the number of characters to be pulled from the string is stated. That's all there is to it.

This command allows you to substitute the label of a string variable for the string itself. If GOODBYE were in variable A\$, the program need only have stated

```
PRINT LEFT$(A$,4)
```

What's more, you can use numeric variables instead of specific numbers in specifying what you wish to pull from the string in question.

These last two capacities are particularly crucial because they make it possible for you to establish any number of methods for examining an array of variables, each of which contains a string. In other words, you can set up loops to request that LEFT\$ peer into a whole array of strings, one variable at a time. Since you can then compare all the strings in an array with any other word, you can find out where something is simply by looking for it.

Pause one moment. We are now describing a very simple, even crude method of finding a needle in a haystack. The technique involves a description of the needle, in the case here, the spelling of the word. Then the computer

patiently examines every bit of hay, one straw at a time, until it discovers the needle. In order to establish a loop that will do this unearthly chore, all you need to know is how many pieces of hay there are in the haystack.

This concept of tedious comparison may seem wasteful to you. And, indeed, there are other, more complex methods of getting a computer to search and discover something. These techniques are quicker, more efficient and require much more complex programming. So, we'll ignore them. To demonstrate the method at hand, let's go in search of a needle:

```
NEW
  5 PRINT CHR$(147)
 10 DIM A$(5)
 20 FOR I=1 TO 5
 30 INPUT "DESCRIBE THE HAYSTACK";A$(I)
 40 NEXT I
```

At this point the program has created an array that is to have five variables in it to hold strings. Each, presumably, will constitute some part of our haystack. Now we must search for something.

```
50 INPUT "DESCRIBE THE NEEDLE";B$
60 L=LEN(B$)
70 FOR I=1 TO 5
80 IF LEFT$(A$(I),L)=B$ THEN PRINT "NEEDLE IN A$(");I;"
90 NEXT I
```

Here, on line 50, the keyboarder is asked to input a word that the computer should look for in the array. The word to be sought is deposited in variable B\$. Then, on line 60, LEN measures the number of characters in B\$. This action tells the computer how long the word to be sought actually is.

On line 70 a loop is established that contains the same number of counts as there are variables in the array. Which is to say, the program will check each variable in the array, one at a time.

Line 80 contains the specification for the search and comparison as well as instructions as to the computer's response to successfully discovering the needle. It reads, "If the leftmost letters in the string that is contained in the variable being examined on this cycle of the loop, in A\$(I), starting with the first letter in the string and proceeding to the Lth letter, are the same as the word in B\$, then print NEEDLE IN and then print the name of the variable in the array where the word 'needle' is stored."

This is somewhat unwieldy, so we'll scrutinize it a bit more slowly.

```
LEFT$(A$(I),L)
```

means that the program must examine each of the variables in the array in a certain manner. The method prescribed is to slice off letters from the string in each variable. The program sets this specification knowing that the string being sought is a total of L characters long. So if the string in the variable being examined is longer, there is no sense in examining the extra characters.

There can be no match.

Run the program and respond as follows:

```
DESCRIBE THE HAYSTACK? HAY
DESCRIBE THE HAYSTACK? GRASS
DESCRIBE THE HAYSTACK? NEEDLE
DESCRIBE THE HAYSTACK? MORE HAY
DESCRIBE THE HAYSTACK? DUST
DESCRIBE THE NEEDLE? NEEDLE
```

In a flash,

```
NEEDLE IN A$( 3 )
```

appears on your screen.

The computer has compared the specified length of the strings in all five array variables with the requested match. And of course, it has found the needle.

It is possible that you are wondering about the use of `LEN` and `LEFT$` for such an undemanding chore. One might have simply requested the computer to compare the entire contents of the variables in question with the string that was specified. In this way the conditional, testing line of the program might have read

```
IF A$(I)=B$ THEN PRINT "NEEDLE IN A$(I);"
```

Put differently, one could have searched for an exact match, comparing each of the haystack's components, in its entirety, to the entire string that had been described as missing. However, doing so presents something of a problem, especially when working with names and addresses. Make the following change in the program as it now exists:

```
30 INPUT "NAME";A$(I)
```

and then run the program. Imagine now that you are a long-time cohort of United States presidents. You have an address book that contains their names, addresses and personal preferences for dinner. First, fill in the array:

```
NAME? JOHN
NAME? LYNDON
NAME? RICHARD
NAME? GERALD
NAME? RONALD
```

Now, further imagine that you are in search of Ronald Reagan's phone number. Thus, you decide you need to know in which variable in your array this particular needle is to be found. But since good ol' Ron is a friend, you inadvertently respond as follows:

```
DESCRIBE THE NEEDLE? RON
```


and hit RETURN. Please note that the computer successfully located the variable in which Ronald resides. The needle, you are told, is in A\$(5). This occurred because the program compared the requested string, RON, with only the first three letters in all the names in the array. Should your address book contain a number of Peters, Richards, Ronalds or other, similar names, this truncating capacity will prove useful. Similarly, if you are in search of the correct spelling of someone's last name, unsure of its exact configuration, you can search for Ruffner regardless of whether you recall if there are two fs in it or only one. Simply search for RUF.

At this juncture you should know that BASIC can pry words apart, looking at their middles with MID\$. However, if you are curious about this command, look it up in your Commodore manual. Discussing MID\$ here would only confuse the issue.

It appears to your authors that it is entirely possible that the readers of this book have reached a threshold: you know just enough about programming to be dangerous. You understand certain peculiarities of the BASIC language, have worked with a wide variety of data and can store and retrieve information almost at will. What's more, you can manipulate both words and numbers in many ways.

To be sure, you haven't been taught all that you need to know to make a living designing software. This book has only scratched the surface of programming. But the kinds of logic, the general structure, the overall approach you must assume when dealing with a computer have been covered. You've more than gotten your feet wet.

It is time, therefore, for a combination of skills just to see if the computer can be coerced into some practical work. So the 64 will now, with a little help, become a nifty little address book.

The Address Book

To commence with an outline of the task ahead, the first lines of programming offered here establish the main menu. That is, the first thing that will appear on the screen when you run this program will be a list of choices from which the keyboarder may select.

```
NEW
100 PRINT TAB(10) "1. ADDITIONS":PRINT
110 PRINT TAB(10) "2. DELETE NAME":PRINT
120 PRINT TAB(10) "3. SEARCH FOR NAME":PRINT
130 PRINT TAB(10) "4. DISPLAY FILE":PRINT
140 PRINT TAB(10) "5. SAVE FILE-EXIT":PRINT
150 PRINT:PRINT:PRINT
160 PRINT TAB(10) "PLEASE SELECT";
170 INPUT S
```

Since we'll be coming back to this menu over and over again during the use of the program, it should appear on the screen in and of itself, with nothing else confusing the view. So, type:

```
90 PRINT CHR$(147)
```

Perhaps the only unsettling element of the program thus far is the use of the command PRINT. It appears, to no apparent purpose, at the end of every line and, even more odd, appears three times in a row on line 150. The purpose of these commands is to unclutter the screen by double-spacing everything on it. Line 100, for instance, reads, "Print ten blank characters, then print the words 1. ADDITIONS on the screen and then, on the next line down, print nothing at all." Line 150 simply skips three lines so that the selection prompt is amply separated from the menu itself.

Having gathered the keyboarder's choice into variable S, the program must evaluate his selection. Considering the unknown program to follow, it might be wise to direct each choice into sizable areas. That is, it is prudent to assume more programming space will be required than seems plausible. So we'll separate the chunks of program that will run off this menu by healthy margins. Type:

```
180 IF S=1 THEN 400
190 IF S=2 THEN 600
200 IF S=3 THEN 900
210 IF S=4 THEN 1100
220 IF S=5 THEN 1200
230 GOTO 90
```

This accomplished, the data to be gathered and stored has to be collected. In this program, all of the information is absorbed by the computer using the INPUT command. As you know, subroutines can be developed to check the typing of such inputs using GET. But this safeguard isn't included here.

Clearly, we wish to gather a considerable amount of information. Thus, an array seems called for. To make it an easily remembered array, label it A\$. Furthermore, since your authors know how many categories are to be covered, trust us when we say that there will be eight. And as a further act of faith, be convinced that the 64 is comfortable handling one hundred names and their associated information without having to be expanded by the addition of more-than-standard memory.

So, we have eight categories of data and one hundred examples of each category. Before fleshing out this array, DIM it.

```
80 DIM A$(8,100)
```

As you can see, this is a two-dimensional array that can handle eight columns of data, one hundred rows deep. The eight categories are relatively easy to keep track of. Each will be numbered, from 1 to 8, so that searches and retrievals can be specified in the group of variables manipulated. But the

number of people on our list is rather large. So, as we take each in, the computer must assign the person a number that will forever be affixed to his or her name. For this work we'll need a simple counter. But it should follow a clearing of the screen before any inputs are offered:

```
400 PRINT CHR$(147)
```

Then the counter must be set up. Because the first person on our list should be numbered 1, the counter should start at 1 and accumulate single counts with each person described. Type:

```
410 C=C+1
```

Thus, when the input screen is first selected from the menu, and before any other work is done, the counter, C, will equal 1.

Now, the first prompt:

```
420 INPUT "FIRST NAME";A$(1,C):PRINT
```

This line indicates that the first question to be asked the keyboarder is the first name of the person being recorded. This string will then be deposited in A\$(1,C), which is the variable in array A\$, first category, person number C. The count of C will be controlled by line 410, increasing by 1 with every person described.

So that you can see the structure of the input section of this program all at once, it appears below in its entirety:

```
400 PRINT CHR$(147)
410 C=C+1
420 INPUT "FIRST NAME";A$(1,C):PRINT
430 INPUT "LAST NAME";A$(2,C):PRINT
440 INPUT "STREET";A$(3,C):PRINT
450 INPUT "CITY";A$(4,C):PRINT
460 INPUT "STATE";A$(5,C):PRINT
470 INPUT "ZIP";A$(6,C):PRINT
480 INPUT "PHONE";A$(7,C):PRINT
490 INPUT "BIRTHDATE";A$(8,C):PRINT
```

As you can see, there is nothing particularly unusual about this screen of INPUT prompts. One after another they will appear before the keyboarder, asking for information about a specific person. Each bit of information will then be deposited in a variable somewhere in array A\$. All the phone numbers, for instance, will fall into the category numbered 7 in this array. And a particular person's phone number can be located by finding his or her "count," variable C, in category number 7. Essentially, this program has created a grid with eight hundred squares in it. Each is labeled and each will contain a string.

Now, assume for the moment that you are sitting before the 64 and have just typed in your first entrant's essential statistics. You have come to the end of line 490 and have hit RETURN. Where ought the program to go? It might

be easy to send it back to the main menu. But doing so would make adding several names to the program a tedious process wherein the typist must select option number 1 over and over. So, let's put a mini-menu at the end of this screen.

```
500 PRINT
510 PRINT TAB(10) "HIT C TO CONT.":PRINT
520 PRINT TAB(10) "HIT M TO MENU":PRINT
525 Q$=""
530 PRINT TAB(10) "PLEASE SELECT";
535 INPUT Q$
540 IF Q$=CHR$(67) THEN 400
550 IF Q$=CHR$(77) THEN 90
560 GOTO 510
```

These lines of the program offer the keyboarder a choice. He may return to the main menu by striking M or he may add yet another soul to his list by striking C. Lines 510 and 520 announce his options. Lines 530 and 535 gather in his choice and deposit it in Q\$. Then line 540 inquires as to whether the contents of Q\$ is a C. If it is, the program is directed to line 400, which clears the screen, adds 1 to the counting variable and begins prompting for another name. Line 550 tests Q\$ to discover if an M has been put in it. If so, the computer returns to the clear screen line of the main menu. Line 560 is another error trap. If neither C nor M has been struck, the program asks again for a choice.

At this stage of its development the program will absorb all the data it was intended to absorb. However, should some friend become an enemy or otherwise disappear, it might be a good idea if he could be removed from our address book. This process presents two problems. First, the computer must locate the person in question. It must then display his file on the screen and ask if you are sure you want him eliminated. Then the program must erase his record and, most important, renumber all the identifying numbers in all the records. This is necessary because when we add to the file, replacing a banished soul, we will do so at the end of the list. So the hole created in the middle of the list by a deletion has to be closed up.

But, one thing at a time. First the program must locate the individual in question. The program should clear the screen and then ask for the name of the individual.

```
600 PRINT CHR$(147)
610 PRINT "NAME TO DELETE":PRINT:PRINT
620 INPUT "FIRST NAME";F$:PRINT
630 INPUT "LAST NAME";L$:PRINT
```

The program has cleared the screen (line 600), announced the purpose at hand (line 610) and then asked for the first and last names of the person to be deleted from the list (lines 620 and 630). These two pieces of information have been stored in F\$ and L\$.

As you know from our search for a needle in a haystack, the program now has to measure the length of the first and last names:

```
640 Y=LEN(F$)
650 Z=LEN(L$)
```

This information having been calculated, the fun begins. For the computer must now inquire into the contents of all the variables containing first names and attempt to match any one of those names with the string in F\$. Thereafter, a similar effort will be made searching all the last names.

Because you may have two or three Jennifers on your list of names, the match should require that both the first and last names match a single file. So, the IF statement should involve an AND, testing for both simultaneously. But before the search begins, the program needs a loop to scroll through the variables. Since the limit of one hundred names has been established, a loop that ends at one hundred efforts might be satisfactory. However, if you have only thirty-two names on your list, there is no sense checking the empty sixty-eight at the end. So the loop should be terminated with the last name. Thus, we use the counter variable, C, that is keeping track of how many entries have been made to date.

```
660 FOR I=1 TO C
```

should do the trick. Next, the search. The program has been established so that all the first names are in category number 1 and all the last names in category 2. So, we will search these categories:

```
670 IF LEFT$(A$(1,I),Y)=F$ AND LEFT$(A$(2,I),Z)=L$ THEN 700
```

This line of code checks the first and last names in any given row of the array (a row contains one individual's data), columns one and two for a match of F\$ and L\$. If it finds a match, the program directs the computer to line 700. Now to close the loop so that the computer will check everyone:

```
680 NEXT I
```

Suppose you requested a deletion and specified a name that didn't exist in your file. Where then should the computer be directed? The simplest answer, and the one to be used here, is the main menu.

```
690 GOTO 90
```

However, if a match has been located, the code on line 670 has sent the computer to line 700. Here we should display the contents of our located person's entire record. But if you ponder the issue for a moment, there is a high degree of probability that such a display will be called for in other parts of this program. Therefore, rather than write the code for the display here, it ought to be located elsewhere in the form of a subroutine. To keep this material out of the road, as it were, isolate it on line 2000.

```
700 GOSUB 2000
2000 PRINT CHR$(147)
```



GOSUB is not the cheer heard as a nuclear submarine clears port.

```

2010 FOR R=1 TO 8
2020 PRINT A$(R,I)
2030 NEXT R
2040 RETURN

```

This bit of programming is straightforward. It assumes that the value of I, the individual's code number, is the active value in memory when the subroutine is entered. This is the case because the moment that any programmed match is located, the count of variable I will remain constant, not adding to its count by going through the loop described on lines 660 and 680. All that needs to be done here is to display the eight categories of information about the person we've located. Hence the eight-count loop on lines 2010 and 2030.

At this point the person to be deleted should be staring the computer operator in the face. Clearly, the keyboarder should be given the option to evaporate the record or change his mind. So we do another mini-menu:

```

710 PRINT:PRINT
720 PRINT TAB(10) "HIT D TO DELETE":PRINT
730 PRINT TAB(10) "HIT M FOR MENU":PRINT
735 Q$=""
740 PRINT TAB(10) "PLEASE SELECT";
745 INPUT Q$
750 IF Q$=CHR$(77) THEN 90
760 IF Q$=CHR$(68) THEN 800
770 GOTO 720

```

Next, we must discover a way to erase this record. At the same time we wish to close in our list of records the hole that will be created by this deletion. Happily, this process can be performed simultaneously. As you know, a variable disappears its old contents when new contents are applied to it. All that needs to be done is move the next record on our file into the location occupied by the person we wish forgotten. And then, move the record thereafter into the newly vacated set of variables. The process looks something like this:

```

In A$(1,I)      ↗ WILLIAM → (Deleted)
In A$(1,I+1)    ↘ GEORGE
In A$(1,I+2)    ↘ RICHARD
In A$(1,I+3)    ↘ FRED

```

The strategy is to leave the variables as they are, but to move the contents of each variable to the one that precedes it. Which is to say, William will disappear as soon as we move George's record into variable A\$(1,I). The following program fragment accomplishes just this end:

```

800 FOR G=I TO C
810 FOR R=1 TO 8
820 A$(R,G)=A$(R,G+1)

```

```
830 NEXT R
840 NEXT G
850 C=C-1
860 GOTO 90
```

Here we begin counting the records to be altered with the match the program has located. That is, with record number I. And the purpose is to change all those that follow to the end, or to the count of C. Hence line 800. On line 810 we nest a loop that will control all eight categories. And then, on line 820 we copy the next record on the list into the variables that precede it. On line 850 the counter is reset to reflect the change (deletion) that has been made. When the corrections have all been made, we return to the main menu with a GOTO on line 860.

The process just described is passably complex. It involves the location of a name from among a list of names. And it employs a looped substitution process that moves a considerable amount of data around in an array. No single action is overwhelmingly difficult, but the entire process is potentially confusing. Try walking through the entire deletion process from the beginning once more. Take notes.

The third option offered in our main menu is the location of a particular name. At this stage of the program's development, you should find this task relatively simple. After all, the computer already accomplished this task once when it searched for a name to delete.

For the sake of variety, the code that appears below searches the program's records for a last name only. You could modify this to include the first name or program a search by first name and city, etc. But for the purpose of this program, last names should suffice.

There is a possibility, in fact not an unlikely one, that your personal files will include a number of relatives. As a result, there will be more than one individual stored in memory with the same last name. If the computer searches the files, selecting only based on last names, the first one it encounters will be delivered to the keyboarder. If this particular person happens to be the wrong "Smith," some method has to be developed to continue searching. The mini-menu at the bottom of this screen should contain a "keep going" option for instructing the computer to continue looking for Smiths.

With the exception of this option, what follows mirrors, in a simpler form, the search executed when we wished to delete a name. You should note that building two different methods of searching for a name, or any other bit of data in an array, is an excellent idea. It offers the forgetful keyboarder a number of methods of finding that which he can barely recall.

At any rate, here is the code for the program's search.

```
900 PRINT CHR$(147)
910 INPUT "ENTER LAST NAME";L$
920 Z=LEN(L$)
```



```

930 FOR I=1 TO C
940 IF LEFT$(A$(2,I),Z)=L$ THEN GOSUB 2000
950 NEXT I

```

In conjunction with the subroutine that displays the located file in its entirety on the screen, these few lines find the person specified. Unless that person is the wrong Smith. Multiple Smiths result in a blur of screens as each is found in turn. To force the computer to continue searching where it left off, insert the following:

```

945 IF LEFT$(A$(2,I),Z)=L$ THEN GOTO 960

```

This keeps the counter on line 950 from continuing its search if the first Smith has been located. Now, the options:

```

960 PRINT TAB(5) "HIT C TO CONTINUE SEARCH":PRINT
970 PRINT TAB(5) "HIT M FOR MAIN MENU":PRINT
980 Q$=""
990 PRINT TAB(5) "PLEASE SELECT";
995 INPUT Q$
1000 IF Q$=CHR$(67) THEN 950
1010 IF Q$=CHR$(77) THEN 90
1020 GOTO 960

```

When the computer has searched the entire file, displaying all the Smiths before you, it needs an exit. That is, when the count of variable I is completed and there are no more names to search through, the option to continue searching is irrelevant. So we must detour the computer around this option. It's simple and operates automatically:

```

955 GOTO 90

```

The program now has the capacity to add, delete and locate names. The fourth option offered in the main menu is the display of the entire list. Selecting this option will allow you to sit back and watch all those you care about scroll across the screen. However, unless some sort of delay is built into this function, the leisurely pace that you might desire will be lost and the names will zip by in a blur. The computer must be made to pause for a moment after displaying each file. Since this sort of pause has been covered in this text, you should recognize it.

```

1100 PRINT CHR$(147)
1110 FOR I=1 TO C
1120 FOR R=1 TO 8
1130 PRINT A$(R,I)
1140 NEXT R
1150 PRINT:PRINT
1160 FOR V=1 TO 1000

```

```
1170 NEXT V
1180 NEXT I
1190 GOTO 90
```

These nested loops should cause you little trouble. The only significant design curiosity here is that there are three of them. The outermost loop, lines 1110 and 1180, count the individuals on the list. The next loop spins through all eight of the categories for each individual. And following each person's display of vital statistics, the loop on 1160 and 1170 simply pauses to count to 1000.

Only one bit of work needs to be done to complete the data base (which is the computer word for a slew of information made accessible to a microprocessor). The program must develop some method of saving to a tape the information you have stored as well as the program that works with that information.

Saving the information that is stored in your array requires some BASIC commands not discussed in this book. We have assumed your ownership of nothing more than a Commodore 64 and a television. However, if you have programmed all the code for your address book and don't own a tape deck, it is likely you will shortly wish for one. If you are terrifically impatient, leave the computer on. Your authors will await your return.

The code that saves the data in your address book's array is below:

```
80 DIM A$(8,100)
1200 OPEN1,1,1, "DBFILE"
1210 PRINT#1,C
1220 FOR I=1 TO C
1230 FOR R=1 TO 8
1240 PRINT#1,A$(R,I)
1250 NEXT R
1255 PRINT#1, "*"
1260 NEXT I
1270 PRINT#1, "END DATA FILE"
1280 CLOSE 1
1290 END
```

When you opt to save the data in your array you must, of course, have a Commodore tape player plugged into your computer, but not turned on. Then all you need to do is elect option number 5 in the main menu and the computer will handle the rest of the work. It will "open" the file on tape called "DBFILE" (for data base file) and then transfer all the information in the array into that file. If changes in the address book have been made, they will all be included on the tape. Any old information, meaning the previously stored data, will be overprinted by the computer.

Very briefly, since this is an optional program section, line 1200 introduces a new BASIC command. OPEN is used whenever you want to store something from 64's memory on to a tape. And it is followed by the

notation 1,1,1, along with the name of the file. We have labeled ours DBFILE.

Line 1210 contains another odd notation: PRINT#1,C. The PRINT#1 is a BASIC instruction that commands the computer to PRINT to a tape rather than to the screen. The variable C, as always, notes the total number of names in the address book. Line 1220 sets up a FOR . . . NEXT loop that will count the people on the file. Line 1230 sets up yet another loop that will dial through all eight categories for each person listed.

When the computer arrives at line 1240, it will PRINT to the tape the first person on the list, with all his data. Thereafter, the loop described on line 1260 will ratchet the individual counter to the next person's file.

Line 1270 conveys the message END DATA FILE to the tape just so the computer is certain that the entire count has been executed. And CLOSE 1 indicates that the transmission is complete.

This much accomplished, it occurs to your authors that you may also wish to retrieve this information. So it seems only kind to add such an option to the main menu and offer the programming here. The drill, as it were, is to load the program from tape from your recorder by typing LOAD "ADDRESS BOOK" and then waiting for a while. When you are informed that the program itself is loaded, you run it and are greeted by the main menu. Thereafter, provided you have stored the data in question on tape, you will opt for selection number 6 from the menu. To add this option, type the following lines into your program:

```
145 PRINT TAB(10) "6. LOAD DATA":PRINT
225 IF S=6 THEN 1300
```

The menu has now been expanded. Next the LOAD DATA routine. In large measure, it mirrors the SAVE DATA option since it is constructed almost identically. If you wish further explanation of the BASIC commands used in either option, again, see your 64 manual.

```
1300 OPEN1,1,0, "DBFILE"
1310 I=1:R=1
1320 INPUT#1,C
1330 INPUT#1,A$(R,I)
1340 IF A$(R,I)="END DATA FILE" THEN GOTO 1390
1350 IF A$(R,I)="*" THEN A$(R,I)="":GOTO 1380
1360 R=R+1
1370 GOTO 1330
1380 R=1:I=I+1:GOTO 1330
1390 CLOSE 1
1400 A$(R,I)=" "
1410 GOTO 90
```

A Challenge

Having completed this rather time-consuming process, test the program. When you are presented with error messages on the screen, investigate the problem by listing the line of code that is reportedly flawed. Compare it to the complete listing of the program offered here. Be patient. Debugging, as the process is called, can take time.

After you've corrected any typing errors in your version of the address book, look it over carefully. And then make a list of those characteristics you think might be improved. By way of a combination of hints and challenges to your new-found programming skills, try to program these features into the address book as it now stands:

1. Instead of using INPUT to gather data, program a GET subroutine to do the job. Create one such subroutine to accept only letters, another to accept only numbers and a third to accept both (for street numbers and addresses).
2. Establish a purely numerical format for birthdays, such as 01-15-48, and use LEFT\$ to search this category of data for birthdays, looking for months.
3. Modify the program so that it will search for anyone with a given first name. Be sure to build a "continue searching" capacity into the change.

```
80 DIM A$(8,100)
90 PRINT CHR$(147)
100 PRINT TAB(10) "1. ADDITIONS":PRINT
110 PRINT TAB(10) "2. DELETE NAME":PRINT
120 PRINT TAB(10) "3. SEARCH FOR NAME":PRINT
130 PRINT TAB(10) "4. DISPLAY FILE":PRINT
140 PRINT TAB(10) "5. SAVE FILE-EXIT":PRINT
145 PRINT TAB(10) "6. LOAD DATA":PRINT
150 PRINT:PRINT:PRINT
160 PRINT TAB(10) "PLEASE SELECT";
170 INPUT S
180 IF S=1 THEN 400
190 IF S=2 THEN 600
200 IF S=3 THEN 900
210 IF S=4 THEN 1100
220 IF S=5 THEN 1200
225 IF S=6 THEN 1300
230 GOTO 90
400 PRINT CHR$(147)
410 C=C+1
420 INPUT "FIRST NAME";A$(1,C):PRINT
430 INPUT "LAST NAME";A$(2,C):PRINT
440 INPUT "STREET";A$(3,C):PRINT
450 INPUT "CITY";A$(4,C):PRINT
460 INPUT "STATE";A$(5,C):PRINT
```

```

470 INPUT "ZIP";A$(6,C):PRINT
480 INPUT "PHONE";A$(7,C):PRINT
490 INPUT "BIRTHDATE";A$(8,C):PRINT
500 PRINT
510 PRINT TAB(10) "HIT C TO CONT.":PRINT
520 PRINT TAB(10) "HIT M TO MENU":PRINT
525 Q$=""
530 PRINT TAB(10) "PLEASE SELECT";
535 INPUT Q$
540 IF Q$=CHR$(67) THEN 400
550 IF Q$=CHR$(77) THEN 90
560 GOTO 510
600 PRINT CHR$(147)
610 PRINT "NAME TO DELETE":PRINT:PRINT
620 INPUT "FIRST NAME";F$:PRINT
630 INPUT "LAST NAME";L$:PRINT
640 Y=LEN(F$)
650 Z=LEN(L$)
660 FOR I=1 TO C
670 IF LEFT$(A$(1,I),Y)=F$ AND LEFT$(A$(2,I),Z)=L$ THEN 700
680 NEXT I
690 GOTO 90
700 GOSUB 2000
710 PRINT:PRINT
720 PRINT TAB(10) "HIT D TO DELETE":PRINT
730 PRINT TAB(10) "HIT M FOR MENU":PRINT
735 Q$=""
740 PRINT TAB(10) "PLEASE SELECT";
745 INPUT Q$
750 IF Q$=CHR$(77) THEN 90
760 IF Q$=CHR$(68) THEN 800
770 GOTO 720
800 FOR G=I TO C
810 FOR R=1 TO 8
820 A$(R,G)=A$(R,G+1)
830 NEXT R
840 NEXT G
850 C=C-1
860 GOTO 90
900 PRINT CHR$(147)
910 INPUT "ENTER LAST NAME";L$
920 Z=LEN(L$)
930 FOR I=1 TO C
940 IF LEFT$(A$(2,I),Z)=L$ THEN GOSUB 2000
945 IF LEFT$(A$(2,I),Z)=L$ THEN GOTO 960
950 NEXT I
955 GOTO 90

```

```

960 PRINT TAB(5) "HIT C TO CONTINUE SEARCH":PRINT
970 PRINT TAB(5) "HIT M FOR MAIN MENU":PRINT
980 Q$=""
990 PRINT TAB(5) "PLEASE SELECT";
995 INPUT Q$
1000 IF Q$=CHR$(67) THEN 950
1010 IF Q$=CHR$(77) THEN 90
1020 GOTO 960
1100 PRINT CHR$(147)
1110 FOR I=1 TO C
1120 FOR R=1 TO 8
1130 PRINT A$(R,I)
1140 NEXT R
1150 PRINT:PRINT
1160 FOR V=1 TO 1000
1170 NEXT V
1180 NEXT I
1190 GOTO 90
1200 OPEN1,1,1, "DBFILE"
1210 PRINT#1,C
1220 FOR I=1 TO C
1230 FOR R=1 TO 8
1240 PRINT#1,A$(R,I)
1250 NEXT R
1255 PRINT#1, "*"
1260 NEXT I
1270 PRINT#1, "END DATA FILE"
1280 CLOSE 1
1290 END
1300 OPEN1,1,0, "DBFILE"
1310 I=1:R=1
1320 INPUT#1,C
1330 INPUT#1,A$(R,I)
1340 IF A$(R,I)="END DATA FILE" THEN GOTO 1390
1350 IF A$(R,I)="*" THEN A$(R,I)="":GOTO 1380
1360 R=R+1
1370 GOTO 1330
1380 R=1:I=I+1:GOTO 1330
1390 CLOSE 1
1400 A$(R,I)=""
1410 GOTO 90
2000 PRINT CHR$(147)
2010 FOR R=1 TO 8
2020 PRINT A$(R,I)
2030 NEXT R
2040 RETURN

```

A CONCLUSION

This book is intended to be an introduction. As was stated at the outset, it is meant to familiarize you with the way computers operate, at least insofar as the BASIC language is concerned. It is a book about a foreign tongue that can be mastered only by listening to it and speaking in it. Endlessly. But as an introduction to computing, it is a mere glossary of the language's most-used words. There are many others not mentioned here. And so this introductory text, by its very nature, must have a rather arbitrary conclusion. We didn't set out to compile a dictionary. What's more, in the style of this book, such a tome would be too heavy to lift. There are simply too many words in the BASIC language.

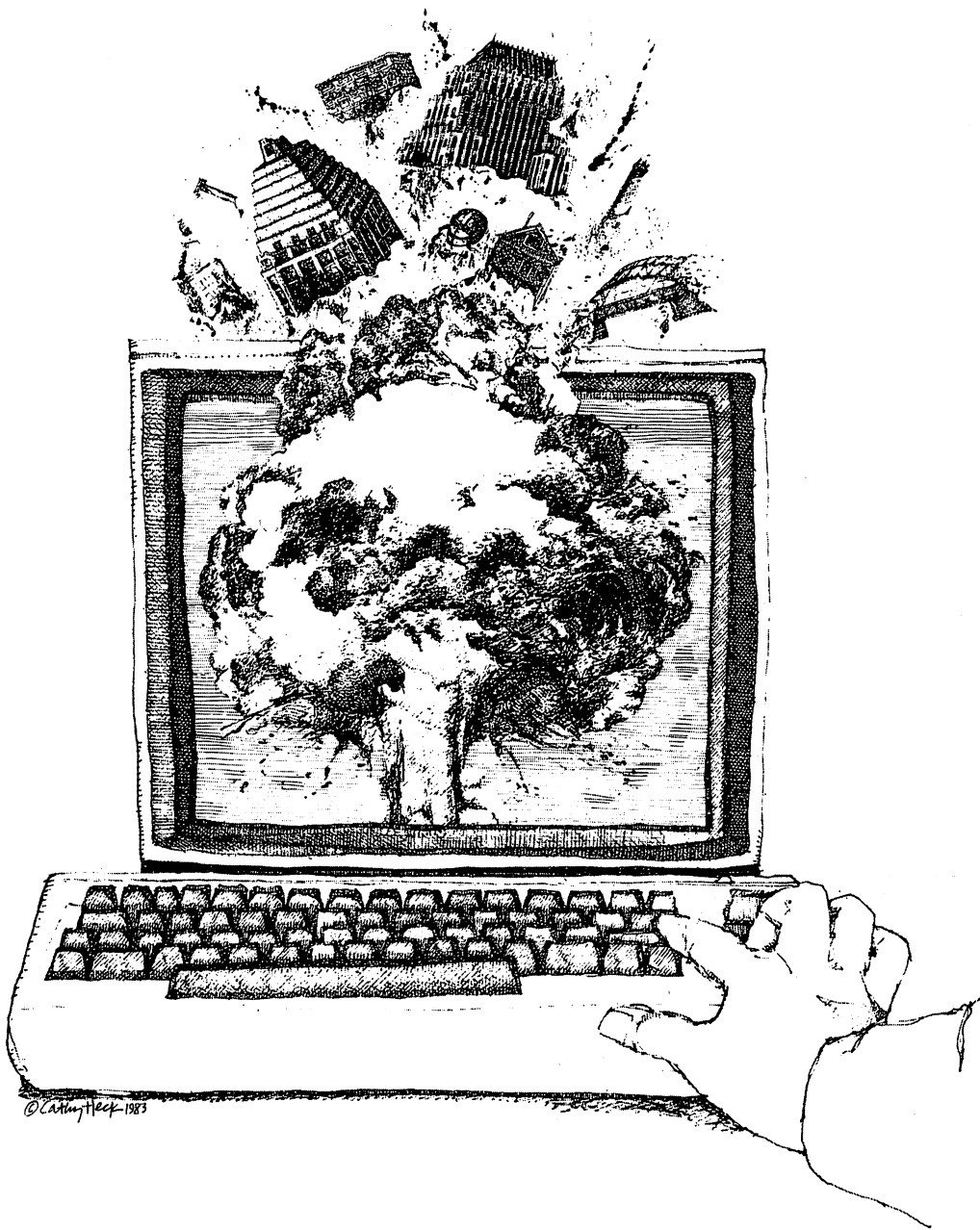
Whether you are aware of the fact or not, you have been exposed to little more than a mere dozen BASIC commands. Somewhat surprisingly, they have been coaxed into performing a fair number of remarkable actions. But the real power of your computer has only been hinted at here. Those commands never discussed, in conjunction with those you now know, are capable of all manner of nifty tricks. And the electronic sleight of hand they manage is great fun to observe and very satisfying to control.

Though the work your authors have performed is now complete, the fun that remains to be discovered is considerable. With a little luck, this book's unstated purpose, to intrigue the reader, to pique his interest and kindle his curiosity, has been accomplished. At this point, if you are fairly comfortable with BASIC's essential approach and more or less unintimidated by its methods, syntax and organization, then you can proceed to other areas of the language. You can peruse the more arcane parts of its vocabulary.

At the same time you will uncover something rarely suggested here: the design of programs. This concept is quite separate from their content. It involves the perspective of the programmer, his method of attack as well as the weapons he brings to bear on the battle. Certainly, you will discover that there are an almost unlimited number of ways to make a computer do any particular chore. And so two programs, each perfectly workable and unquestionably accurate, may be totally different from one another in spite of the fact they do the same thing.

And this business of "programming shape," the overall plan behind a program, is a challenging area to probe. To do so you'll have to read and critique other people's code. You'll have to break their work down and build it up again to discover what was intended and what was accomplished. This kind of study is fun, especially because you get to find the inevitable flaw in someone else's thinking. You can chuckle at the minor gaff in even the best of work.

Fortunately, in both the development of a larger vocabulary and in the effort to gain program design skills, you are offered a wealth of material. Books of all kinds crowd computer stores' and booksellers' shelves. Many are dry. Some are brilliant. But they all have code in them. And if you spend a little time with it, you'll undoubtedly discover you can actually read and understand the lines.



Fear not. An errant plunk at the return key is unlikely to blow up Pittsburgh.

AN INDEX

Accumulator 47, 84, 117
Addition 22
Array 158
ASCII 81
Assembly, Commodore 64 12ff

Boole 79

CHR\$ 56
Circles 59
Comma 52
Cursor 17

DATA 50ff
Delete 16, 70
DIM 158
Direct mode 41
Division 23
Division by zero 77, 85
Double-space 69

END 94
Error trap 86
Extra ignored 64

FOR. . .NEXT 99ff

GET 145ff
GOSUB 119ff
GOTO 89ff

IF 79ff
Immediate mode 41
INPUT 60ff
Integer 39, 102

LEFT\$ 180
LEN 136
LET 31ff
Line numbers 43
LIST 44
Loop 99ff, 111ff

Memo 99
Menu 178
MID\$ 183
Mismatch, type 25ff
Multiplication 22ff

Nanosecond 24
NEW 42
Next without For 115

ON. . .GOSUB 122
ON. . .GOTO 122
Out of Data 55

π 58
PRINT 22ff

Random numbers 103
READ/DATA 50ff
READY 14

Redo from Start 65
REM 152
Reserved words 33
Return 43
RND 103
RUN 44, 75
RUN/STOP 90

Scientific notation 104
Semicolon 52
STEP 100
STR\$ 128
Strings 31
Subtraction 23
Syntax error 16

TAB 135ff
Time, TI\$ 109ff
Type mismatch 25ff

Variables 28ff

ORDER BY MAIL . . . FIRST LOOK BOOKS

A three-book series written by J.M. Johnston and illustrated by Len Epstein, FIRST LOOK BOOKS introduce children to every aspect of computers. The tone is lively, conversational and informative, and new concepts are introduced by means of everyday examples. Practical applications of computer logic are considered in a simple context (boiling an egg) as well as through a more sophisticated approach (controlling heat in a solar home).

COMPUTERS: SIZES, SHAPES AND FLAVORS (First Look Book #1) examines the different types of computers, where they are used and how they work.

COMPUTERS: BEEPS, WHIRS AND BLINKING LIGHTS (First Look Book #2) takes you inside the heart of the computer and helps you understand the Central Processing Unit.

COMPUTERS: MENUS, LOOPS AND MICE (First Look Book #3) explains fundamentals of BASIC programming — loops, flow charts and syntax.

Please send me the following FIRST LOOK BOOKS:

COMPUTERS: SIZES, SHAPES AND FLAVORS _____ (No. of copies)

COMPUTERS: BEEPS, WHIRS AND BLINKING LIGHTS _____ (No. of copies)

COMPUTERS: MENUS, LOOPS AND MICE _____ (No. of copies)

I am enclosing \$3.70 per copy (includes 75¢ postage and handling).

I would like a complete set(s) of the three FIRST LOOK BOOKS. I am enclosing \$9.00 per set, postpaid.

Indicate method of payment below:

___ Check/Money order ___ VISA ___ MasterCard ___ American Express

Card no. _____ Exp. date _____
(No cash or C.O.D. orders accepted.)

Send my order to:

Name _____
(Please print)

Address _____ Apt. _____

City _____

State _____ Zip _____

Send this coupon to:

BANBURY BOOKS, INC. 37 West Avenue Wayne, PA 19087

Do you (or your family) own a personal computer? _____ Yes _____ No

If so, what kind? _____

Please allow 6-8 weeks for delivery. PA residents add 6% sales tax.

ORDER BY MAIL . . .

A DICTIONARY OF COMPUTER WORDS

by Robert W. Bly
Illustrated by Jack Freas

Now, for every young reader, a definitive lexicon of everyday words from the dazzling world of computers. Starting with Abacus and ending with Zuse, A DICTIONARY OF COMPUTER WORDS is a complete and fascinating guide to the world of computer language. Detailed and accurate, the DICTIONARY introduces all the buzz words, jargon and baffling acronyms of this new technology. Illustrated with dozens of drawings, A DICTIONARY OF COMPUTER WORDS contains over four hundred entries, including biographical information about notable computer pioneers, and essential technical information.

Please send me _____ copies of A DICTIONARY OF COMPUTER WORDS. I am enclosing \$4.70 per copy (includes 75¢ postage and handling).

Indicate method of payment below:

___ Check/Money order ___ VISA ___ MasterCard ___ American Express

Card no. _____ Exp. date _____
(No cash or C.O.D. orders accepted.)

Send this coupon to:

BANBURY BOOKS, INC. 37 West Avenue Wayne, PA 19087

Send my order to:

Name _____
(Please print)

Address _____ Apt. _____

City _____

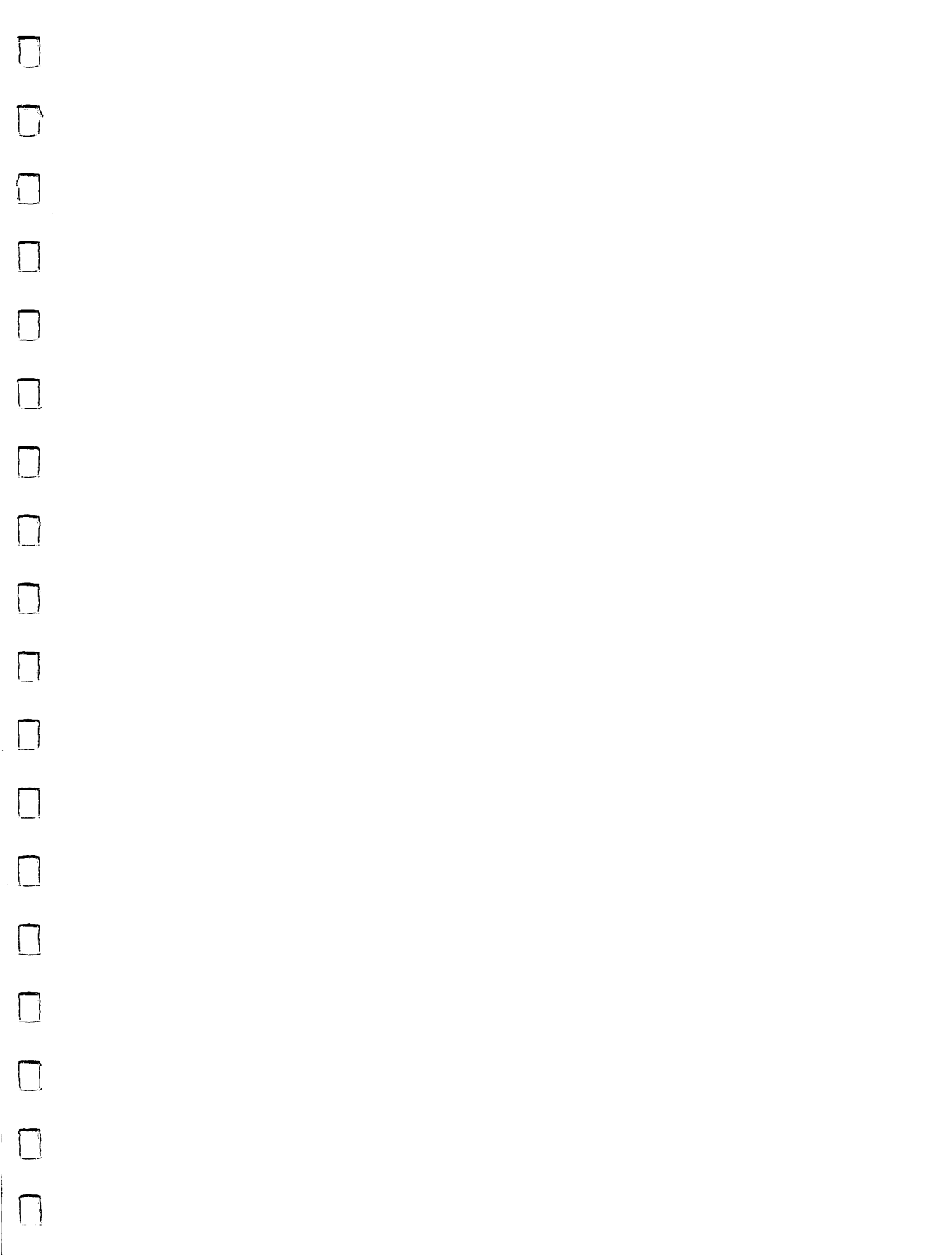
State _____ Zip _____

Do you (or your family) own a personal computer? _____ Yes _____ No

If so, what kind? _____

Please allow 6-8 weeks for delivery. PA residents add 6% sales tax.





Computer literacy has nothing to do with how much you know.
It's all in how you look at the question.

COMMODORE 64

The Intelligent and Intelligible Guide for the Inquisitive Adult takes a fresh look at understanding how computers work.

Unlike other guides, Heil and Martin's book leads you through the idiosyncrasies of programming in utterly manageable bits and pieces. At every step, the book relates the logic behind computers to the world around us.

Parallels from day to day living, images we all understand, are employed instead of mathematical gibberish. Skillfully written, using the language of writers, not computer-types, the book adopts an unusual perspective. In its straightforward manner it debunks a world of myths. And, with its amusing spirit, it makes the process fun.

At long last, here's a no-nonsense guide for those of us who feel the pinch of ignorance in the much ballyhooed age of the computer. A book that makes it easy and exciting to discover exactly what's going on inside the wondrous workings of the Commodore 64.

Patiently and thoroughly, assuming intelligence but absolutely no knowledge of the machine itself, Heil and Martin take the BASIC computer language apart. And then, word by word, phrase by phrase, they put it back together again.

Commands are considered carefully, one at a time.

Miniprograms are developed that illustrate every crucial point.

So, after an hour at the keyboard, you'll have your 64 paying attention to you, not talking back in a jargon you don't understand.

**FOR THOSE WHO KNOW NOTHING ABOUT COMPUTERS . . .
FOR THE INQUISITIVE, INTELLIGENT ADULT . . .
THIS BOOK WILL BE
AN UNFORGETTABLE INTRODUCTION TO COMPUTING.**



ISBN 0-88693-067-7

A Banbury Book \$14.95
Distributed to bookstores by
The Putnam Publishing Group
COVER PRINTED IN U.S.A.