ARTHUR AND ELAINE AARON

DOCTOR AARON'S GUIDE TO THE CARE, FEEDING, AND TRAINING OF YOUR COMMODORE 64®

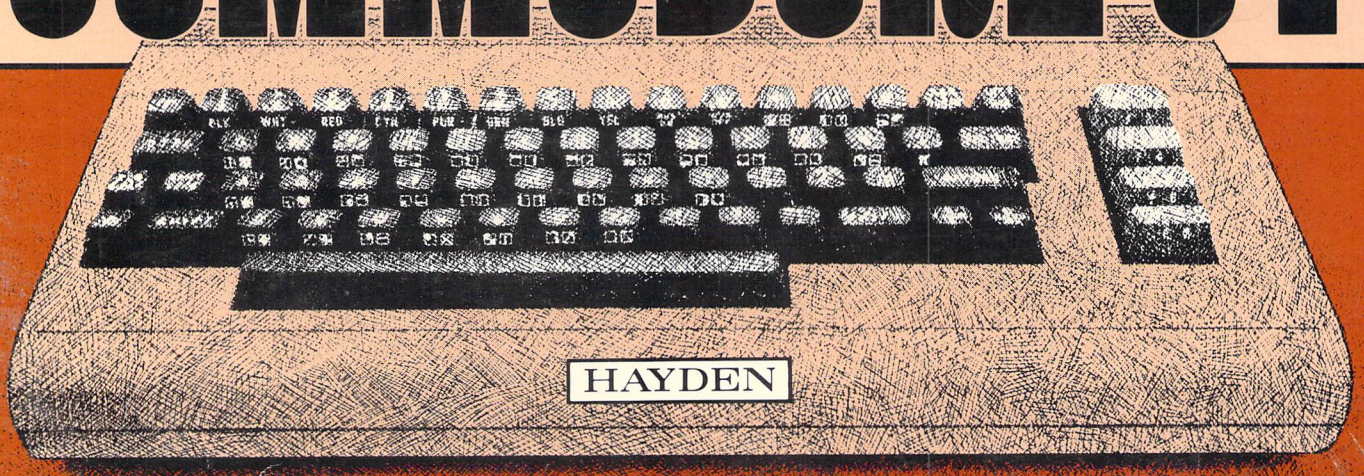
THE USABLE INTRODUCTION TO THE COMMODORE 64

DOZENS OF HELPFUL HINTS FOR THE NOVICE

PROGRAMMING IN BASIC

SPRITE GRAPHICS AND SOUND

TIPS ON PRINTERS, DISK DRIVES, AND INSTALLATION



HAYDEN

DR. ARON'S GUIDE TO THE CARE, FEEDING, AND TRAINING OF YOUR COMMODORE 64®

DR. ARON'S GUIDE TO THE CARE, FEEDING, AND TRAINING OF YOUR COMMODORE 64®

ARTHUR ARON AND ELAINE ARON

HAYDEN BOOK COMPANY
a division of Hayden Publishing Company, Inc.
Hasbrouck Heights, New Jersey

Acquisitions Editor: DOUGLAS K. McCORMICK
Developmental Editor: KAREN PASTUZYN
Production Editor: DENNIS MENDYK
Cover Design: JIM BERNARD
Text Design: JOHN M-RÖBLIN
Illustrations: JOHN McAUSLAND
Compositor: MAPLE-VAIL BOOK MANUFACTURING GROUP
Printer and Binder: COMMAND WEB OFFSET INC.

Printed in the United States of America

1	2	3	4	5	6	7	8	9	PRINTING
84	85	86	87	88	89	90	91	92	YEAR

CONTENTS

DR. ARON'S GUIDE TO THE CARE, FEEDING, AND TRAINING OF YOUR COMMODORE 64®

Introduction

This is probably the best book you will find on computer programming. It's pleasant. And after about 12 hours with it, you will be able to write almost any type of program for the Commodore 64.

Why the confidence? How can we claim that this book is so good? Because it is written by experienced teachers rather than by engineers.

For each make of computer, there are many books, including the instruction manual. But as you may have found, the wonderfully talented computer experts and technical writers who write these books and manuals are usually not very good teachers. Their books are full of jargon, unnecessary technical material, and assumptions about your engineering background! They may start out well, but then they move too fast or make jumps in logic. Good teachers don't do that. But why expect computer engineers to be good teachers? That isn't their specialty.

Teaching *does* happen to be our specialty. Both of us are university researchers with years of computer experience, plus years of teaching such "technical" subjects as statistics to students with no technical background whatsoever. Recently, we became interested in teaching computer programming to very new, sometimes computer-shy computer owners.

We have taught the material in this book many times and revised it many times. We have had computer innocents learn programming with this book entirely on their own—and corrected anything they found confusing. As a result, *this book is clear.*

Here's something else you should know about this book: We've written it as if we were speaking to you in class. And that's because, in some ways, taking a class is a better way to learn to use a computer than reading a book. But you can't take a class any time you have a minute. You can't take it in your own home. And you can't find a class for the price of this book. You can't usually retake a class or pass it on to a friend either. Books have their advantages.

What classes *do* have are teachers. Good teachers anticipate where a student will have difficulties. They repeat unfamiliar ideas until they become familiar. They answer questions *as they arise.* They ask questions to check whether things are getting absorbed. And they spend extra time on points they know most people will nod in agreement with but not really grasp.

At times, you may find our style redundant. Please bear with it. It's good teaching. With computers even more than other disciplines, your understanding will build on your firm grasp of what has come before.

By the fifth chapter of the average computer book, most readers are spending half their time frantically combing the previous chapters, trying to recall terms and techniques that the book assumes to be obvious—they were mentioned once, weren't they? This book repeats terms, techniques, and concepts. It repeats everything that might not be obvious. *This book repeats itself.* It's for your benefit.

Because you are not an engineer, we have tried to avoid using the computer jargon that those wonderfully clever computer engineers enjoy so much. Still, jargon is often born of necessity. When there is no better word than the jargon term, we provide the jargon and its everyday English equivalent together in the text several times. You will also find these terms in the Glossaries. (There's one word in the new computer jargon we ask you to forget forever: *computerphobia.*)

We know from experience that anyone who can read this introduction can learn to program a computer. Basic computer programming is one of the easiest skills to learn. What a fortunate irony that these days it carries such status! It's even sweeter because you don't need good grammar skills or good math skills. Computers require very little more than the patience to learn their language. Choosing to learn this skill was really very clever of you.

One last word. Computers are our creations, designed only to serve us. So, enjoy yourself!

How to Use This Book

The goal of this book is to teach you how to use the Commodore 64 *as quickly and thoroughly as possible.* For those of you who were in a hurry and skipped the Introduction, one point bears repeating: This book reads as a good teacher would speak to you. It repeats itself when a point bears repetition. This saves time in the long run.

Nevertheless, different readers come to this book with different backgrounds. We've designed this book to be optimally efficient for each of you. This book has four sections:

Part I: The Fundamentals

Part II: More on Programming

Part III: Show Business: Graphics and Sound

Part IV: Putting It All Together

If you want a quick introduction, use only Parts I and IV. This takes only about five or six hours, including all the practice exercises.

If you want to learn programming but are *not* concerned about "graphics"—fancy displays, business graphs, animation, original video games, computer art, and so forth—use Parts I, II, and IV.

If you want a thorough introduction to programming the Commodore 64, use Parts I, II, III, and IV.

As for your prior computer experience:

If you are a novice—

1. Sit with this book and the computer in front of you. Follow each step. Do everything the book asks you to do. A large part of computer competence develops in the actual doing.
2. Do the practice exercises at the end of each chapter faithfully. They are sometimes fun, always practical, and essential to your learning.

If you have worked with other computers, but the Commodore 64 is new to you—

1. Follow the guidelines above for Chapters 3 and 4, the section in Chapter 5 on editing program lines, and all of Part

III if you want to explore the sophisticated graphics and sound of the Commodore 64.

2. For the other chapters, try reading just the summaries at the end of each chapter. They are very terse, but very complete. If you do not understand a topic from the summary, you can then read the relevant section of the chapter.

If you have been learning to use the Commodore 64 from the manual or other books, but this is your first computer—

1. It may be enough to read only the summaries of Chapters 1 and 3.
2. After that, we suggest you use the book much like a novice, although you will probably proceed more quickly. Self-taught owners are often quite skilled, yet they have unanimously extolled the value of following the lessons in this book step by step. It fills in all those gaps of knowledge and answers all those nagging questions. Still, you are welcome to try reading just the summaries at the end of each chapter. They are very complete. If you find they are explanation enough, and you can do the practice exercises at the end of each chapter, proceed until you come to something unfamiliar.

PART I
The Fundamentals

PART I CHAPTER 1

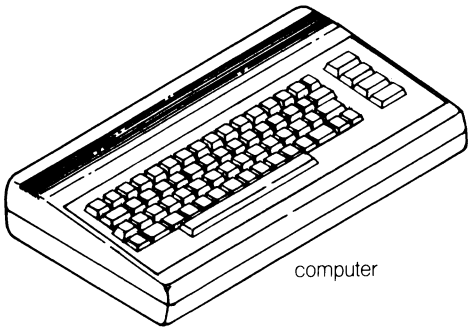
Setting Up Your Machine

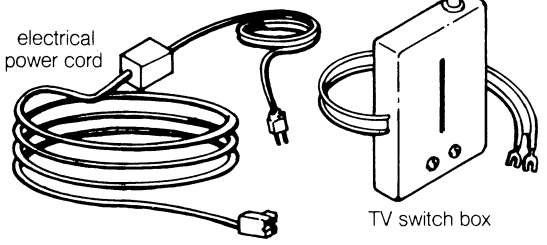
You will need three things to set up your Commodore 64 computer:

1. Your Commodore 64, including all the cords and connectors that come in the box with it.
2. An ordinary TV. Color would be nice, but most of what you can do on the computer you can do just as well with black and white.
3. Tools: a screwdriver and a pencil. The pencil is to move a small switch deeply recessed into the computer and hard to get at with your finger. A pen, small stick, or very thin finger would also do.

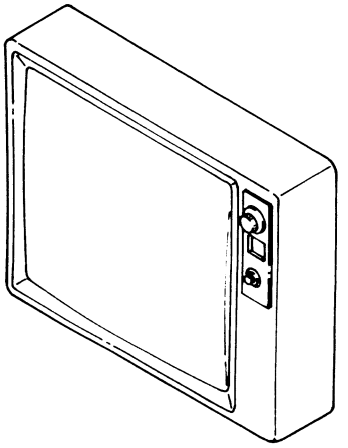
THINGS YOU NEED

1. Computer and Connectors

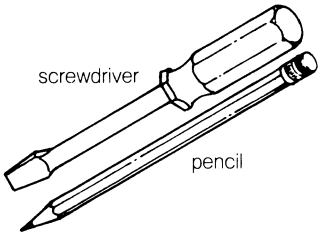




2. TV



3. Tools



If you have also bought a Datassette, a printer, a modem, a disk drive, cartridges, software, or any of the other accessories available for your computer, put them aside for now. You will learn soon enough what they do and how to use them. For now, all you need are the basics.

Of course, in using the manual you may have already set up your computer—once, or many times. Did you have trouble? Let's go through the process very carefully so it is trouble-free.

1. Open the box and put your computer on a table near your TV.
2. With the screwdriver, loosen the VHF antenna connector screws on the back of your TV and disconnect the antenna wires. (If you look at the back of your TV and can't find the VHF antenna screws, see Box 1-1.)
3. In place of the antenna wires, stick in the connectors attached to the wire on the TV switch box (the little tin box that comes in the package with your computer). It doesn't matter which wire gets attached to which VHF screw. Tighten the screws.

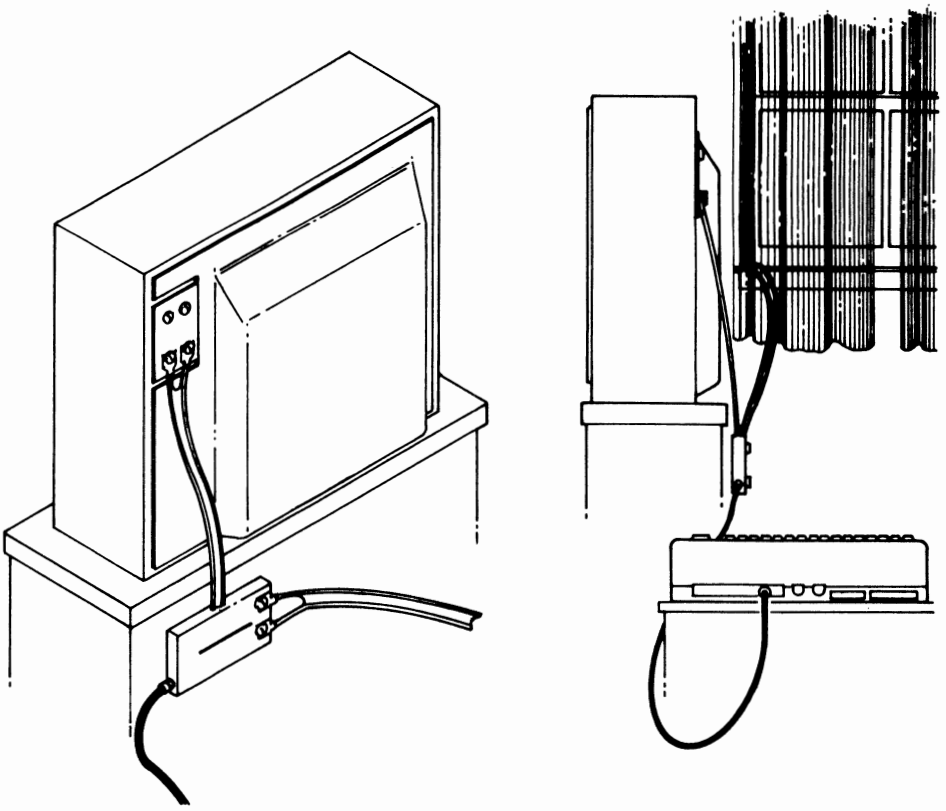
Box 1-1
If You Can't Find the VHF Antenna Screws

If you have a very old TV set, it may have only one pair of antenna connector screws. Those are VHF.

If there are two sets of antenna connector screws and you can't tell which is VHF, try one pair, then the other. It won't hurt your computer to use the wrong one; it just won't work.

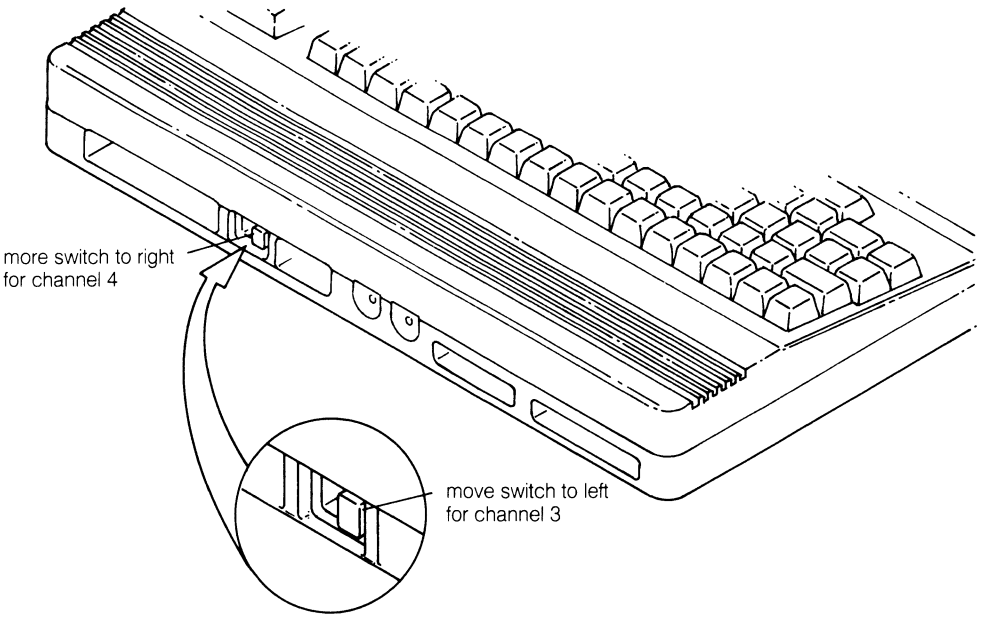
If you have a very new TV set, it may have a round, threaded 75 ohm VHF antenna connector. In this case, you will have to go to a local electronics or TV store and buy a 300 ohm to 75 ohm adapter. (This should cost about $4.) If you also plan to use your set for regular TV, buy a second adapter that goes the other way—a 75 ohm to 300 ohm adapter—to connect the TV switch box to the antenna! (It's about the same price.)

CONNECTING THE TV SWITCH BOX



4. If you also plan to watch TV on this set, attach the now loose antenna wires to the two screws on the TV switch box labeled "CONNECT TO ANTENNA." By using the slide switch on the TV switch box, you can switch back and forth between using your set for the computer and for TV. For now, slide it to "COMPUTER."
5. Put away the screwdriver now. That's all the "electrical engineering" you'll be doing.
6. Plug one end of the TV connector cord (the thinner black cord) into the little, round, reddish hole on the back of the computer (the only hole this cord will plug into). Plug the other end into the similar hole on the TV switch box.
7. Check to make sure that the switch on the right side of the computer is NOT on. Plug the small end of the power cord (the one with the heavy black box in the middle) into the hole marked "POWER" just behind the "on" switch. To do this, the notch on the plug has to be at the top. Plug the other end into an ordinary three-prong electrical outlet.
8. On the back of the computer, right next to where you plugged in the TV wire, is a square hole in the plastic, with a little slide switch recessed about a quarter of an inch inside. This is a TV channel selector switch. Use a pencil (or something similar) to slide it. Sliding it to the left (as you look at it straight on from behind) sets it to channel 3, sliding it to the right sets it to channel 4. Slide this switch to whichever channel you *don't* get in your area. Then tune the TV to that same channel, turn down the volume, and turn it on. You should get nothing very interesting on the screen.

THE TV CHANNEL SELECTOR SWITCH



Box 1-2
If It Doesn't Work

1. If the little red power light doesn't come on, check the following:
   a. Is the power cord plugged in securely to a standard household electrical outlet?
   b. Is the power cord plugged in securely to the correct place in the computer?
   c. Is the computer on? If it is not clear which way "on" is, try it both ways.
   d. Check the electrical outlet (and any extension cord you might be using) by plugging in a lamp or a radio.

If the power light still does not come on, take your computer back to your dealer and ask for help.

2. If the little red power light comes on, but you can't get a clear picture on the TV:
   a. Don't be a perfectionist. The Commodore 64 picture will always be a little fuzzy on a TV screen no matter what you do. You *can* get a sharp picture on a special TV made for computers, called a *monitor*. Monitors are discussed in Chapter 19.
   b. It is not uncommon to have to do a lot of adjusting of your TV set to get the com-

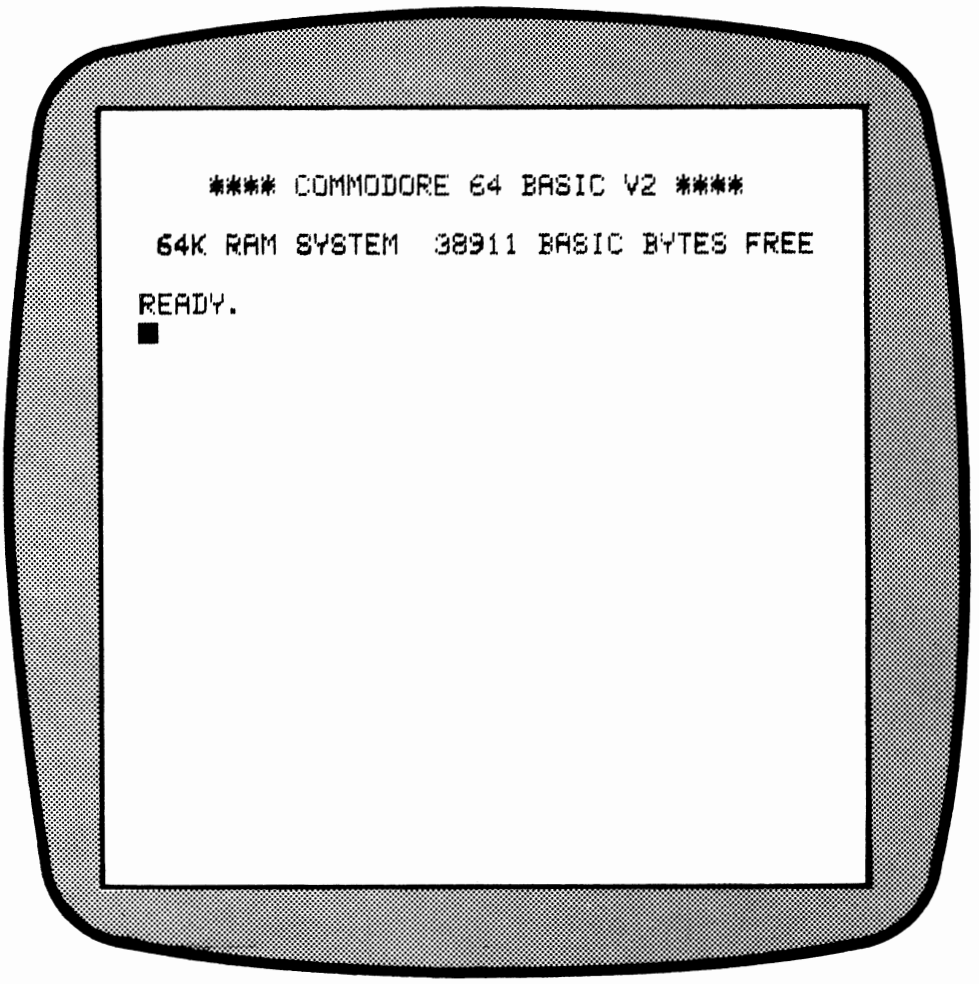
puter picture. First try fine tuning, then horizontal and vertical hold, then the color controls.

c. Check the TV-computer connection:

- Are the wires that come out of the TV switch box screwed in firmly to the VHF antenna connector screws on the TV set?
- Is the TV switch box slide switch set all the way on "COMPUTER"?
- Is the thin, black connector cord plugged in firmly to the TV switch box and the back of the computer?
- Is the little TV channel selector slide switch (the one recessed into the back of the computer) set to the same channel as the TV?
- Try setting the TV and the computer to the other channel.
- Check to see if your TV has a 300 ohm-75 ohm switch or any other special switch on the back and try setting it the other way.
- Try another TV.

If you *still* don't have a clear picture, see your dealer for help. Be sure to take all your cords and the TV switch box with you—these are more likely the problem than the computer itself.

9. Now, turn on the computer. The little red power light on the top of your computer should come on, and then in a couple of seconds (and after what is often quite a bit of adjusting of your TV), the screen should look like the following figure:



10. Admire your investment.
11. If it doesn't work—if the screen doesn't look like the figure above—see Box 1-2. Fix the problem and *now* admire your investment!

Your Commodore is **READY**.[1] You are the admiral. It's time to give it your commands.

In the next chapter, you will learn to make the computer do all kinds of wonderful things by simply plugging in a special cartridge or by playing a special cassette tape. These cartridges or tapes (called *software*) are prepackaged sets of instructions *(programs)* that tell the computer what to do in a language it understands.

If you don't own any such cartridges or tapes, or if you are not in a hurry to use them, skip directly to Chapter 3. There, you will learn to speak the computer's language yourself, so you can give the computer instructions to do exactly what you want, interacting with it directly.

3. If you get a clear picture from the computer, but the TV screen doesn't look the same as the figure showing the screen directly after you turn on the Commodore 64:
   a. Turn your computer off and then on again. (You may have accidentally pressed some keys.)
   b. Check the back of your computer to be sure there is no cartridge plugged in to the big rectangular hole (next to the little TV channel selector slide switch). If there is, turn the computer off, gently but firmly pull out the cartridge, and turn the computer on again.
   c. Check to make sure there are no other accessories attached that might somehow be on—such as a disk drive or a Datassette tape recorder.

If you still have problems, see your dealer.

Summary

This chapter describes the steps involved in setting up your computer: Take the computer out of the box and set it near your TV. Attach the computer to the TV by (a) connecting the TV switch box to the TV's VHF antenna connector screws and the antenna wires to the switch box, (b) plugging one end of the thin, black TV connector cord into the TV switch box and the other end into the computer, and (c) setting the TV and computer to the same channel (either "3" or "4"—whichever you don't get in your area). With the computer off, plug the power cord into the computer and into the wall. Turn on the TV and the computer.

[1] Did you wonder just what 64K RAM SYSTEM 38911 BASIC BYTES FREE means? RAM means "Random Access Memory"—the kind of memory in which you can put in and take out information (as opposed to ROM, or "Read Only Memory"—the kind of memory that the computer uses to remember how to operate properly). You can think of both kinds of memory in terms of hundreds of thousands of little switches that can be either on or off. One switch being on or off is one "bit" of memory. The combined memory of eight bits is called a "byte." Every letter you type in takes up one byte of memory. One "K" (short for "kilobyte") is 1024 bytes. BASIC is the name for the system of giving instructions (the "language") to most personal computers. So, what the screen is telling you is that the Commodore 64 has 64K of RAM, of which 38911 are available for programs you write in BASIC. But you really don't need to know any of this to use your Commodore 64 to do all the great things it can do for you.

PART I CHAPTER 2

Canned Programs ("Software")—Getting Started Right Away

Your Commodore 64 is standing at attention, **READY** for instructions. You are the admiral.

The easiest way to get started is to let someone else give it the instructions. Commodore Business Machines and hundreds of independent companies sell cartridges, tapes, and disks for your Commodore 64. You just plug them in, and you're on your way to anything from playing "Frogger" to preparing a financial statement.

These prepackaged sets of instructions for your computer are called *software* (as opposed to the actual machines, which are called *hardware*). They come in a variety of cartridges, tapes, and disks and are very easy to buy and use.

In this chapter, you will learn the simple details of how to plug software into your Commodore 64. You will also look at the kinds of software that are available, as well as some pointers on what and how to buy. Then, in subsequent chapters, you will learn how to write your own programs! (If you do not own any software, or if you want to skip this topic for now, you can go right on to Chapter 3.)

Types of Software Available

Software on Cartridges

The easiest kind of prepackaged programs to use come on Commodore 64 *cartridges*—plastic boxes about the size of a large bar of soap. (Do not confuse these with "tape cartridges" used in tape recorders; software also comes on cassette tapes, but we will save that for discussion in the next section.)

To use these cartridges, first turn off your computer. Gently but firmly push the end of the cartridge with the metal connectors on it into the cartridge slot (the biggest opening) on the back of your computer next to the TV channel selector slide switch. Once the cartridge is firmly in place, turn your computer on. The instructions on the screen should tell you exactly what to do.

Cartridges have two big advantages over the other kinds of prepackaged programs. First and most important, you don't have to buy any additional accessories, like a tape player or a disk drive. Second, cartridges are extremely easy to use. You just plug them into the slot.

But cartridges also have their limitations. You can't store your own information on a cartridge. A program that helps you type out a school report or prepare a payroll is not nearly so useful as it would be if you could store the results for future reference. Second, cartridges can be used only for fairly simple programs. Therefore, most business and educational programs—and even many of the more interesting games—are not available on cartridges. Furthermore, although cartridges do not require the purchase of any additional accessories, cartridges are often more expensive than tapes or disks.

Software on Cassette Tape and the Commodore Datassette

Some prepackaged programs are available on ordinary cassette tapes—the same kind you use to record a lecture or listen to music on a portable cassette player. To ensure smooth operation, Commodore sells a special tape player called a Datassette, which is specifically designed for use with the Commodore 64.[1]

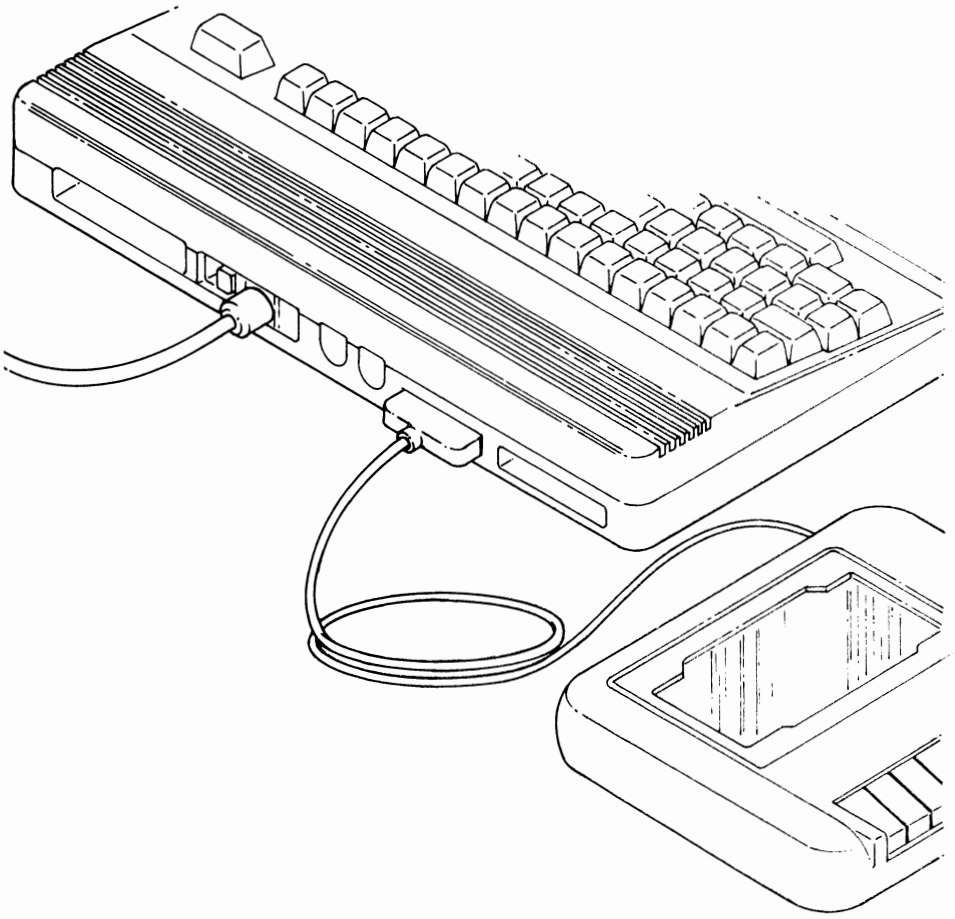
A Datassette is a very good investment. It makes many prepackaged programs available to you that would not be available on cartridge. It gives you a way to save the information you get from running your prepackaged programs. When you start writing your own programs, it provides a way to save them for future use. In fact, unless you will be buying a disk drive (see next section), a Datassette is indispensable for any but the most minimal use of your computer.

To use a program that comes on cassette tape:

1. Plug the Datassette into the back of your computer (see the next figure). The gray plug, with the screw head side on top, goes into the smaller rectangular hole with the tongue with little metal strips on it. This is the only place it fits.
2. Use the EJECT button on the Datassette to open the cassette pocket. Insert the cassette with the first side of the tape up and the open-tape end facing outward. Close the Datassette.
3. With the computer turned on (the little red light should be on and the TV screen should show the computer display), press the REWIND button on the Datassette and wait for it to stop. If it is a new tape, it is probably already rewound,

[1]You can buy—or make, if you are electronically handy—a device to convert an ordinary cassette player to work with your Commodore 64. But the Commodore Datassette is not very expensive and is trouble-free with the Commodore 64 computer. You may not have the same luck with another cassette player.

PLUGGING IN A DATASSETTE



so it will stop right away. (If it takes quite a while to rewind, you might have put on the wrong side of the tape.) When it is rewound, press the STOP button on the Datassette.

4. Press the [L], [O], [A], and [D] keys on your computer. This should make the word **LOAD** appear on the TV screen in all capital letters. In Chapter 3, you will learn how to use the keyboard and how to fix mistakes. For now, if you make a mistake, turn the computer off and on again, and try once more.
5. Press the [RETURN] key. The screen will say: **PRESS PLAY ON TAPE**
6. Press the PLAY button on the Datassette. The TV screen will go blank—actually light blue—for several seconds. (How long depends on how complicated the program is.) Then the Datassette will stop moving and a message will appear on the screen that your program has been found. It will actually give the name of the program it has found. If the program it found was not what it was supposed to find, you probably put in

the wrong tape or the wrong side of the tape. In that case, press the [RUN/STOP] key on the computer, press the STOP button on the Datassette, take out the tape, and try again.

7. Presuming the computer found the right program, press the COMMODORE KEY. It is the bottom left key on the computer and has the Commodore corporate symbol on the front of it. The screen will go blank again. But when the screen comes on again, you are ready to start. The screen (or the instructions that came with the tape) should tell you what to do from there.

Box 2-1
If You Own a Disk Drive

To use your disk drive to load software into your computer, follow these steps:

1. Plug in the electric cord.
2. Plug one end of the black cord into the *serial port*—the hole on the back of the computer next to the slot for plugging in the Datassette. Plug the other end into the disk drive.
3. Turn on the disk drive's on-off switch. The little green light should come on.
4. Open up the flap on the front.
5. Find the name of the program (it will be a brief title of sixteen letters maximium) for the diskette you want to use. It should be in the papers that came with the diskette, but it might also be on the label.
6. Hold the diskette by the label, with the label facing up (the little square notch will be on the left near the front), and slip it gently into the slot.
7. Close the flap.
8. Type the word **LOAD**, the program name (within quotes), a comma, and the number **8**. For example, for a holiday game named "HOHOHO," type **LOAD "HOHOHO", 8**. Check the screen

Software on Disks and Disk Drives

A *disk drive* is a box about the size of a telephone. It is like a tape player—except that instead of playing tapes, it plays *diskettes*. These are like small records. They are about five inches across, and they are always kept wrapped in a square paper envelope. To "play" one, you just insert the diskette into the opening on the disk drive and press the appropriate keys on your computer.

Disk drives are expensive. They cost about as much as or slightly more than the computer itself. Since most users don't buy one to start with, we will not go into detail here about using one. For those who have one, see Box 2-1.

Most serious computer users will, eventually, buy a disk drive. They do what your Datassette does—but they do it better. You can play and record information with them much faster than with tape. If you have a program with many parts, or one that handles a lot of information, the computer can locate it on a disk without having to go through everything else first (as it has to do on the very linear tape). Most important, sophisticated software—such as top-quality word processing and financial management programs—usually are available only on disks!

Unless you plan to use your Commodore 64 right away for some purpose that requires a disk drive, we recommend starting out with a Datassette. Later, you can upgrade to a disk drive.

Kinds of Software

There are five categories of software used with the Commodore 64:

1. Entertainment
2. Word processing
3. Educational
4. Business
5. Miscellaneous

The following sections explain each of these and then discuss some points to consider when selecting software. We will not rec-

ommend specific programs or brands. What is available is increasing every day. There is already so much in every category that we could not do it all justice. But we will give you an idea of the issues involved in making your buying decision.

Entertainment Programs

The Commodore 64 is a highly sophisticated computer. But it can also play games with you. Game programs are probably the most popular of all software sold, so don't feel shy if that's what you want to buy. You have a lot of company. There are games to interest almost everyone—from adventure to sports to chess. New games hit the market every day, with increasingly clever themes, as well as better and better sound and graphics.

Most games are available on cartridge. But as games have become more sophisticated, there has been a trend toward disks. So even if all you're interested in is games, you may want to buy a disk drive eventually.

Word-Processing Programs

Word processing is one of the most exquisitely efficient uses of the Commodore 64.

A word processor is simply a computer program that lets you use your computer to type things like letters and reports—and then sets up what you have typed so it can be printed out on paper. But that's just the nose of the horse.

With a word processor it is much, much easier to correct mistakes. You can go back and put in or take out a letter, word, or even a whole paragraph or page. The computer adjusts everything else and leaves no trace of the problem on the page it prints out.

Some word-processing programs even allow you to do such things as replace one word with another every place it is used (for example, to personalize a form letter, change the name of a character in a short story, or correct the misuse of a term). Some allow you to type footnotes right into the text as you go along, then put them perfectly arranged at the bottom of the page when everything is typed out. If you decide to revise what you have written, or use it again with some changes, you have the old version saved on tape or disk. Just make the corrections and presto—a new version is ready to go. There's no need to retype the whole thing.

Besides helping you write and revise, word processors type the final product in exactly the form you need. You can type it any old way on the machine and later decide on margin settings and spacing. Some word-processing programs even allow you to do such things as "justify"—space the words to make the right margin come out even, like book pages—or to change text to italics or bold type.

In addition, with some word-processing programs, you can buy a subsidiary program that checks what you have typed for spelling errors!

to make sure you typed it correctly, including the spaces and punctuation, then press the [RETURN] key.

9. The TV screen will say **SEARCHING FOR HOHOHO,** then it will add **LOADING**. Next, it will let you know it is ready.
10. Then type **RUN** and press the [RETURN] key. From there, follow the instructions that came with the disk!

Note: Diskettes are a bit delicate. It is a good idea to make a copy of any expensive disk software as soon as you buy it (Load it, then save it on a new disk—see Chapter 6, Box 6-1, for details of **SAVE**ing.) Then put it away in a safe place. Also, be careful not to get the diskettes wet or dirty. Most important, keep them away from anything magnetic (don't set them on your TV or computer, and keep them away from electric motors—such as vacuum cleaners, magnetized paperclip holders, and other such paraphernalia).

Of course, to get all this you have to buy a word-processing program and a printer. Let's consider each.

Word-processing programs differ dramatically. Some allow you to do little more than type onto the screen, do minimal correcting, and print out what you have typed just as you see it on the screen. Others allow all kinds of fancy editing and printout adjustments.

If you do a lot of typing, the extensive features of a topnotch word processor are well worth having. You may not see all their advantages in a ten-minute trial at the computer store (but try them out anyway). When you are typing day after day, however, you come to appreciate them. It is especially important to be able to correct mistakes easily, move sentences and paragraphs around, and center words on the page. With a feature called *word wrap,* you don't have to press the carriage return key because the computer knows when to move on to the next line. Also very important is the flexibility with which the word processor reorganizes the layout of what you type for printing.

Price is not necessarily an indicator. The very best programs may cost less than poorer ones! If you are serious about word processing, look into what is available. Try out the programs at the computer store. Type the kind of material you will actually be using the word processor for. Read the reviews in computer magazines. And, of course, take into account all the points we will discuss later in the section on buying software in general.

There are basically two kinds of printers. *Dot-matrix* printers print what you have written with little dots. The letters are a bit fuzzy, gray, and "space age" looking. Printers that type fully formed letters as a typewriter would are called *letter-quality* printers. Letter-quality printers are slower and more expensive, and do not usually have any ability to produce a computer's special graphic characters. But the output does look like real typing. For business correspondence, manuscripts, or any professional use, you will probably have to have a letter-quality printer.

If you can make do with a dot printer, there are tremendous differences among them. The only printer that is made to go with the Commodore 64 is the VIC 1525 GRAPHIC PRINTER, a dot printer. It is currently the only printer that can produce all the special graphic symbols you see in the little boxes on the keys of of your Commodore 64. It hooks up directly to the Commodore 64. To use any other printer, you have to buy a special attachment. The VIC 1525 makes fair quality letters and is reasonably fast. You could do worse. But if you have a little more money, you can do a lot better. We have more to say about printers in Chapter 18, when we consider the various accessories you can buy for your computer.

One other note about word processing. In most ways, the Commodore 64 is ideal to use as a word processor. It has a good keyboard, plenty of memory, and good word-processing programs available to it. It is also inexpensive. The one disadvantage is that

it writes only 40 letters across the screen on any one line. A standard typed line has about 60 to 80 letters on a line. This is not an insurmountable problem. You type things with 40 letters across and then the printer makes the lines as long as you choose. But some people find it a little annoying not to be able to see a full line on the screen. You will have to try it for yourself. Test out a word-processing program on a Commodore 64 at your local computer store.

Educational Programs

Perhaps the next most popular kinds of software purchased for the Commodore 64 are programs that help you learn—from typing to advanced math, from flying a plane to second-grade reading.

Educational programs vary in quality even more than other kinds of software. In this case, quality means how well it teaches and how easy it is to use. Many programs are designed to supplement what a student is learning in school. In that case, be sure that what is considered grade-seven geography in your school system is what the software company considers grade-seven geography. Generally, what is interesting and relevant definitely differs from student to student!

It is important to actually see an educational program in action before buying it. Be sure that you (or your child) can use it easily and that it covers what you want. And is it *interesting?* Some programs are incredibly dull. Others give fancier (and in effect, more rewarding) responses when you are wrong than when you are right. Some progress like a nervous antelope, others like a slug. Many are written not by educators but by computer people.

Unfortunately, since most computer stores do not carry a large line of educational programs, it is hard to try them out. But call around town. Also, a local school (perhaps a private school) that uses the Commodore 64 may let you try out programs it has bought. If it is using Commodore VIC-20s, you can still try them out because many programs available for the VIC are also available for the Commodore 64.

Business Programs

The computer can make managing your affairs or the affairs of a business much easier. There are lots of programs to help you do this. The Commodore 64 is an excellent business computer. It does just about anything computers ten times its price do. The main advantages of the more expensive machines are status and greater availability of software. As for the first, perhaps there is also status value in being the kind of businessperson who keeps costs down! As for the second, more and more software is becoming available for the Commodore 64 every day.

There are three main kinds of programs available for business use. Most useful are what are called *database management* programs—programs that file and sort information for you. You

will learn how these work (and how to write your own) in Chapter 9. Basically, they keep track of such information as mailing lists, inventories, or customer files. Then they let you call it up sorted in various ways, such as all the names from California on your mailing list or all the items in your inventory that are from a particular manufacturer.

Next most useful are what are called *spreadsheet programs*. The original and most famous of these is *VisiCalc*. VisiCalc is not available for the Commodore 64. But there are other programs like it—some quite good—that can be used on the Commodore. These programs make the TV screen a chart with columns going across and rows going down. You then lay out on it anything from your travel itinerary to a financial statement. Its real value is that you can write in formulas that transform what goes into a particular box on the sheet depending on what went into the boxes before. That allows you to project future sales, for example, or calculate the expense of your trip plan. Spreadsheet programs use a jargon all their own. They take a while to learn to use well but are extremely flexible and useful once you have them down.

In addition to these two classes of programs—database management and spreadsheet programs—many specialized programs are available, particularly in the area of payroll, property management, billing, stock-market analysis, and the like. These vary greatly according to whether they are designed for someone who is knowledgeable about the field (such as an accountant) or for someone who wants the program to serve as the expert.

Miscellaneous Programs

This is the most exciting software area because this is where all the new developments are taking place.

One whole category is called *utility and language programs*. These help you use your computer in the most general way. They include languages that are easy for kids to learn, software that help you write your own programs in what is called *machine language,* and software programs that help you take better advantage of the graphics and sound ability of the Commodore 64.

The other main group of prepackaged programs comprises those for specialized professional and personal interests. For example, there are programs to help photographers produce optimal lens settings, to help you determine your home's energy efficiency, and to help doctors make diagnoses and lawyers prepare cases. There are programs to help you cook better, stay healthier, and shop better. There are even programs that claim to do counseling or make you laugh—even programs of catalogs of other programs!

Deciding Whether to Buy Software

When looking at an enticing ad in a computer magazine or listening to a friend extol the virtues of some program, how do you decide if it is really worth the investment?

The first question is: Do you really need it? Will you use it, or will it just sit on a shelf? Even game programs are often used only once and then left sitting.

Even if you would use it, have you considered the alternatives? Some costly programs you can actually program yourself in a matter of an hour or two—once you have worked your way through this book. Often, too, you could copy a program out of a book or magazine that would do the same thing.

In general, programs involving a lot of graphics or that do highly complex activities are worth buying. Few people would want to spend the time and effort to write their own word-processing programs or create a video game with highly sophisticated animated graphics.

On the other hand, after you finish this book, you will be able to write, in a reasonable amount of time, educational programs precisely suited to your level of ability—programs to memorize language vocabulary or technical terms, or to study more complex topics. You will be able to write database management programs—exactly the way you want them—and some simple but interesting games. In short, you will have a customized program and the satisfaction of using your own handiwork.

Some Tips for When You Do Buy Software

Software can run anywhere from $15 to $1500. Make your decision carefully, avoid impulse buying, check out more than one store, and compare the different programs that claim to do the same thing. Ask the store dealer to let you try the program out in the store. Work with it long enough to get a feel for it. Check the new software catalogs and the computer magazines (including back issues at your library) for reviews. Ask anyone you know who owns the program about their experience with it.

Above all, be sure any software you buy works on your machine. Even software made for the VIC-20 won't usually work on the Commodore 64. Those made for the IBM®, Apple™, or some other machine will certainly not work. Even if the program says it's for the Commodore 64, check what additional accessories you must have. Some programs, in fact, will work only on Commodore 64s made before the spring of 1983. (Ask your dealer about this.) Does the program require a Datassette or disk drive, a p.inter, a

special TV set, or a joystick? Be sure it comes with written instructions (called *documentation*) that are clear, so you can figure out how to use it.

Finally, there is the question of whether to buy from a store or by mail. Mail order has the advantages of much greater selection and cheaper prices. If you are absolutely sure you want the program, and if a reputable company produced it—or if you cannot get (or order) the program from a local store—then buy through the mail. Otherwise, do not.

A store must stand behind its merchandise. Stores usually let you try programs out in advance. Even after you've bought their software, if you cannot figure out how to use it or it does not work right, it is in the store's best interest to help you.

Of course, all this help is why stores have to charge a bit more—but with computers, it is worth it. So much of all this technology is still in the debugging phase that ordering by mail is like picking blueberries on a foggy night: Not only is it difficult to find the fruit, but you also have no way of knowing whether it has bugs.

There is no reason not to shop around for the best price among computer stores or even to bargain with them about price. There is usually a pretty good markup on software, and you may be able to knock down the price on an expensive item with a little haggling.

If you do buy by mail, try to check out the seller. See if you know anyone who has bought something from the seller. (If you belong to a users' group, this is important information to share.) You can call the Better Business Bureau in the city where the mail-order house is located. You can also check with the magazine that ran the ad to see if there have been any complaints. None of these are a guaranty, but they help. For a guaranty, you can always try asking the mail-order house to send you one with your purchase. There's no harm in asking.

In this chapter, you have learned how to use prepackaged programs, been introduced to what is available, and received some purchasing advice. If you wanted to get started right away with programs all prepared for you, this was your chance. In the next chapter, you will get started on a different course—writing your own programs. The first step is learning to operate the Commodore 64 keyboard!

Summary

This chapter explains how to use, select, and buy prepackaged programs.

Prepackaged programs are called *software*. They come in three forms—cartridges, cassette tapes, and disks.

Cartridges plug directly into the back of your computer. No additional accessories are needed to use them. Cartridges, how-

ever, cannot store any information or programs you produce, and much of the more interesting software is not available in cartridge form. Also, when the same program is available on tape or disk, it is usually cheaper than the cartridge form.

You must own a special cassette player—the Commodore Datassette—for software that comes on tape. To use the tapes, first plug the Datassette cord into the back of your computer. Next, put the tape in the Datassette, rewind it to the beginning, and type **LOAD** on the keyboard. The computer tells you what to do from there to help it find the program. Once it has found it, you press the key with the Commodore symbol on it and you're on your way.

A Datassette is a good investment for anyone who is at all serious about using his or her computer. It permits you to store information and the programs you write, and many prepackaged programs are available on tape that you could not get in cartridge form.

Eventually, if you are using your computer quite a bit, or if you need sophisticated software, you'll want to buy a disk drive. This is a more advanced way of saving and playing programs and information.

There are several major categories of software you can buy:

1. Entertainment programs: These come mainly on cartridges, although some of the most interesting games are available only on disk.
2. Word-processing programs: These programs allow you to use your computer as a typewriter. They make it much easier to correct and revise your typing and to lay it out effortlessly on the printed page. With some word-processing programs, you can even buy a subsidiary program to check your spelling. Word-processing programs differ a lot and require comparison shopping to make a good choice for your needs. Keep in mind that the most versatile programs require a disk drive. In all cases, you will have to buy a printer.
3. Educational programs: These programs vary considerably in how well they teach and how easy they are to use. If the program is to supplement a child's curriculum, you must consider whether the grade level for the subject matches that of your local school system. Other factors to consider are whether the programs are interesting, well paced, and generally educationally sound—which is not necessarily so! If possible, view the program in advance at a local store or school.
4. Business programs: The most useful programs in this category file and sort information. They are called *database managers*. They help you keep track of such things as a mailing list or an inventory. You will learn about these (and how to write your own) in Chapter 9. Another very useful category is *spreadsheet* programs, which make the computer screen into a columnar pad on which you can lay out such

things as itineraries and sales reports, and write in formulas that allow you to project sales, compute expenses, and so forth. Finally, more specialized programs are available that do accounting and other management tasks.

5. Miscellaneous programs: These include programs that make your computer work in program languages a child can understand or help an adult use machine language or make better use of the graphic or sound capabilities of the Commodore 64. Then there are all the programs for highly specialized personal, household, hobby, or business uses.

In general, it is worth buying a prepackaged program that does a very complex task (such as word processing) or involves extensive graphics (such as many video games). Otherwise, you can often write the program yourself (when you are finished with this book), or copy one from a book or magazine without spending any money.

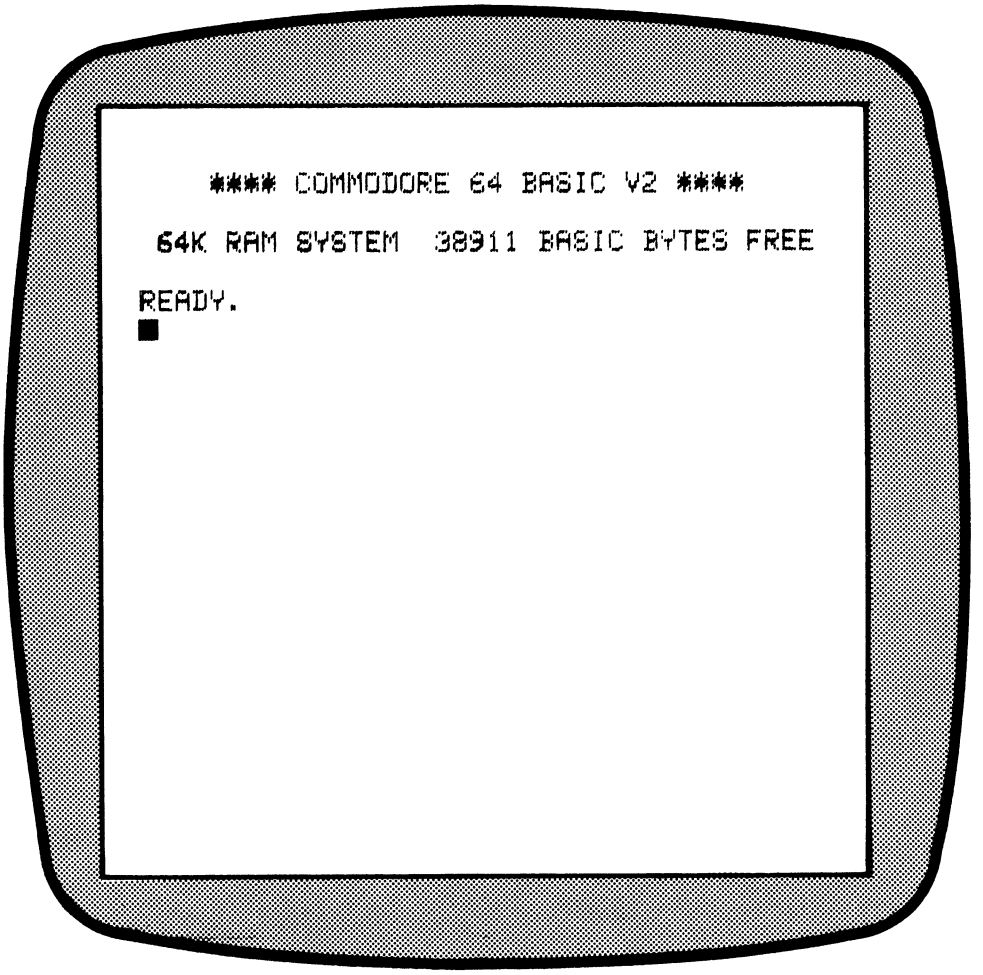
If you do decide to buy a prepackaged program, do some comparison shopping. Be sure the software you buy works on the Commodore 64 and doesn't require any accessories you don't have. In general, it's better to buy from a local store. If you must buy by mail, check out the company first as thoroughly as you can.

PART I CHAPTER 3

Using the Commodore 64 Keyboard

Now, you are going to systematically play around with the Commodore 64! Remember when you first saw it? Did its high-tech look impress you, with all those mysterious buttons on the keyboard? Did you wonder, "Will I *ever* be able to use all that?" In less than an hour, you will.

After you turn on the computer, your TV screen should look like this:



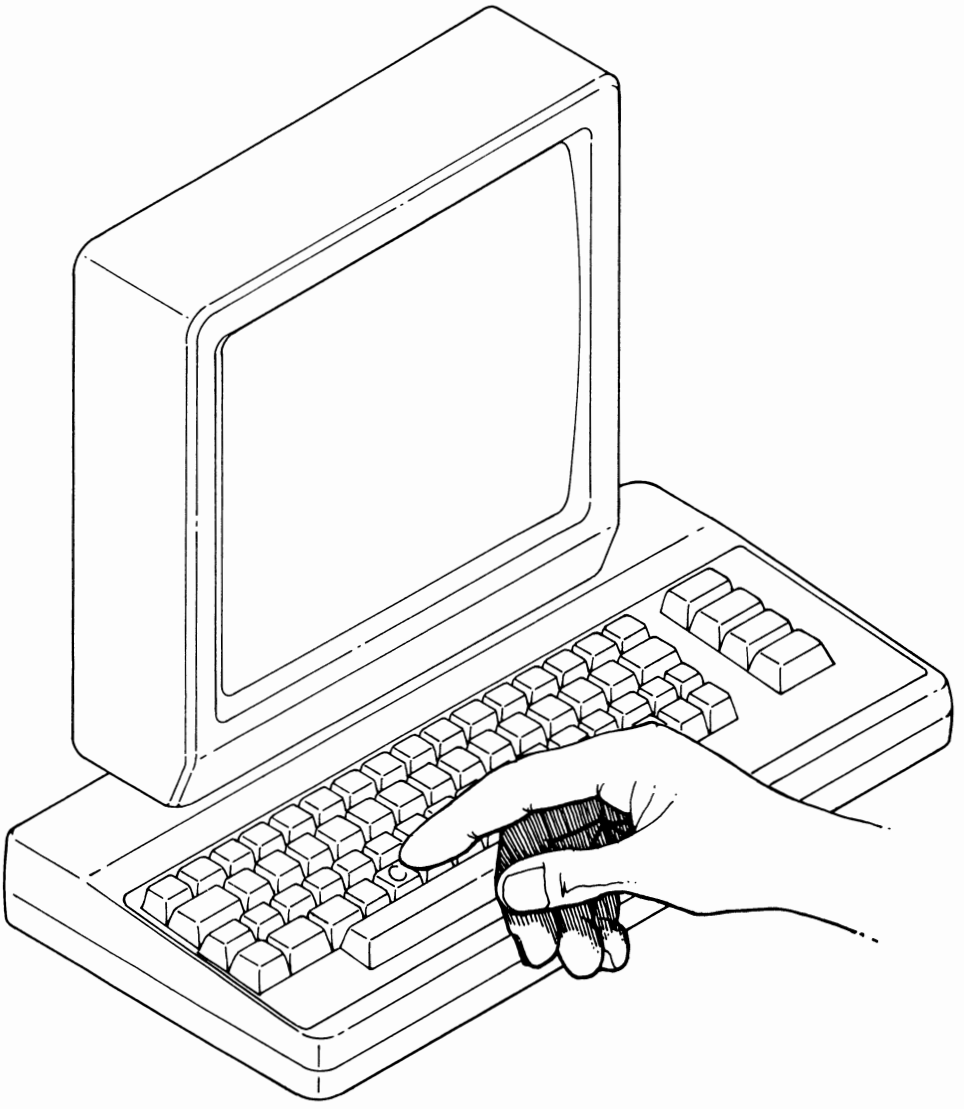
If that is not what is on the screen, turn the computer off, take out any game or program cartridges you may have plugged in, and then turn it on again. If you still have problems, see Box 1-2 in Chapter 1.

Typing Letters and Symbols

Look below the word **READY**. There is a small flashing box. This is called the *cursor,* and it shows you where the next letter you type will appear on the screen.

Press the [C] key, and then release it.

USING THE KEYBOARD



A **C** appears on the screen where the cursor (the little flashing box) was. Also notice that the cursor has moved over one space to show you where the next thing you type will appear.

Did anybody miss the [C] key? Well, you aren't the first. As we go through these exercises, don't worry if you make a mistake. If you press the wrong key, just press the [RETURN] key. Ignore the **?SYNTAX ERROR** business that appears on the screen. Then start over farther down on the screen where the cursor now is. Or delete (erase) the character you just typed by pressing the [INST/DEL] key. (We will discuss correcting *(editing)* your typing in some detail at the end of this chapter.) Above all, don't worry about hurting the computer by typing the wrong thing. *Nothing* you type in can hurt the computer—unless you are typing with an ax. It is possible to type something that will create a bizarre effect. You could even accidentally do something that would prevent you from typing anything else—when you press the [RETURN] key, nothing would happen. Even in that case, all you have to do is turn the computer off and then on again, and you'll be completely back to normal.

Press the [A] key and release it, then press [P], then [S].

The screen should now look like this:

```
**** COMMODORE 64 BASIC V2 ****

 64K RAM SYSTEM  38911 BASIC BYTES FREE

READY.
CAPS■
```

Box 3-1
If You Have Never (or Rarely) Used a Typewriter

You do not need to know how to type to use the Commodore 64. Typing the various commands and instructions, even the fairly long programs you will eventually be writing, is a small part of using your computer. Even if you type very slowly, it will not slow down your overall use of the computer very much.

There are, however, a couple of things you need to know about typing:

1. To make a space between letters, press the *space bar*—the very long key, closest to you.
2. To produce the symbols above the letters (and certain other things you will learn about in this chapter), hold down the [SHIFT] key with one finger, and press the key you want with a different finger. It's a bit awkward, but you get used to it.
3. There are two [SHIFT] keys, one on each side of the keyboard. You can use either. The [SHIFT/LOCK] key, when pressed, stays down until pressed again. It makes the keyboard operate as if you were constantly holding down [SHIFT]. You will rarely need the [SHIFT/LOCK] key.

The [SHIFT] Key

Notice that you automatically get all capital letters.[1] Some keys, however, have two things on the face of the key—for example, the number and punctuation keys. These keys work just as they would on a typewriter. Just press the key to get the lower symbol; to get the upper symbol, hold down the [SHIFT] key while pressing the key. (If you have never used a typewriter, see Box 3-1.) Let's try it:

Press the [4/$] key.

This should display a **4** on the screen.

Now, hold down the [SHIFT] key and press the [4/$] key.

This should display a **$** on the screen. Try some more.

[1] The Commodore 64 *can* produce both lower-case and upper-case (capital) letters. This ability is important because it makes the Commodore 64 capable of word processing. Although you would almost never use lower case in ordinary programming or for any of the things you will learn in this book, if you are curious, you can get into lower-upper case typing by pressing the Commodore key ([◘]) and the [SHIFT] key at the same time. Press these again to get back to normal, all-capital typing.

Press the necessary keys to make the following appear on the screen: #5.>?

```
**** COMMODORE 64 BASIC V2 ****

 64K RAM SYSTEM  38911 BASIC BYTES FREE

READY.
CAPS4$#5.>?■
```

Congratulations. You have just typed a bunch of gibberish—but all in the interest of learning to use the keyboard.

Clearing the Screen

Before going on, let's start with a clean slate:

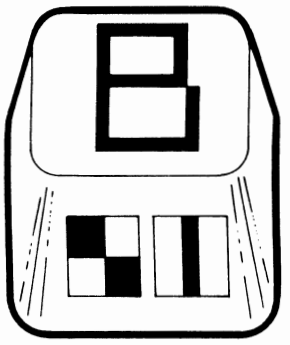
Hold down the [SHIFT] key and press the [CLR/HOME] key.

The screen should now be completely clear except for the cursor (the flashing box) in the upper left corner. This works because "clear" (or CLR as it is abbreviated on the key) is on the upper part of the face of the key. If you had pressed the [CLR/HOME] key without holding down [SHIFT], you would have told the computer to go "home"—which means the cursor would be sent to its home position at the upper left corner of the screen. But that would *not* clear the screen. CLR (that is, [SHIFT] and the [CLR/HOME] key) actually *both* clears the screen *and* sends the cursor home. (By requiring you to use two keys to clear the screen, the Commodore designers have cleverly made it harder to erase a screenful of your typing accidentally.)

Special Graphic Characters

Notice that on the front of most of the keys there are two little box pictures. These are special Commodore 64 *graphic characters* that you can type and make appear on the screen. They are very helpful in making pictures and designs.

SPECIAL GRAPHIC CHARACTERS



Box 3-2
If You Are an Experienced Typist

The keyboard of the Commodore 64 is very much like that of a typewriter. If you are an experienced typist, you will be able to type things into the computer quickly and easily. There are, however, a few differences to watch out for between a computer keyboard and a typewriter. Otherwise, your good typing habits can lead to annoying computer mistakes.

1. The Commodore 64 has a whole extra column of keys on the right. If you find yourself setting your right hand over one row too far, you might put a little piece of tape on the [J] key so you can always feel when you are in the right position.
2. As explained in this chapter, you automatically get capital letters without [SHIFT]. You use the [SHIFT] key for the symbols above the number keys and with the few other keys having two symbols on their surface. But in general, [SHIFT] is used to get the little pictures on the right front of the key. Sometimes, from force of habit in the beginning, you may hold down [SHIFT] to get capital letters. This is easy to see when you look at the screen, but annoying to correct. If you tend to do this, be especially vigilant about the [+] and [L] keys, because the little pictures they produce with [SHIFT] are very similar to a + and an L!
3. The [RETURN] key of the com-

To get the little picture on the right front of the key, hold down the [SHIFT] key and press the key with the picture you want on it. To get the little picture on the left, hold down the Commodore key—the one with the Commodore corporate symbol ([C=]) on it—while pressing the key with the picture on it. (To help you remember, notice that the [C=] key is to the left of the [SHIFT] key.)

Let's use this procedure to make a little boat:

Clear the screen.

While holding down the [C=] key, press the [*] key and then the [I] key twice. Then, hold down the [SHIFT] key and press the [£] key.

It should look like this:

In later chapters, you will learn to use these symbols to create interesting graphics and animation. For now, just enjoy playing with them.

You can also use the keyboard to play with color and "reverse" characters. These are of little use in the actual creation of color graphics, which you will be doing in Part III of this book. Color graphics are definitely not produced by typing designs in color! But if you are interested in typing your letters in color or reverse see Box 3-3.

puter does take you to the start of the next line, just as a typewriter would. There are other ways, however, of starting a new line (such as the arrow keys you will learn about in this chapter). Or, if you type along to the end of a line and continue typing, the computer will go to the new line on its own—a wonderful feature.

Most important, the [RETURN] key has a very special meaning for the computer. It tells the computer to carry out the command you have typed in. Or, if you have typed in what is called a *program line*, [RETURN] tells the computer to enter it into memory. (You will learn all about this in this and the next chapter.) Thus, if at the end of a line you press the [RETURN] key, the computer thinks you are telling it to do something in addition to going to the next line. The result will be an error message if you have not really typed in a proper command, or premature carrying out of your command (if only part of your command has been typed).

4. To the computer, the letter "**O**" and the number "**0**" are two completely different things. In fact, the computer puts a slash through the zero (**Ø**) to help you remember the difference. The lower-case letter "**l**" and the number "**1**" are also very different. The fact that a letter and number may have about the same shape means nothing to the computer. Be sure when a number is called for, you use the number. And when a letter is called for, you use the letter.

Making Corrections—Editing Your Typing

This is one of the most important skills for everything else that follows, so let's see some serious studying in this section!

Clear the screen, then type EDIT AND DELETE

Now, press the [INST/DEL] key 11 times.

The cursor (the flashing box) moves back, *deleting* letters and spaces as it goes along. You should now have simply **EDIT** on the screen.

The face of the [INST/DEL] key says "INST" on the top and "DEL" on the bottom. Thus, you *don't* need to hold down [SHIFT] to delete. We'll see later what "INST" is all about.

By the way, if you hold the [INST/DEL] key down, it automatically repeats. That's handy if you have a lot to delete. But it goes very fast, so be careful!

Now, retype, right where it was before, AND DELETE

Hold down [SHIFT] and press the [CRSR] key 10 times.

Notice the cursor ("CRSR" for short) moves to the left. (If you hadn't held down the [SHIFT] key, it would have gone to the right.) Also notice that the cursor passes over the letters without deleting them or affecting them in any way.

Press the space bar 10 times.

Whatever you type—spaces or anything else—appears wherever the cursor is, covering any letter or other character that was there before.

Move the cursor back again to the right after the I in EDIT (using the right-left cursor-arrow key).

```
EDIT■
```

Type, starting right there, ING TEXT

This should produce **EDITING TEXT** on the screen.

Now, move the cursor to the space between EDITING and TEXT (using the cursor-arrow key), and then use the [INST/DEL] key to delete the ING

Notice that the word **TEXT** moves over to fill up the space you left when you deleted the letters **ING**. The screen should now show **EDIT TEXT**.

Move the cursor onto the X in TEXT

Type an S

The screen now shows **EDIT TEST**.

Move the cursor back to the space between EDIT and TEST

Hold down [SHIFT] and press the [INST/DEL] key four times.

This inserts four spaces, and thus moves the word **TEST** over four places. But the cursor does not move. "INST" stands for "INSERT"—not for "Instant" or anything like that. This poor older brother on the same key with DEL is too often misread and underused.

Beginning where the cursor is (which should be right after the T in EDIT), type OR'S

The screen should now read **EDITOR'S TEST**.

You can also move the cursor up and down, using the [CRSR] key, with or without [SHIFT].

See if you can use the arrow keys (as well as the [INST/DEL] key to fix your mistakes) to make the screen show the following:

```
EDITOR'S TEST
 Ü
```

Box 3-3
Typing in Reverse and Color

It is possible to make the letters you type on the screen come out in color or in reverse (dark letter on light background). The following explains how you can do this.

For reverse:

Hold down the [CTRL] key and press the [9] key. (Notice that the [9] key has "RVS ON"—short for "REVERSE ON"—printed on the front of it.)

Nothing should have happened on the screen when you did this. But the computer is now set to display whatever you type in "reverse." Do you see what we mean?

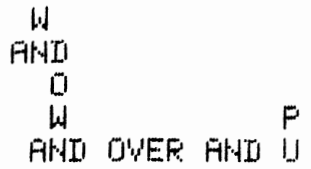
Type YOU

It should look like this on the screen:

```
YOU
```

To get out of reverse typing:

Hold down the [CTRL] key and press the [0] key. (Notice that the [0] key has "RVS OFF" on the front of it.)



Admire your work.

That's it. You have learned to operate the Commodore 64 keyboard. In the next chapter, you will learn how to type things on the screen that will tell the computer to carry out instructions for you.

You're doing very well. Read the Summary and then take a break. We'll meet you again at Chapter 4.

To make letters appear in any of eight different colors, hold down the [CTRL] key while pressing one of the keys numbered [1] to [8]. (Notice that these keys have color names abbreviated on the front of the key.) Then any letter you type will be in that color.

Here, for example, is how you make what you type appear in yellow on the screen:

Clear the screen. Then hold down the [CTRL] key and press the [8] key. (Notice that the [8] key has "YEL" for "YELLOW" on the front of it.) Then release the keys.

Now, type COLOR

COLOR should appear on the screen in yellow. Use the same procedure for any of the other colors abbreviated on the front of the keys.

To get eight *more* colors, use the [1] through [8] keys with the Commodore key [ ] instead of the [CTRL] key. A table of which number keys to press for which colors is on page 57 of the *Commodore 64 User's Guide*. When you want to get back to normal colors, hold down the [RUN/STOP] key and press the [RESTORE] key.

Summary

In this chapter, you see that the Commodore 64 keyboard is similar to a typewriter keyboard, but with a few differences. The Commodore 64 can do many things a typewriter can't.

A little flashing box called the *cursor* shows where the next letter you type will appear on the screen. When you press a key, the letter on the key appears on the screen, and the cursor moves over one place.

Usually, the letters you type will automatically be all capitals. Keys that have two things on their face, (such as the [#/3] key), however, produce on the screen the lower character ordinarily, and the upper character if you hold down the [SHIFT] key at the same time.

To clear the screen (and simultaneously move the cursor to its home position at the upper left corner of the screen), hold down the [SHIFT] key and press the [CLR/HOME] key.

The little pictures on the front of most keys (called *graphic characters*) are produced on the screen as follows: For the left picture, hold down the Commodore key [C=] and press the key with the picture on it. To get the right picture, use the [SHIFT] key.

Tools available for editing (correcting) your typing include the following:

1. The [INST/DEL] key, which moves the cursor backwards (to the left), erasing everything in its path. When you press this same key while holding down [SHIFT], it inserts spaces (to the right), beginning wherever the cursor is located.
2. The ability to type over letters. Whenever you press a key, that letter will appear wherever the cursor is located, even if it means replacing some other letter already there.
3. The right-left and up-down directional cursor-arrow keys, which allow you to move the cursor anywhere on the screen without erasing or otherwise affecting anything that is already typed in.

Terms and Concepts Introduced in Chapter 3

Cursor

How to type letters and symbols

How to clear the screen

How to type graphic characters

How to correct typing mistakes

Cursor-arrow keys

Insert

Delete

Practice Exercises

1. Type the following ballad exactly as written in old English. (Use your correcting skills to fix your mistakes as you go along, and remember not to use the [RETURN] key to go to the next line—use the cursor-arrow keys.)

```
O WHERE HAE YE BEEN,
   LORD RANDAL, MY SON?
O WHERE HAE YE BEEN,
   MY HANDSOME YOUNG MAN?

I HAE BEEN TO THE WILD WOOD;
   MOTHER, MAKE MY BED SOON,
FOR I'M WEARY WI' HUNTING,
   AND FAIN WALD LIE DOWN.
```

2. Now, translate this into modern English. *Edit*—don't retype it. Use the delete, insert, and cursor-arrow keys.
3. Make a design on the screen. It can be anything you like: a fish, a sailor, a sea gull, or a creature from outer space. Keep it simple, and use only the graphic symbols.
4. Practice clearing the screen:

   Type **1**, then clear the screen.

   Type **2**, then clear the screen.

   Type **3**, then clear the screen.

   Type **4**, then clear the screen.

   Type **5**, then clear the screen.

   Type **6**, then clear the screen.

   Type **7**, then clear the screen.

   Type **8**, then clear the screen.

Type **9**, then clear the screen.

Type **THAT'S ENOUGH**, then clear the screen.

5. This exercise is for editing practice (you will have to insert, delete, and type over letters to do this):

   Type **IT IS A FINE DAY**

   Without replacing **A FINE DAY**, make it say:

   IT WAS A FINE DAY

   IT WILL BE A FINE DAY

   IT SHOULD HAVE BEEN A FINE DAY

   IT SHOULD BE A FINE DAY

   IT IS A FINE, FINE DAY

   IT WAS A FINE DAY—BEFORE EDITING!

PART I CHAPTER 4

Giving Orders

In this chapter, you will learn how to give the computer some elementary commands that are not quite functions of "programming" but that will prepare you to begin learning that sacrosanct space-age art in Chapter 5. In this chapter, you will learn commands for arithmetic calculations and commands for displaying your typing on the screen in a variety of interesting ways.

Clear the screen. (Remember how? Hold down [SHIFT] and press [CLR/HOME].)

Type PRINT 40

Notice that the zero (**Ø**) is *not* the letter "oh" (**O**). The computer puts a slash through the zero to help you keep track of the difference.

Press the [RETURN] key.

```
PRINT 40
 40

READY.
■
```

When you pressed the [RETURN] key, the computer carried out your command to print—which means display on the screen—the number **40**. **PRINT** is obviously a very common instruction. Your computer will print what you've commanded on the next available line below the command. Then it tells you it's ready for a new command.

In general, to get the computer to carry out any command, type in the command and then press the [RETURN] key. It's a very important button!

Commanding Your Computer to Do Arithmetic Calculations and to Display the Results on the Screen

You can also make your Commodore 64 do arithmetic problems by using the **PRINT** command.

Type PRINT 40*25

The asterisk (*) is computerese for "times." You do not use the usual "×" for times because the computer would confuse it with the letter "X."

Press the [RETURN] key.

```
PRINT 40*25
 1000

READY.
```

The *result* of this calculation—**1000**—should be displayed on the screen right below the command. Let's try some more.

Type PRINT 40*25*265 and press [RETURN].

The screen should show **265000**. Notice that the computer does not put in a comma. By the same token, if *you* were to put in commas when typing large numbers, you would confuse the computer.

Type PRINT 40/8 and press [RETURN].

The slash (/) means "divide." In the above command, you told the computer to display the result of **40** divided by **8**—which should have given you **5**

Type PRINT 8+40/8 and press [RETURN].

```
PRINT 8+40/8
 13

READY.
```

This should give you **13**, since the computer follows the usual rules of arithmetic by dividing and multiplying before adding and subtracting. (If you don't remember the rules from high school math, see Box 4-1. For those who do a lot of mathematics, see Box 4-2.)

One thing is very important to understand: *you don't need to be a math genius to use a computer.*

Box 4-1
If You Don't Remember the Rules of Arithmetic from High School Math

Here is the order in which calculations are carried out: Multiplication and division are done first, then addition and subtraction. Thus, you calculate $2+3\times4$ by first multiplying 3 by 4, then adding the result to 2. If somebody else went at it by adding 2 and 3 first, he or she would get a different result. What arguments we could have! Instead, everyone follows the above rules of order and everyone always gets the same result.

You *can* change the order, though, by putting some of the calculations in parentheses. Things in parentheses are treated as a unit. Whatever is inside the parentheses is done before any other calculations. Thus, $(2+3)\times4$ now requires first adding 2 and 3, then multiplying the result by 4. This time, 20 is correct.

If you use them, square roots, cube roots, and other "powers" are always done before anything else.

Commanding Your Commodore 64 to Display on the Screen Exactly What You Tell It To

So far, you have learned how to command the computer to do arithmetic and display the answer. But more often you will want the computer to display words or symbols exactly as you have typed them, without any calculations of its own. This is easy to do. Again, you use the **PRINT** command. But this time put quotation marks around what you want displayed on the screen. Everything within those quotes will appear on the next line, uncalculated and untouched!

Type PRINT "40*25" and press [RETURN].

The screen should show this:

```
PRINT "40*25"
40*25

READY.
■
```

Once again, when you use quotes, the computer displays exactly what you told it to instead of giving you the result of a calculation. It will do that for numbers, symbols, letters—anything you put within quotes. (The fancy technical term for the stuff in quotes is a *character string,* since it is a string of characters that get displayed.)

For reasons you will learn about later in the chapter, you will get into trouble if you use the cursor-arrow keys to correct what you've typed within quotes. Until we teach you how to avoid this problem, just use the [INST/DEL] key to make corrections. Or press [RETURN], ignore the **?SYNTAX ERROR** business, and begin retyping.

Type PRINT "CHARACTERS" and press [RETURN].

```
PRINT "CHARACTERS"
CHARACTERS

READY.
■
```

The word **CHARACTERS** should be displayed below the command, just as you requested.

You can also use the **PRINT** command with quotes for the various symbols and graphic characters.

Box 4-2
If You Do a Lot of Math

To take a number to a power, use the up-arrow symbol. Thus, 5 squared is written **5 ↑ 2** and 8.2 cubed is **8.2 ↑ 3**. The cube root of 12 would be **12 ↑ (1/3)**.

There are also several "functions" available on the Commodore 64. For example, **ABS** gives the absolute value (the value disregarding whether a number is positive or negative). **PRINT ABS(5 − 20)** would produce **15**.

Some other functions are:

INT—integer: **PRINT INT(6.342)** gives **6**.

SQR—square root: **PRINT SQR(16)** gives **4**.

SGN—sign: **PRINT SGN(−71.3)** gives **−1** and **PRINT SGN(.067)** gives **1**.

LOG—logarithm (base e): **PRINT LOG(100)** gives **4.60517019**.

EXP—raise e to that power: **PRINT EXP(3)** gives **20.0355369**.

In addition, there are several trigonometry functions:

ATN—arctangent
COS—cosine
SIN—sine
TAN—tangent

Type the following:

```
PRINT "BOAT #1 ▀▄▄▄▀"
```

(Remember, you get the boat figure in Chapter 3 by pressing [C] and [*], then [C] and [I] twice, and finally, by pressing [SHIFT] and [£].)

Press [RETURN].

You should see exactly what you put within quotes displayed just below your command:

```
PRINT "BOAT #1 ▀▄▄▄▀"
BOAT #1 ▀▄▄▄▀

READY.
■
```

Remember, the quotes are essential. If you told the computer to print this without putting it within quotes, it would not understand the instruction. When you pressed [RETURN], it would give you an *error* message. (If, however, you ask it to print a number, such as **PRINT 40**, that works without quotes since it is treated as a calculation. When you "calculate" 40, you get 40!)

If you do get an error message, retype what you have done and fix the error if you can. Leave the error message on the screen. It doesn't hurt anything. If you are curious, there is a list of error messages with their definitions in Appendix F.

There is one other thing to watch out for in typing **PRINT** commands: [RETURN] does not mean "carriage return." Yes, pressing it does result in the cursor's going back to the left edge on a new line. But [RETURN] really means, "Now, carry out my command." Or maybe it will help you remember to think of it as, "Return to work," or "Return to ship." Let us show you something.

Type PRINT "THE COMMODORE 64 SCREEN LINE IS FORTY SPACES LONG."

Then press [RETURN].

```
PRINT "THE COMMODORE 64 SCREEN LINE IS F
ORTY SPACES LONG"
THE COMMODORE 64 SCREEN LINE IS 40 SPACE
S LONG
```

On the Commodore 64, whenever you are typing something and you come to the end of a line, the computer automatically continues on the next line. *You don't have to hit a carriage return,* as you would on a typewriter, to go to the next line. In fact, if you *do* hit the [RETURN] key, the computer will think you have finished your command and will try to display what you have typed up to that point!

Two other points about long instructions: The computer can treat a command of up to a total of 80 spaces as a single instruction, but no more than 80. The word **PRINT** and the quotes take up some of those spaces. So you have to start a new **PRINT** command if you want to print more. We will get to how to write a sequence of **PRINT** commands in Chapter 7.

As you have no doubt noticed, no matter how your material within quotes is broken up on successive lines as you type it in, it looks different when printed, since it begins at the left edge of the screen.

Clear Screen

If, while typing *within* quotes, you press the [SHIFT] and [CLR/HOME] keys to clear the screen, that procedure is carried out only *after* you have pressed [RETURN].

Let's see what happens when you order the screen cleared after you have typed the first quote but not the second:

Type PRINT "

Now, without pressing the [RETURN] key yet or doing anything else:

Hold down the [SHIFT] key and press the [CLR/HOME] key.

This is the usual procedure that you learned in the previous chapter for clearing the screen. But this time the screen does not clear. Instead, ♡, a reverse figure (light printing on dark background) of a heart appears. This special symbol is there only to let you know that the computer is remembering that you have given the clear-screen instruction.

When you close the quotes and press [RETURN], the computer carries out the command to clear the screen.

Type another quote mark.

The screen should now show this:

```
PRINT "♡"
```

Now, press the [RETURN] key.

The screen should now be clear and ready for your next command!

The beauty of this technique is that you can *first* have the computer clear the screen of commands or any other typing, and *then* print what you want displayed:

Type PRINT "♡123GO" (using [SHIFT] and [CLR/HOME] to get the reverse-heart figure).

Remember, you get the reverse-heart figure by doing the clear-screen procedure *before you have closed the quote marks.* At any

other time, holding shift and pressing the [CLR/HOME] key simply clears the screen right then and there!

Press the [RETURN] key.

The screen should have cleared, displayed **123GO** in the upper left corner, and then indicated it was ready for the next command. The computer displayed **123GO** in the upper left corner because after a clear screen, unless you give the computer some special instruction to do otherwise, the computer always starts up again in the home position.

Also, note that you put the reverse heart *before* the **123GO**. Thus, the computer first cleared the screen and then printed the **123GO**. If you put the reverse heart at the end (**PRINT "123GO♡"**), your command would first print **123GO**, then immediately clear the screen. You would end up with a blank screen.

Cursor Movements within Quotes

Occasionally, you want your characters printed somewhere else on the screen. The cursor-arrow keys work much like [CLR/HOME] when typing within quotes. Normally, when you are not typing between quote marks in a **PRINT** command, these keys do just what they say they do. To review, if you press the [CRSR] key four times, the cursor (the flashing box) moves over four spaces to the right. If you press this key while holding down the [SHIFT] key, it moves the cursor to the left. The [CRSR] key works in the same way to move the cursor up or down.

If, however, you press these arrow keys while typing between quote marks in a **PRINT** command, something different happens. Instead of moving the cursor, some funny symbols will appear (for example, a reverse-bracket symbol for the right arrow). Then, after you have put in your closing quote mark and pressed [RETURN] to carry out the command—then, and not until then—the cursor movements are carried out. Let's see how that works.

Type PRINT "

Press the [CRSR] key six times.

Type OVER 6"

The screen should show this:

```
PRINT "▶▶▶▶▶▶OVER6"
```

These reverse-right brackets are the computer's way of showing you that it will remember to move to the right when it carries out your **PRINT** command.

Press [RETURN].

Now, the computer carries out the command and displays **OVER 6** six places to the right of the edge of the screen.

```
PRINT "      OVER6"
      OVER6
```

We won't go into using arrows within quotes in more detail because there are better ways to position what you print, which you will learn later.

Making Corrections within Quotes

The wonderful ability of the Commodore 64 to remember cursor movements in a **PRINT** command does create one little problem: If you make a mistake while typing within quote marks, it's a bit tricky to make corrections.

When you're typing within quote marks, you cannot simply move the cursor back to the place with the problem and correct the error. When you try to move the cursor back, it simply puts in the ■ symbol, right? This is its way of showing you it will remember to move left when it carries out your **PRINT** command. Havoc!

Fortunately, the [INST/DEL] key still works just fine. You can still do your editing with those procedures. But if you have typed a long line and only need to change one letter at the very beginning, it is rather frustrating to have to delete the whole line. There is a better solution: Finish the line you are typing and *put in the quote mark at the end of the line. Then* move your cursor back to the letter you want to change. This is only a little cumbersome, and it works quite well.

Type PRINT "I I SIR"

Using the procedure in the paragraph above, correct this to read PRINT "AYE AYE SIR"

Okay. A very important section is over. Shrug your shoulders and wiggle your toes.

So what have you learned in this chapter? You have learned how to give the computer certain simple but crucial commands. Now, in the *next* chapter, you will see how you can give the computer a whole shopping list of commands to be carried out in order and remembered so it can do them again and again, as many times as you want, any time you want. In other words, in the next chapter, you will begin to learn *programming*!

Summary

In this chapter, you learn the various ways to use the **PRINT** command to give instructions to the computer.

A **PRINT** command can be used to make your computer work like a calculator. You type **PRINT** followed by the calculation you want done. Then you press the [RETURN] key. The result of your calculation is displayed on the screen. The computer uses the usual + and − symbols, but uses * for multiply and / for divide. The Commodore 64 carries out calculations in the standard arithmetic order of multiplication and division first, then addition and subtraction.

A **PRINT** command can also be used to make the computer display exactly what you have typed without any calculations, including letters and all the various graphic symbols. You type **PRINT** and then what you want displayed *within quote marks*. For example, **PRINT "CHARACTERS"** would result (after you press the [RETURN] key, of course) in the word **CHARACTERS** appearing on the screen just below your command.

A **PRINT** command can be up to 80 spaces long. You must be careful when typing a long command not to press [RETURN] as if it meant carriage return, because it doesn't. It means, "carry out my command," and the computer will think you've completed your command. Instead, keep typing when you come to the end of a line. The computer goes automatically to the next line. Press [RETURN] only when you are finished with your instruction.

When giving a command to print something, you can arrange first for the computer to clear the screen. Type **PRINT** and a quote mark, then [SHIFT] and [CLR/HOME]—the same procedure you learned in the previous chapter for clearing the screen. Instead of immediately clearing the screen, a special reverse-heart symbol ♡ appears. It tells you the computer has registered your command and will indeed clear the screen in carrying out your **PRINT** command.

You can also arrange to have what you print displayed some place other than the usual left edge of the screen immediately below your command. To do this, you follow a procedure similar to that for the clear screen. You use the cursor-arrow right-left and up-down directional keys. Instead of moving the cursor (the flashing box) right then and there, the computer makes those moves only when the entire command is carried out. But this procedure is rarely used.

The ability of the Commodore 64 to remember cursor movements within quotes creates a complication when making corrections within quotes. If you move the cursor to correct something, instead of moving the cursor, the computer puts in a marker to remember the cursor movement when the entire command is carried out. Thus, if you make a mistake within quotes, don't use the cursor-arrow keys. Use the [DEL/INST] key, or finish the command and then go back and make the change.

Terms and Concepts Introduced in Chapter 4

PRINT
Use of the [RETURN] key
Command
How to display the result of arithmetic calculations
+ − * /
Order of arithmetic calculations
Character string
How to display a character string
Clearing the screen within quotes
PRINT "♡"
Reverse-heart figure
Cursor-movement arrow key presses within quotes
Editing within quotes

Practice Exercises

(Answers and Comments in Appendix A.)

1. Make the computer do the following arithmetic:

a. 198 + 2Ø7
b. 123.7 − 1ØØ
c. 1,789 + 34
d. 1ØØ × 54
e. 547 × 67 × 32.19
f. $132.8Ø divided by 4

2. Have the computer print the following:

a. **1 TO GET READY**
b. **THE WORLD IS TOO MUCH WITH US; LATE AND SOON,**

3. Have the computer display the following pictures:

a. A small boat
b. A bridge

4. Redo the above (3a and 3b), but this time have the computer clear the screen before it displays these pictures.

5. Type: **PRINT "HOW ARE YOU"**
Edit it to read as follows:
PRINT "HOW WERE YOU"

```
PRINT "HOW WILL YOU BE"
PRINT "HOW WERE YOU DOING"
```

PART I CHAPTER 5

Introduction to Programming

This chapter introduces the basic principles of computer programming. In less than an hour, you will be a computer programmer. A novice, yes—but you'll be able to say, "Some computer programming experience" on your resumé.

A program is a series of instructions that the computer remembers and, when asked to do so, carries out in a predetermined order. That's all it is. Programming is as simple or as complicated as the instructions you want carried out. It need not be difficult at all.

Giving Single Commands Versus Writing a Program

In the last chapter, you learned how to use the very versatile **PRINT** command to tell the computer to do calculations for you or to display on the screen whatever words or characters you desire. All you had to do was type the command and press the [RETURN] key. Your Commodore 64 carried out the order. This procedure is called using the computer in *command mode* because you give a command and then ask the computer to carry it out.

Command mode has two major shortcomings. First, once the computer has carried out the command, the command is no longer available to be carried out again. In command mode, the computer does not *remember* a command once it is carried out.

Second, in command mode the computer can carry out a series of commands only if you give it one command, press [RETURN], wait for it to be carried out, then type in another command, press [RETURN], wait for it to be carried out, and so on.[1]

The real power of a computer is precisely in its ability to remember, and carry out whenever asked, a whole series of instructions. That power is *the* power of computer programming.

[1] Actually, there is a way in command mode to get the computer to do a few instructions in a series. You put a colon (:) between commands. Then when you press [RETURN], the whole series of instructions is carried out in the order written. But the entire series of instructions can't be longer than 80 spaces. Also, you cannot change the order of instructions according to what happens during the program—something that, you will learn, is a very important aspect of programming.

Numbered Program Lines and the RUN and LIST Commands

To make the computer operate in *program mode,* all you do is put a number before each of your instructions. It's that simple!

Clear the screen, then type 70 PRINT "I CAN PROGRAM"

Press [RETURN].

When you pressed [RETURN], nothing appeared to happen, except that the cursor went to the beginning of the next line.

But something very important *did* happen. When you type a line beginning with a number—a "numbered" or "program" line—and then press the [RETURN] key, the computer enters that line into its memory. The line sits there waiting for you to tell the computer to carry out the commands in the numbered lines in its memory. You do that by giving the **RUN** command. Try it:

Type RUN and press [RETURN].

Now, the computer carries out your instruction.

(If the computer did not carry out your instruction, it probably said **?SYNTAX ERROR IN 70**. That means you made a mistake—most likely a typographical error. Retype the line and try again.)

What exactly happens when you give the **RUN** command? The computer searches its memory for numbered lines. It determines which line has the lowest number at the beginning of it. (In this case, there was only one number, anyway.) Then it carries out whatever that line asks to be done. Next, it looks for the line with the next lowest number at the beginning of it. If it finds another line, it carries out that instruction. Then it searches for the line with the next lowest number, and so on, until it runs out of numbered lines. At that point, it lets you know it is done by displaying **READY** on the screen.

In the present case, when you gave the **RUN** command (and then pressed the [RETURN] key), the computer found only one line in its memory—line number **70**. Of course, it was the line with the lowest number. The computer carried out that instruction and printed on the screen the phrase you asked for. Having carried out that instruction, it searched for more lines. Finding none, it let you know it was finished by displaying **READY**.

Your Commodore 64 did all this in the blink of an eye. All you had to do was type **RUN** and press [RETURN]. Most important, even though the command has been carried out, *it is still remembered by the computer.* That means you can ask for the command to be carried out again without having to type it again. Just type **RUN** again, press [RETURN], and the computer carries out ("runs") your one-line program again. Try it:

Type RUN and press [RETURN].

It does it again. In fact, you don't even have to have the command on the screen.

Clear the screen (hold down [SHIFT] and press the [CLR/HOME] key).

Now, type RUN and press [RETURN].

It works. Now, for another trick. If you want to see your program at any time, use the **LIST** command:

Type LIST and press [RETURN].

```
LIST

70 PRINT "I CAN PROGRAM"
READY.
■
```

Your entire program (the one line, numbered **70**) appears on the screen right below the **LIST** command.

Let's review all this again because it is the foundation on which your future as a computer programmer will be built. So far, we have seen that when you put a number before an instruction and then press [RETURN], the computer does not carry out the instruction immediately (as the computer would in command mode). Instead, the computer enters the numbered line into its memory. It won't carry out the instruction until you tell it to by commanding it to run your program. (You run your program by typing the word **RUN** and then pressing [RETURN].) We also have seen that once a numbered line has been entered into memory, it stays there. Even after you have run it, it waits in memory so you can call on it as many times as you like. Even if the numbered command is not on the screen, you can carry it out with the **RUN** command. If you want to look at all the numbered lines you have put into memory, simply type the **LIST** command (and press [RETURN] of course), and a list of your numbered lines appears!

Adding a Second Program Line, and the GOTO Instruction

As we said at the start of this chapter, the ability of the computer to remember instructions after they have been carried out is one main advantage of program mode. An even more important advantage is that you can give the computer a whole series of instructions and it remembers and carries out all of them, in the order specified by the line numbers! Let's try it by adding another numbered line to what you already have in memory.

Type 90 GOTO 70 and press [RETURN].

Now, look at your entire program.

Type LIST and press [RETURN].

Just below where you typed **LIST**, the computer should display your two-line program. It should look like this:

```
70 PRINT "I CAN PROGRAM"
90 GOTO 70
```

Let's analyze what the computer does when you ask it to run this little program. First, it searches in its memory for all the numbered lines it can find. It finds two—your lines **70** and **90**. Then it decides which one begins with the lower number. That's line **70**. It then carries out the instruction in that program line—it prints the words **I CAN PROGRAM** on the first unused place going down the screen.

Having carried out the instruction on the lowest numbered program line, the computer looks for the line with the next lowest number. That's line **90**. It then carries out line **90**.

The **GOTO** instruction in line **90** means exactly what it says—go to the line in the program that begins with the specified number. Line **90** tells the computer to GO TO line **70**. The computer dutifully goes back to line **70**, where it carries out the instruction on that line. That instruction is to print the words **I CAN PROGRAM** on the first available line on the screen—which will be one line farther down from the last time it printed those words. So, at this point (just an instant), the screen shows **I CAN PROGRAM** twice, one under the other.

Having carried out line **70** again, the computer once more searches for the line with the next lowest number after the one it has just done. Sure enough, it finds line **90** again. So it does what line **90** says to do—**GOTO 70**. At line **70**, it does the line **70** instruction again. Then it again searches, goes to **90**, to **70**, does it, goes to **90**, and so on.

The computer is in an *endless loop*. (Computer people would call this a case of an *infinite* ***GOTO*** *loop*.) The computer will never get bored with this.

Of course, all of this happens incredibly quickly. When you run this program, the words **I CAN PROGRAM** appear in a column going down the screen. When the screen is full, the computer moves everything above upwards off the screen, one line at a time. At that point, it looks as if it's just flashing at the bottom, trying to keep going. Actually, it is such a workaholic that it is still printing **I CAN PROGRAM** on the bottom line and shoving the top line off the screen.

Let's try it. While you are watching it, think about what is actually happening inside the computer to produce this screen activity.

Type RUN and press [RETURN].

Stopping a Program Midway and Starting It Again—The [RUN/STOP] Key and the CONT Command

After you have watched and thought about this for a while, you're probably wondering how to stop this whole activity. Even if the computer is not bored, you may be! There is a procedure to stop a program while it is in the process of being carried out.

Press the [RUN/STOP] key.

```
I CAN PROGRAM
I CAN PROGRAM
I CAN PROGRAM
I CAN PROGRAM
I CAN PROGRAM
I CAN PROGRAM
I CAN PROGRAM
I CAN PROGRAM
I CAN PROGRAM
I CAN PROGRAM
I CAN PROGRAM
I CAN PROGRAM
I CAN PROGRAM
I CAN PROGRAM
I CAN PROGRAM
I CAN PROGRAM
I CAN PROGRAM
I CAN PROGRAM
I CAN PROGRAM
I CAN PROGRAM
I CAN PROGRAM
BREAK IN 70
READY.
■
```

This stops the program dead in its tracks. The computer says either **BREAK IN 70** or **BREAK IN 90** (depending on which line the computer was carrying out when you stopped it), and **READY** Occasionally, as we will see later, this procedure will not stop the program, but usually it works just fine.

The program is still in memory. You could start it again, from the beginning, by using the **RUN** procedure. If, however, you stop a program with the [RUN/STOP] key, you can also make it pick up where it left off by using the **CONT** (for "continue") command.

Type CONT and press [RETURN].

Since there are only two lines to the program, **CONT** and **RUN** give about the same result. But if you were in the middle of a 50-line program and stopped it, it would make a lot of difference. **RUN** would start it up again at the start, beginning with the lowest numbered line in the program. But **CONT** would start it up at exactly that point in the program where the computer was when you stopped it.

Press the [RUN/STOP] key again.

The program should stop running again.

To review a little: We have seen it is easy to write a program. For *each* instruction, simply put a number before it, then enter it into memory by pressing the [RETURN] key. You press [RETURN] to enter each numbered line of instruction. To carry out your instructions, type **RUN** and press [RETURN]. To see what you have put into memory, type **LIST** and press [RETURN]. To stop a program while it's being carried out, press the [RUN/STOP] key. To start it up again where it left off, type **CONT** and press [RETURN]. You have now also learned five very important words of the BASIC computer language: **PRINT**, **GOTO**, **RUN**, **LIST**, and **CONT**.

Editing Program Lines

This is another *very* important section.

You can at any time make changes on a program line, even after you have entered it into memory. Just correct the line and press [RETURN]. Even if you have already listed or run the program, the changes apply the next time you list or run the program.

In Chapter 3, you learned how to edit what you typed by using the right-left and up-down cursor-arrow keys, along with the [INST/DEL] key. In Chapter 4, you learned the special considerations for editing what you have typed within quotes in a **PRINT** command.

Editing a numbered program line follows exactly the same rules, with one additional point. After making a change in a program line, you must press the [RETURN] key. And you must press the [RETURN] key *while the cursor is on the screen line containing the numbered program line you have corrected.* The cursor can be *anywhere* on the line. You can come back to the line later, while it is still on the screen, and press [RETURN]. But you must, at some point, press [RETURN] with the cursor on that line. If you remember to press [RETURN], you will see the new version when you list the program. But if, after making a change, you forget to press [RETURN], then even though the program line looked changed on the screen, you will see the old version is still in memory if you list your program.

Clear the screen and type LIST

Change line 70 to 70 PRINT "I CAN EDIT PROGRAMS"

Press [RETURN] while still on that line, and move the cursor down past where it says READY. Type LIST and press [RETURN].

The edited version of the program should appear.

Now, change line 70 back to its original form.

```
70 PRINT "I CAN PROGRAM"
```

(Incidentally, once you have pressed [RETURN] on a program line, or if it has just been listed, or even if you have just closed quotes, you can use the cursor controls within quotes without problems, as long as you do not type a new quote mark.)

Move the cursor down past where it says READY, without pressing [RETURN]. Then type LIST and press [RETURN].

This time the listing will *not* show your corrections!

Move the cursor up to the earlier listing, where you corrected line 70 back to the original, and press [RETURN] on that line.

This should work.

There is another way to edit program lines. You can type a new version of the line, anywhere on the screen, using the same number for it. Then press [RETURN]. The new version replaces in memory the old one with the same number.

Type 70 PRINT "NEW VERSION"

Press [RETURN], type LIST, and press [RETURN].

The new line **70** should have replaced the old one in the new listing.

If you want to completely remove an entire numbered line from a program (not just change every word), there are two ways to do it. One way is to move your cursor to where that program line is listed on the screen and delete everything typed on the line *except* the line number. Then press [RETURN] while still on that screen line. (If you delete the line number too, when you press [RETURN] the computer would not know which line to eliminate!) Since a line with only a line number is of no value to the computer, it deletes the line completely from its memory.

Move the cursor to the screen line that has program line 90 listed on it, delete everything on the line except the line number, and press [RETURN].

Type LIST and press [RETURN].

The new listing does not include line 90.

Can you guess the second way to eliminate a line completely? Type in the line number by itself at some new place on the screen and press [RETURN]. This tells the computer to replace the old version with the new one, but, again, since the new version is simply a line number with nothing after it, it is eliminated.

Type 70 and press [RETURN].

Type LIST and press [RETURN].

Since you have now eliminated both lines of your program, the computer has nothing left to list. When you ask it to do so, it leaves a blank line and tells you it is ready for your next instruction.

One other thing: If you try running a program and it stops midway, the computer will tell you the line number with the problem. Usually, you will find the mistake right away. If, however, the error message, as such things are called, is not clear (or just makes you curious), *see Appendix F for a detailed explanation of it*. Also, we discuss the whole topic of correcting errors in Chapter 17. You are welcome to skip ahead to look over that section.

Terrific! Today, you wrote your first program! Congratulations!

In the next chapter, you will be introduced to more of the fundamental vocabulary of the BASIC computer language.

Summary

A program is a series of instructions that the computer remembers and, when asked to do so, carries out in a predetermined order.

In Chapter 4, you learned to give a command by typing an instruction (such as **PRINT "CHARACTERS"**) and then pressing the [RETURN] key. That method of telling the computer what to do is called operating in *command mode*. Command mode has two limitations: A command is not remembered once it is carried out, and for the most part you can give only one command at a time.

The real power of a computer is its ability to operate in *program mode*. To do this, you put a number before your command (for example, **70 PRINT "I CAN PROGRAM"**) and then press the [RETURN] key. That puts the numbered line into memory. To see all the numbered lines you have put into memory, type **LIST** and press [RETURN]. When you want the computer to carry out your instruction, type **RUN** and press [RETURN].

The computer can remember as many numbered lines as you can think of (within its memory limitations). They are all kept in memory, even after you have carried out the program or cleared it from the screen. When you type **RUN** and press [RETURN], the program lines are located and carried out in the order of their line numbers (from lowest to highest).

The **GOTO** instructions tells the computer to GO TO a particular program line. For example, suppose you have asked the computer to remember the two lines **70 PRINT "I CAN PROGRAM"** and **90 GOTO 70**. When you run your program, the computer first carries out the instruction in the lowest numbered line in its memory (line **70**); then it goes to the line with the next lowest number (line **90**). In this case, **90** calls for the computer to go back to line **70**—which the computer does. After carrying out line **70** again, the computer then comes to line **90** again, goes back to line **70** again, and so on, in an endless **GOTO** loop.

You can usually stop the computer in the middle of running a program (for example, when it is in an endless loop) by pressing the [RUN/STOP] key. To start the program going again, either use the usual **RUN** procedure, which still start it over again from the beginning, or type **CONT** and press [RETURN], and it will continue from where it left off.

You can make changes in numbered program lines at any time, even after you have run or listed them. Simply use the editing procedures learned in previous chapters, and then press [RETURN] while the cursor is still on the line you have changed. Another way of editing program lines is simply to type the new version and press [RETURN]. The new version replaces the old one in memory.

You can completely eliminate a program line either by deleting all of the line except the line number and pressing [RETURN], or by typing just the line number and pressing [RETURN].

Terms and Concepts Introduced in Chapter 5

Command mode

Program mode

Numbered line or numbered program line

Entering a numbered program line into memory

LIST

RUN

GOTO

GOTO loop

[RUN/STOP] key

CONT

Editing numbered program lines

Practice Exercises

(Answers and comments in Appendix A.)

1. Write a program of one line to make the words **ONE-LINE PROGRAM** appear on the screen.
2. Clear the screen.
3. List your program.
4. Clear the screen.
5. Run your program.
6. Edit your program so it produces on the screen **2 PROGRAM LINES**.
7. List your program.
8. Run your program.
9. Add a new program line that makes the program repeat itself endlessly.
10. List your program.
11. Run your program.
12. Stop the program while it is running.
13. Make it start up again without using **RUN**.

PART I CHAPTER 6

A BASIC Vocabulary

In a way, this chapter is the heart of the book, so stretch, take a Vitamin C, kick the wastebasket—do whatever you do to get yourself all fired up—and we're on our way.

In the last chapter, you learned the fundamental principles of programming, along with a few of the most important words in the BASIC computer language. This chapter introduces a few more BASIC terms you need in order to have an adequate core vocabulary for writing interesting programs: **REM**, **FOR . . . TO . . .**, **STEP . . .**, **NEXT**, **IF . . . THEN . . .**, **INPUT**, **SAVE**, **VERIFY**, and **LOAD**.

Although there is more to know about programming and the BASIC language (you will learn many of these fine points in later chapters), most of the useful and fun things you'll want to do with a computer can be done with what you will have learned by the time you finish this chapter.

Before starting the new material, first retype the two-line program of the last chapter:

```
70 PRINT "I CAN PROGRAM"
90 GOTO 70
```

List your program (using LIST).

Be sure to press [RETURN] after typing in each program line and the **LIST** command. *Whenever you want a program line entered into memory, or whenever you give a command (such as* **RUN** *or* **LIST***), you must press the* [RETURN] *key.* From now on, we will assume you will remember to do this. We will therefore no longer remind you to press [RETURN]. Don't forget: Very little happens on the Commodore 64 until you press [RETURN].

Making Notes to Yourself—the REM Instruction

Type 10 REM PRACTICE PROGRAM

(Did you remember to press [RETURN]?)

After you have written a program, it can be hard to figure out later what you were trying to do, especially if it was a long or

complicated program. You may not even remember its purpose. It's handy to put little notes to yourself at various points in the program to help you figure out later what you were doing. The **REM** instruction allows you to do this.

REM stands for "remark." After the word **REM**, you can type anything you like (up to a total of 80 spaces for the whole numbered line, of course). When a program is being run and the computer comes to a **REM** line, it ignores it! A **REM** line does absolutely nothing. It just sits there in your program. Whenever you list the program, the **REM** line appears, reminding you of whatever you wanted to be reminded of. It is actually a very handy thing. We strongly advise you to use a lot of **REM** lines.[1]

Now let's see how **REM** looks in a program.

List your program.

```
10 REM PRACTICE PROGRAM
70 PRINT "I CAN PROGRAM"
90 GOTO 70
```

The three-line program you now have works exactly the same as the two-line one you had before. When you run it, the computer searches for all program lines in memory. This time, it finds three. First going to the lowest numbered one, the computer attempts to carry out line **10**. As soon as it sees that line **10** is a **REM** instruction, it ignores it and goes on to the next line, your **PRINT** instruction. You know what happens after that: It prints as instructed, goes to the next lowest numbered line (line **90**), goes back to **70** as instructed, and so on, until you tell it to stop by pressing the [RUN/STOP] key.

Run the program.

After a little while, press the [RUN/STOP] key.

Now, list the program again.

See? Your **REM** line is still there!

Some Notes on Numbering Program Lines

You may have noticed that even though your **REM** line was the last numbered line you entered into memory, it was listed first. That's because it has the lowest number. No matter what order you write your numbered lines in, they are always listed (and carried out) in their numerical order.

[1]The only reason not to use a lot of **REM** lines is that they take up room in memory. But on the Commodore 64, you have enough memory for just about any home or small business use.

Also, you may have wondered why we didn't just number the lines **1**, **2**, **3**, and so forth, instead of using numbers like **10**, **70**, and **90**. The reason is that, when you write a program, you often discover part of the way through that you left out an instruction. Then you want to stick it between two earlier lines. If, however, you have numbered the lines **1**, **2**, **3**, and so on, you have to renumber—a big hassle. Instead, we leave lots of room between numbers. In fact, some programmers use **100**, **200**, **300**, and so on, to be sure they have left lots of room for additions later.

Taking Charge of Your Loops—FOR . . . TO . . . and NEXT, STEP . . . , and Variables

Most BASIC terms are easy to understand and remember from the meaning of the English word. **GOTO** and **RUN** are very straightforward. In all of BASIC, there is only one important procedure that is peculiar—the **FOR . . . TO . . .** and **NEXT** procedure. Although this takes a little more concentration to grasp, it is not difficult. And, since it is a very powerful programming tool, it is worth the trouble. Let's get started.

Type the following new program lines:

```
50 FOR X=1 TO 10
80 PRINT X
90 NEXT X
```

List your program.

```
10 REM PRACTICE PROGRAM
50 FOR X=1 TO 10
70 PRINT "I CAN PROGRAM"
80 PRINT X
90 NEXT X
```

You should now have a five-line program. (Notice, incidentally, that in the listing, the new line **90** you typed replaces the old line of the same number.)

When you run this program, what happens? First, the computer comes to line **10**—a **REM** instruction, which it keeps remembering but does nothing about. It then comes to your new line—**50 FOR X=1 TO 10**. When the computer encounters this instruction *for the first time,* it sets **X** equal to **1**.

X is just a letter that stands for a number. You could have used **Y** or **J** or **A** or even several letters together—**AB**, **SM**, **NAME**, **HELLO**, **GOODBYE**, or whatever—and it would work the same way.

A letter that stands for a number is called a *variable.* Some people make a big deal about variables, and if you like fine mathematical details, it is indeed an interesting concept. But for practical purposes, it is really very simple. A variable is a letter or name that stands for a number. It's possible to change the number it stands for (you'll see how later on). In other words, the number it stands for *varies*—which is why it is called a variable. That's simple enough.

```
10 REM PRACTICE PROGRAM
50 FOR X=1 TO 10
70 PRINT "I CAN PROGRAM"
80 PRINT X
90 NEXT X
```

What the computer does with these variables is also very simple. It sets up a special place in memory for each variable you use. It then puts in that place whatever number you want the variable to be. If you later change the number you want that variable to be, the computer puts the new number into that place in memory and forgets the old number.

So, when the computer gets to line **50**, it sets **X** equal to **1**. At this point, nothing has happened on the screen—so far, there has been no **PRINT** instruction. The computer just makes a note to itself that **X** equals **1** and then proceeds to the next lowest-numbered line. That line is good old line **70**, whereupon the computer dutifully prints out **I CAN PROGRAM**.

Then comes **80 PRINT X**. Notice that there are no quote marks around **X**. If **X** were within quotes, the computer would just print the letter **X**. *Without quotes,* **X** *is treated as a number.* The computer searches its memory to see what number the variable **X** has been set equal to. (If you had not set **X** equal to anything, the computer would assume it equals zero!) Since line **50** set **X** equal to **1**, at line **80** the computer prints **1**.

Then comes **90 NEXT X**. *Whenever there is a* **FOR . . . TO . . .** *line in a program, there must always be a* **NEXT** *instruction somewhere later on.* They work together.

What happens is this: After the line with the **FOR . . . TO . . .** in it, the computer carries out all the lines that follow it—*until* it comes to a **NEXT** instruction. Then, instead of going on, it goes directly back up to the **FOR . . . TO . . .** line, where it sets **X** (or whatever variable you named in the **FOR . . . TO . . .** line) *one number higher.* In the present case, when the computer gets to line **90**, it immediately goes back to line **50**, sets **X** equal to 2, and goes on.

The computer goes to line **70** and again prints **I CAN PROGRAM**. So, at this point (after maybe a millisecond or so), the screen would show three lines after the **RUN** command:

```
I CAN PROGRAM
 1
I CAN PROGRAM
```

The computer comes again to **80 PRINT X**. This time, **X** equals **2**. So, the computer prints **2**.

The next program line is **90 NEXT X**. When the computer gets here, it again goes back to the **FOR . . . TO . . .** instruction and sets **X** one number higher still—this time to **3**.

The computer then goes through the lines until it again comes to line **90**, where it again goes back to line **50**, again sets **X** one number higher, and then again proceeds to the lines that follow. And so on, until it has gone through this **FOR . . . TO . . .** and **NEXT** loop so many times that **X** is now equal to **10**.

Now, the computer dutifully carries out all the lines after line **50** one last time. This time, when it gets to **90 NEXT X**, it does *not* go back to line **50**. Instead, it goes on to the next line in numerical order, if there is one. If there isn't, the computer realizes it is finished and says **READY.**

The computer stops going through this loop when **X** becomes equal to **10** because the **FOR . . . TO . . .** instruction set **10** as the limit. The line said **50 FOR X=1 TO 10**. If it had said **50 FOR X=1 TO 23**, then the computer would have kept going through this loop until **X** equaled **23**.

Run the program.

The screen should look like this:

```
I CAN PROGRAM
 1
I CAN PROGRAM
 2
I CAN PROGRAM
 3
I CAN PROGRAM
 4
I CAN PROGRAM
 5
I CAN PROGRAM
 6
I CAN PROGRAM
 8
I CAN PROGRAM
 9
I CAN PROGRAM
 10
```

Notice this instruction differs from an endless **GOTO** loop. With the **FOR . . . TO . . .** and **NEXT** loop, we go through the loop exactly the number of times specified (in this case **10**)—no more, no less. Each time, the computer automatically increases the value of **X** by **1**.

The Variable in a FOR . . . TO . . . and NEXT Loop

Let's play with this procedure a little to be sure you understand how it works and to see some of its possibilities. First, let's change the variable name.

Edit lines 50, 80, and 90 as shown:

```
50 FOR Z=1 TO 10
80 PRINT Z
90 NEXT Z
```

List your program.

```
10 REM PRACTICE PROGRAM
50 FOR Z=1 TO 10
70 PRINT "I CAN PROGRAM"
80 PRINT Z
90 NEXT Z
```

This program is exactly the same, except that instead of the variable being **X**, it is now **Z**. Notice that you had to change the variable name every place you used it.

Run the program.

Naturally, you get the same results as the last time.

The Numbers in a FOR . . . TO . . . and NEXT Loop

What happens when you change the numbers?

Edit line 50 to read 50 FOR Z=1 TO 6

List your program.

This time, all you've changed is the number after the **TO**. This should result in the computer going through the **FOR . . . TO . . .** and **NEXT** loop only six times, with **Z** starting at **1** the first time and going up until it gets to **6** the last time.

Run the program.

Also, you don't have to start with **1**.

Edit line 50 to read 50 FOR Z=5 TO 8

List and then run the program.

Using STEP in a FOR . . . TO . . . and NEXT Loop

You can arrange to have the variable increase each time by some amount other than **1**. To do that, you tell the computer to go up in *steps* of something other than **1**.

Edit line 50 to read 50 FOR Z=1 TO 9 STEP 2

List your program.

This instruction tells the computer to make **Z** equal **1** the first time (as usual). The second time, however, it increases **Z** not by the usual **1**, but by **2**. So **Z** is set at **3**! The next time, it goes up another step of **2**, bringing it to **5**, then **7**, and finally **9**. Try it:

Run the program.

You can have the numbers go down:

Edit line 50 again to read 50 FOR Z=9 TO 1 STEP −2

List your program.

This time, the computer starts by setting **Z** at **9**. It then proceeds *down* by steps of **2**. It goes from **9**, to **7**, to **5**, to **3**, and finally to **1**.

Run the program.

```
I CAN PROGRAM
 9
I CAN PROGRAM
 7
I CAN PROGRAM
 5
I CAN PROGRAM
 3
I CAN PROGRAM
 1
```

You can even go up or down using decimal fractions:

Edit line 50 to read 50 for Z=1 TO 4 STEP .5

List and run your program.

```
10 REM PRACTICE PROGRAM
50 FOR Z=1 TO 4 STEP .5
70 PRINT "I CAN PROGRAM"
80 PRINT Z
90 NEXT Z
```

```
I CAN PROGRAM
 1
I CAN PROGRAM
 1.5
I CAN PROGRAM
 2
I CAN PROGRAM
 2.5
I CAN PROGRAM
 3
I CAN PROGRAM
 3.5
I CAN PROGRAM
 4
```

The **FOR . . . TO . . .** and **NEXT** procedure is really always a **FOR . . . TO . . . STEP . . .** and **NEXT** procedure. It's just that if you don't put in the step, the computer assumes that you want **STEP 1**.

The Computer as a Decision Maker—The IF . . . THEN . . . Instruction

One of the most powerful abilities of your Commodore 64 is decision making. The main tool for this is the **IF . . . THEN . . .** instruction. It is very easy to use and does just what it says it does.

Edit lines 50 and 60 as shown:

```
50 FOR Z=1 TO 10
60 IF Z=7 THEN GOTO 90
```

List your program.

```
10 REM PRACTICE PROGRAM
50 FOR Z=1 TO 10
60 IF Z=7 THEN GOTO 90
70 PRINT "I CAN PROGRAM"
80 PRINT Z
90 NEXT Z
```

You now have a six-line program. When you run this program, what does the computer do? It starts as usual. It comes to line **10**—the **REM** line—and passes right on by. It then comes to line **50**. Since this is the first time through, it sets **Z** equal to **1**.

Then, when it gets to line **60**, it looks at the comparison after the **IF**. It asks, "Does **Z=7**?"

The general rule is that if the comparison (in this case an equation) is true, the computer does whatever comes after the **THEN**. If the comparison is false, it does *not* do the thing after the **THEN**. It simply goes on to the next program line.

In the present case, the first time through, **Z** equals **1**. Accordingly, the comparison **Z=7** is not true. **Z** does *not* equal **7**. Therefore, the computer does *not* do what comes after the **THEN**. Instead, it goes on to the next program line as if the **IF . . . THEN . . .** had not been there at all.

As the computer proceeds through the program, it carries out lines **70** and **80**, displaying on the screen the phrase **I CAN PROGRAM** and then on the next line the number **1**. When it comes to line **90**, it shoots back up to line **50**, sets **Z** at **2**, and comes again to line **60**.

Again, the comparison is not true. The computer goes on. It continues to go on until it has gone through the **FOR . . . TO . . .** and **NEXT** loop enough times for **Z** to equal **7**. Now, when the computer gets to line **60**, the comparison is true. The computer looks at what comes after the **THEN** and carries it out. In this case, it is told to **GOTO 90**. So it does. The computer skips all the lines between **60** and **90**. As a result, you see **I CAN PROGRAM** one less time. (You are probably getting tired of the phrase by now, anyway.) For the same reason, the number **7** doesn't get printed.

The next time through the loop, **Z** equals **8**. It is not true that **Z=7**, and the computer follows the usual procedure of going directly to the next line. And so on.

Run the program.

```
I CAN PROGRAM
 1
I CAN PROGRAM
 2
I CAN PROGRAM
 3
I CAN PROGRAM
 4
I CAN PROGRAM
 5
I CAN PROGRAM
 6
I CAN PROGRAM
 8
I CAN PROGRAM
 9
I CAN PROGRAM
 10
```

Notice that on the screen display the usual sequence of **I CAN PROGRAM** alternating with numbers **1** through **10** works just fine, except that the seventh time was omitted.

If you change line **60** slightly, the effect is quite different:

Edit line 60 to read 60 IF Z>7 THEN GOTO 90

List your program.

```
10 REM PRACTICE PROGRAM
50 FOR Z=1 TO 10
60 IF Z>7 THEN GOTO 90
70 PRINT "I CAN PROGRAM"
80 PRINT Z
90 NEXT Z
```

This time, when the computer comes to line **60**, it checks whether **Z** is greater than **7**. (The > sign means *greater than;* the other two main comparisons that can be used are < for *less than* and <> for *not equal to.* You can also combine so that <= means *less than or equal to* and >= means *greater than or equal to.*)

RUN the program.

```
I CAN PROGRAM
 1
I CAN PROGRAM
 2
I CAN PROGRAM
 3
I CAN PROGRAM
 4
I CAN PROGRAM
 5
I CAN PROGRAM
 6
I CAN PROGRAM
 7
```

Notice that although the **FOR . . . TO . . .** instruction goes from **1** to **10**, the display on the screen goes only from **1** to **7**. Once **7** gets to **8** or higher, the comparison after the **IF** is true (**8** is greater than **7**), so the computer accepts the command after the **THEN**, which is **GOTO 90**, and hence skips the **PRINT** lines **70** and **80**.

We realize that we have you playing around with a pretty useless, inane program. In this case, useless simplicity is the better part of valor. In the next chapter, you will write a program that "does" something. Actually, this program does a lot—it is teaching you to program.

Giving the Computer Information While It Is Carrying Out a Program—the INPUT Instruction

The **INPUT** instruction is used to *put* information *in* while the program is running.

Type 30 INPUT Y

Edit line 80 to read 80 PRINT Z,Z+Y

List your program.

```
10 REM PRACTICE PROGRAM
30 INPUT Y
50 FOR Z=1 TO 10
60 IF Z>7 THEN GOTO 90
70 PRINT "I CAN PROGRAM"
80 PRINT Z,Z+Y
90 NEXT Z
```

When you run the program, the computer will stop cold when it gets to line **30**. When the computer encounters an **INPUT** instruction, it stops and waits for the person sitting at the computer to type in information (usually a number). The computer lets you know it is waiting by putting a question mark (**?**) on the screen. What you are supposed to do is type a number and press the [RETURN] key. Then the computer makes the variable named in the **INPUT** line—in this case **Y**—equal to the number you typed.

Once it has the needed number, the computer proceeds to the next line in the program—but it remembers from then on that **Y** is equal to the number you typed.

Let's follow our expanded program step by step. At line **10**, you have a **REM** instruction. The computer passes it by. At line **30**, the computer hits your **INPUT Y** (you could, of course, have used any letter or variable name here). The program stops running and a **?** is displayed on the screen. You type a number—let's say **20**. When you press [RETURN], the computer sets **Y** equal to **20**.

Continuing on to the next numbered line, which is **50 FOR Z=1 TO 10**, the computer sets **Z** equal to **1**. At line **60**, the **IF . . . THEN . . .** line, it tests whether **Z** is greater than **7**. Since **Z** is not greater than **7**, the computer proceeds to line **70** and prints out **I CAN PROGRAM**.

The next line, your new line **80**, tells the computer to print two things: first, the value of **Z**—which at this point is **1**—and then

the value of **Z+Y**. Since **Z** equals **1** and **Y** equals **20**, **Z+Y** is **21**. Thus, the computer prints a **1** and a **21** on the screen line just below **I CAN PROGRAM**.

The computer goes on to **90 NEXT Z**, which sends it back to line **50**, where it sets **Z** equal to **2** and continues.

The second time, everything happens the same way as the first, except that at line **80**, because **Z** is now equal to **2**, what gets displayed on the screen is **2** and **22**. (**Y** has not changed. It is still **20**, so **2+20=22**.)

```
10 REM PRACTICE PROGRAM
30 INPUT Y
50 FOR Z=1 TO 10
60 IF Z>7 THEN GOTO 90
70 PRINT "I CAN PROGRAM"
80 PRINT Z,Z+Y
90 NEXT Z
```

Run the program. When you see the ? on the screen, type 20, press [RETURN], and watch what happens.

```
I CAN PROGRAM
 1             21
I CAN PROGRAM
 2             22
I CAN PROGRAM
 3             23
I CAN PROGRAM
 4             24
I CAN PROGRAM
 5             25
I CAN PROGRAM
 6             26
I CAN PROGRAM
 7             27
```

Run the program again. This time, when the ? appears, type in a different number. Try this several times.

Stopping Your Program—[RUN/STOP]; [RUN/STOP] and [RESTORE]

A small but important digression: "How do I stop this thing?"

Usually, as you have already learned, if you want to stop a program while it is running, you press the [RUN/STOP] key. This stops the program, but it does not clear the screen or erase your program. It just stops it wherever it is, and the computer tells you **BREAK IN 70**, or **BREAK IN** whatever line it has stopped at.

If, however, the computer is at a point in the program where it is waiting for you to type some input, pressing the [RUN/STOP] key doesn't work. The computer will think the [RUN/STOP] key is somehow some information you want to give it!

Run your program.

When the computer is waiting for input (the ? is on the screen), press the [RUN/STOP] key.

Nothing happens.

In this case, you must hold the [RUN/STOP] key and firmly press the [RESTORE] key. This stops the program almost any time. It also clears the screen. But unlike what happens when you turn off your computer, the program is still in memory.

Hold [RUN/STOP] and press [RESTORE].

This stops the program and clears the screen. The program is still in memory.

List your program.

See? As a general rule, any time [RUN/STOP] doesn't stop a program, try [RUN/STOP] and [RESTORE]. You could always use this procedure, except that you may not always want to clear the screen.

Prompts—Telling the Person at the Keyboard What to INPUT

It is sometimes hard to remember what you are supposed to do when you see the **?** from an **INPUT** line. It is therefore a good idea to add a **PRINT** line, just before the **INPUT** line, that tells you what to do. This is called a *prompt* because it prompts the person at the keyboard (you). (A prompt is sometimes called a *screen message.*)

Type 20 PRINT "TYPE A NUMBER & PRESS RETURN"

List your program.

```
10 REM PRACTICE PROGRAM
20 PRINT "TYPE A NUMBER & PRESS RETURN"
30 INPUT Y
50 FOR Z=1 TO 10
60 IF Z=7 THEN GOTO 90
70 PRINT "I CAN PROGRAM"
80 PRINT Z,Z+Y
90 NEXT Z
```

Here is what will happen this time. Just before the computer gets to your **INPUT** line, it comes to line **20**, which tells the computer to print your prompt. Thus, when the computer immediately goes on to line **30** and stops to wait for you to input a number, you know what to do—because displayed on the screen (in addition to the **?**) is the phrase in line **20**.

Run the program.

This should have worked pretty well.

There is one other fine point. It could be confusing to leave your prompt up on the screen after you input the information it called for—particularly if there was another **INPUT** line somewhere later in the program. The solution is to put a clear-screen *after* the **INPUT** line.

Type **40 PRINT** "♡" (Remember, when you are typing within quotes, to get the reverse-heart figure—meaning that when the computer is running the program, it will clear the screen when it gets to it—hold [SHIFT] and press the [CLR/HOME] key.)

List the program.

```
10 REM PRACTICE PROGRAM
20 PRINT "TYPE A NUMBER & PRESS RETURN"
30 INPUT Y
40 PRINT "♡"
50 FOR Z=1 TO 10
60 IF Z=7 THEN GOTO 90
70 PRINT "I CAN PROGRAM"
80 PRINT Z,Z+Y
90 NEXT Z
```

Run the program.

In general, then, an **INPUT** instruction involves a total of three program lines.[2] Most important is the **INPUT** line itself. Just before it is a **PRINT** instruction to give the person using the computer a prompt as to what kind of information to type in. Just after the **INPUT** line is a clear-screen instruction so that your prompt is not left up on the screen to confuse you.

Storing Your Programs

After you have written a program and seen how it works, you may want to save it for future use. You could, of course, write it down on paper and type it again when you want to use it. But if the program is long, that's rather time-consuming—especially since the computer is so fussy about typographical errors. You *could* just leave the program in your computer, but the Commodore 64 can only remember one program at a time. Also, if you turn the computer off, the program is lost.

[2]The Commodore 64 has a special procedure that allows you to combine a prompt with an **INPUT** instruction on the same program line. You type the word **INPUT**, then put your prompt (within quotes, but without the word **PRINT**), then a semicolon, then the variable name. For our example here, lines **20** and **30** would become just one line—**30 INPUT "TYPE A NUMBER & PRESS RETURN"; Y**. We recommend, however, using the slightly longer procedure of making the prompt a separate line with a **PRINT** instruction because if your prompts are very long, the Commodore 64 often rejects the combined form as incorrect. Also, it is easier to see what you are doing when you use a separate prompt instruction.

The solution is simple if you own the special cassette player the Commodore people sell for just this purpose—the "Datassette." Actually, a Datassette is an ordinary cassette tape recorder with a little bit of special wiring. (You may recall that we discussed the Datassette in Chapter 2.) If you plan to do much programming, or if you want to have available to you the many prepackaged programs that come only on cassettes, then the Datassette is probably a very good (and not very large) investment. In the section below, we will tell you how to use it to *save* a program.

If you will be keeping a lot of programs, or very long ones, or if you will be doing programs that involve handling a lot of information, you may very well want to buy an even more efficient recording device, a computer *disk drive.*

Box 6-1
"SAVEing" Your Programs on Diskettes

First, see Box 2-1 for how to set up a disk drive, connect it to your computer, handle and put in diskettes, and use it to load programs from the diskette into the computer.

Saving a program is quite simple, but you must first have a *formatted* diskette to save it on. If you buy any blank 5-1/4 inch diskette at the store and slip it into the disk drive, it won't work when you try to save a program on it. It has to be set up (formatted) to receive information from the Commodore 64.

Formatting a diskette requires these three simple steps:

1. Turn on the disk drive and put in a diskette, following the procedures described in Box 2-1. (Be careful to use either a blank diskette or one you want to erase, because this procedure wipes out everything that was previously on the diskette.)
2. Type **OPEN 15,8,15** and press [RETURN].
3. Type **PRINT#15 "NEW0: NAME,01"** and press [RETURN]. You can put in any letters you want instead of *NAME*, and you don't have to use **01** for your numbers. But you must put in some name and some two numbers to identify the diskette, or the procedure doesn't work.

Recording Programs on Your Datassette—SAVE, VERIFY, and LOAD

If you don't own a Datassette, skip this section. If you own a disk drive, see Box 6-1.

If you do own a Datassette, here is how you use it to store programs.

First, set up the Datassette as described in Chapter 2. In case you've forgotten: Plug the rectangular plug at the end of the gray cord into the only slot it will fit into on the back of the computer. (It won't fit at all unless you have the side with the cross-head screw on top—check both sides to see what we mean.)

Now let's record your practice program on tape as an example.

Put a blank tape into the Datassette, rewind it completely, and press the little button by the counter to set it to 0 (it may not go quite to 0, but it should get at least to 1).

The next step is to type **SAVE** and then, within quotes, the name you want to use for this program (using 16 or fewer spaces).

Type SAVE "PRACTICE PROGRAM" and press [RETURN].

The screen displays **PRESS RECORD & PLAY ON TAPE** (that is, on the tape recorder!)

Follow the screen directions. Simultaneously press both the RECORD and PLAY buttons on the Datassette.

The Datassette starts moving and the screen goes completely blank. This respite lasts about 15 to 20 seconds. It would be longer if you were saving a longer program.

When the program is recorded, the Datassette stops, the normal TV display reappears, and just below where it said **PRESS RECORD & PLAY ON TAPE**, it skips a line and says:

```
SAVING PRACTICE PROGRAM
READY
■
```

Your program is now on the tape. (In fact, it was automatically recorded twice.) It is also still in the computer's memory.

It is a good idea at this point to press the STOP button on your Datassette so that the Datassette is not left on.

It is possible to have a bad tape or to have some other problem. You can check whether your program has been accurately recorded with the **VERIFY** command.

Rewind the tape.

Type VERIFY "PRACTICE PROGRAM" and press [RETURN].

The computer tells you **PRESS PLAY ON TAPE**.

Press PLAY on the Datassette.

Again, the Datassette starts up and the TV screen goes blank. About 15 seconds later, the screen should say:

```
VERIFY "PRACTICE PROGRAM"

PRESS PLAY ON TAPE
OK

SEARCHING FOR PRACTICE PROGRAM
FOUND PRACTICE PROGRAM
```

If it had not found your program, either it would have told you it had found some other program or it would have kept running through the tape, searching for your program. You can press the [RUN/STOP] key to keep it from searching on and on through the whole tape. If it didn't find your program, it is probably because you made a typographical error in the name in either the **SAVE** or **VERIFY** command (for example, leaving out the blank). Whatever the cause, if it hasn't found your program, press the [RUN/STOP] key, rewind the tape, and try again.

If it does find your program, press the Commodore key to get it to verify that the program on the tape is the same as what is in the computer's memory.

Press the Commodore key [C=].

When you complete these steps, after a few seconds the screen says **READY.** You now have a formatted diskette, on which you can save programs.

Saving a program on your formatted diskette requires just one step:

Type SAVE "PRACTICE PROGRAM", 8 and press [RETURN].

You can use any program name you want, up to 16 letters, but you must have the **8** at the end so that the computer knows you want to save it on a diskette. If you forget the **8,** the computer assumes you want to save the program on tape.

After you have done this, the screen says **SAVING PRACTICE PROGRAM** (if that's the name you've used). Then after a few seconds—or longer if you are saving a very long program—it says **READY.**

Verifying that the program is saved correctly also involves just one step:

Type VERIFY "PRACTICE PROGRAM",8 and press [RETURN].

The screen says **SEARCHING FOR PRACTICE PROGRAM** and then **VERIFYING.** (If it doesn't say this, it means the computer can't find your program. That probably means you simply mistyped the program name. For example, in this case you might have accidentally left out the space between **PRACTICE** and

Again, the Datassette starts up and the screen clears. This time it stops after about five seconds. If all went well, the screen should say (after **FOUND PRACTICE PROGRAM**):

```
VERIFYING
OK

READY.
■
```

Now, you can feel confident that your program has been saved on tape. You can turn off your computer or start typing a new program and know that you have a record of your old program available any time.

Loading a program back from tape is done the same way you load a prepackaged program, as we discussed in Chapter 2. The procedure is as follows: Type **LOAD** and the program name (within quotes) and press [RETURN]. The computer tells you **PRESS PLAY ON TAPE**. When you do so, the screen goes blank. When it comes on again (in this case, in about 15 seconds), it tells you it has found your program. To get the program into the computer's memory right away, press the Commodore key. The screen goes blank again (for about 5 seconds). When it comes on this time, it tells you it is ready. The program is now in the machine. You can run or list it at will. Note that any other program you had in memory at the time has been replaced by the one you just loaded. When you are done with one program and want to load a new one, be sure you have saved the old one first (unless it is one you have already saved).

Now, you know the key tools of programming. It's been a long chapter, but you learned it all—all the essentials.

What's up next?

In the next chapter, you will—finally—use these tools to make a *useful* program. No more "**I CAN PROGRAM**" because now you really can!

PROGRAM. The computer is very exacting in wanting the name just so.)

If the computer finds your program, it says **OK**. **OK** means the program you saved (which is still in the computer's memory) matches precisely the version that is now on the diskette. (If it does not say **OK**, you probably accidentally changed something in the program before verifying. Occasionally, however, there is an imperfection in the diskette material.)

You can save more than one program on a diskette. To keep track of what you have saved, you can make the computer give you a list of what is on the diskette.

To list the program names on a diskette:[1]

1. Type **LOAD "$",8** and press [RETURN]. After a few seconds the screen says **READY.**
2. Type **LIST** and press [RETURN].

[1]Do not use this procedure if you have a program in the computer's memory that you don't want to lose but that you haven't saved yet. Save that program first because this procedure wipes out any program currently in memory.

Summary

This chapter introduces the rest of the essential BASIC terms: **REM**, **FOR...TO...,STEP**, **NEXT**, **IF ...THEN...**, and **INPUT**. You also learn the procedures for saving, verifying, and loading a program.

REM stands for "remark." When the computer encounters a numbered line beginning with **REM** while it is carrying out a program, it ignores the **REM** line and proceeds to the next line in numerical order. **REM** is used for titles or comments that you want to appear in the program listing.

A **FOR . . . TO . . .** program line (such as **5Ø FOR X=1 TO 1Ø**) sets the first value for the variable named the first time the computer encounters that line. (In this example, it would set **X**

equal to **1**.) The computer then carries out all subsequent lines as usual until it comes to a **NEXT** instruction (for example, **90 NEXT X**). At this point, it automatically jumps back up to the **FOR . . . TO . . .** line and sets the variable one number higher. (In the example, **X** would now be set equal to **2**.)

The computer proceeds again in the usual fashion until it comes again to the line with the **NEXT** instruction. Again, it goes back to the **FOR . . . TO . . .** line and sets the variable one number higher still. It keeps going through this loop, carrying out all instructions until it has set the variable equal to the number after the **TO**. (In the example, that would be **10**, since the line says **90 FOR X=1 TO 10.**) Once the variable has been set at its highest value, when the computer comes to the line with the **NEXT** instruction, it does not go back. It simply proceeds to the line below.

The **FOR . . . TO . . .** and **NEXT** procedure is very flexible. You can use any variable name (for example, **50 FOR Z=1 TO 10**). You can go from any number to any number (for example, **50 FOR Z=3 TO 7**). You can make the numbers go up by steps other than **1** by specifying a **STEP** value (for example, **50 FOR Z=1 TO 9 STEP 2**). You can even make the steps go down by giving a negative **STEP** value (for example, **50 FOR Z=10 TO 1 STEP −1**).

An **IF . . . THEN . . .** instruction (for example, **60 IF Z=7 THEN GOTO 90**) requires the computer to test whether the equation (or other comparison, such as < or >) after the **IF** is true. If it is true (in the example, when **Z** *does* equal **7**), then the computer does whatever comes after the **THEN**. (In the example, it would **GOTO 90**.) But if the equation is *not* true, the computer proceeds to the next numbered line, ignoring whatever comes after the **THEN**.

INPUT permits you to type information into a program while it is running. When the computer encounters an **INPUT** instruction (for example, **30 INPUT Y**), it stops. It displays a **?** on the screen. It waits for you to type in a number and press [RETURN]. Once you have done so, it sets the variable named in the **INPUT** line equal to the number you typed. (For example, if you typed **20**, it would set **Y** equal to **20**.) The computer then proceeds to the rest of the program.

If you want to stop a program just when the computer is waiting for you to type something for input, pressing [RUN/STOP] doesn't work. Holding down [RUN/STOP] and pressing the [RESTORE] key, however, stops a program almost any time. It also clears the screen. But it does not erase your program from memory.

To help you recall what kind of input to provide, it is a good idea to have a program line that prints some instructions just before an **INPUT** line. (For example, **20 PRINT "TYPE A NUMBER & PRESS RETURN"**.) This is called a *prompt*. It stays on the screen after the computer has gone on to the **INPUT** line and waits for you to type your input.

After the **INPUT** line, before anything else is displayed on the screen later in the program, you may want to have a line that eliminates the prompt by clearing the screen. You do this by putting a line containing the reverse-heart figure right after the **INPUT** line.

If you own a Datassette tape recorder, you can record your programs for future use. First, of course, you must plug your Datassette into the computer, put in a blank tape, and rewind it to the beginning. Then type **SAVE**, followed by a program name, within quotes, of up to 16 spaces. Press [RETURN]. The computer tells you what to do from then on.

You can check that the version recorded on tape matches the program still in the computer's memory by rewinding the tape and typing **VERIFY**, followed by the program name (within quotes). The computer tells you what do do next. Once it has found your program, you press the Commodore key. In a few seconds, if the two versions match, the computer reports **OK**.

To get a program into the computer from tape, type **LOAD**, followed by the program name within quotes. The computer tells you what to do next. When it reports it has found your program, press the Commodore key to actually put the taped program into the computer's memory. The program from the tape replaces any program previously in memory.

Terms and Concepts Introduced in Chapter 6

REM

FOR . . . TO . . . , STEP, and **NEXT** loops

Variable

IF . . . THEN . . .

= > < <> <= >=

INPUT

[RUN/STOP] and [RESTORE]

Prompt

Three-line prompt-and-**INPUT** sequence

SAVE

Saving programs on the Commodore Datassette

VERIFY

Practice Exercises

(Answers and comments in Appendix A.)

1. Write a program to produce the following on the screen. (Use the **FOR . . . TO . . . , STEP**, and **NEXT** procedures and other procedures as needed.)

```
SHIP NUMBER
 1
SHIP NUMBER
 2
```

2. Modify it to produce the following:

```
SHIP NUMBER
 100
SHIP NUMBER
 101
SHIP NUMBER
 102
```

3. Modify it further to produce this:

```
SHIP NUMBER
 100
SHIP NUMBER
 200
SHIP NUMBER
 300
SHIP NUMBER
 400
```

4. Modify it even more to produce this:

```
SHIP NUMBER
 100
SHIP NUMBER
 200
SHIP NUMBER
 400
```

5. Modify it so that you can put in a starting number, and the screen will produce the usual display, but the numbers will increase by 1 each time, starting at the number you put in. For example, if you put in **789,** such a program would produce:

```
SHIP NUMBER
 789
SHIP NUMBER
 790
SHIP NUMBER
 791
```

6. Have the program display on the screen, just before you tell it the number, a message that tells you to type a starting ship number.
7. Run the program, and stop it while it is waiting for you to type the starting ship number.

PART I CHAPTER 7

Some Practical Applications for What You've Learned

Finally, some practical applications! In the last few chapters, you learned the fundamental tools of writing computer programs in BASIC. Now, the payoff. In this chapter, you will begin to apply those tools to writing practical programs.

First, you will work out a short program to convert meters to feet. This is an example of a whole category of useful computer programs that do conversions. Then you will work out a program to balance your checkbook. This is a little more involved than the conversion example and gives you a good understanding of the logic of how programs go together. It also introduces you to the main features of programs that do various kinds of bookkeeping and financial management.

First, you need to know how to clear the computer's memory of old program lines so that you can start out fresh.

NEW—Starting with a Clean Slate

Type NEW and press [RETURN].

Nothing happens on the screen except that **READY** appears below your **NEW** command and the cursor (flashing box) moves down to a new line.

But something very important has happened inside the computer's memory: Any numbered program lines that were in memory have been wiped out. So use **NEW** with great caution. A whole day's programming could be wiped out in the blink of a Commodre's eye. On the other hand, when you are starting to write a new program, it is important to clear the memory of any old program lines. Otherwise, the old lines would get mixed in with the new ones.

A Conversion Program

Type in the following five-line program. (Remember, you get the reverse-heart figure for the clear-screen command in line **30** by holding down [SHIFT] and pressing the [CLR/HOME] key. Also, don't forget to press the [RETURN] key as soon as you have typed each program line.)

```
10 PRINT "HOW MANY FEET"
20 INPUT F
30 PRINT "♥"
40 M=F*.3048
50 PRINT F "FEET =" M "METERS"
```

List your program.

When you run this program—as you will in a minute—the computer first looks to see what numbered lines it has in its memory. Since you typed **NEW** before starting, the computer should find just these five lines. It searches first for the lowest numbered line. That's **10 PRINT "HOW MANY FEET"**. Accordingly, the computer displays that phrase on the screen, then goes on to the next lowest numbered line. That's **20 INPUT F**. The computer puts a question mark on the screen and stops. It sits there, waiting for you to type a number and press the [RETURN] key. Once you do, the computer has in its memory that **F** is now equal to the number you typed. It then goes on to line **30**, which tells the computer to clear the screen.

Thus far, you have a very standard three-line prompt-and-**INPUT** sequence, as discussed in the last chapter. First there is a **PRINT** instruction that displays a prompt so you know what to type. Then there is the actual **INPUT** line, which tells the computer to stop and wait for a number to be typed and to set the letter specified in the **INPUT** line equal to the number typed. Finally, there is a line to clear the screen so that the prompt and the number are no longer on the screen to confuse anyone.

Now, line **40** tells the computer to set **M**—this is a new variable—equal to the *result* of multiplying **F** by **.3048**. This is a very important line because it actually makes the conversion. If you look it up, you'll see that multiplying any number of feet by .3048 gives you the number of meters. For example, 1000 feet equals 304.8 meters. Obviously, the computer does not already know the various numbers needed for conversions, so you must supply them—as you have done here in line **40**.

Line **40** is actually a new type of BASIC instruction, sometimes known as a **LET** (or "assignment") instruction because you could also write the line as **40 LET M=.3048*F**. But it would be longer that way and is clear enough without the **LET**. This kind of program line tells the computer to make the variable before the equals sign equal to the number—or the number that is the result of the arithmetic—on the other side of the equals sign.

For example, **100 X=4** tells the computer to set **X** equal to **4**. A later numbered line in such a program might read **200 Y=X+6**. At this line, the computer would remember that **X** is equal to **4**, so it would set **Y** equal to **10**. The computer would now remember not only that **X** is **4**, but also that **Y** is **10**.

The equals sign in these examples, however, does not mean quite the same thing as an equals sign in arithmetic. For example, you could also have a program line that reads **300 X=X+1**. In arithmetic, the statements at the right and left sides of the equals sign must always be actually equal. But **X** and **X+1** obviously cannot be equal.

What the line **300 X=X+1** means to the computer is quite straightforward. It means the computer should make **X** equal to whatever was the previous value of **X**, plus **1**. Thus, if **X** had been equal to **4**, after this program line it would be remembered as equal to **5**. It is really not at all complicated, but we mention this because it sometimes confuses people when they see what looks like an arithmetic mistake!

So, by the time the computer gets to line **50**, it has already calculated how many meters are equivalent to the number of feet you input. It remembers that number as **M**. Although the computer knows the value of **M**, *you* are still in the dark. You need a program line to have the computer share its knowledge. That's what line **50** does, and it does it with flair. Line **50** doesn't just print **M**—all you really need to know. It plans for **M** to be easy for you to *understand!* After the word **PRINT**, the computer is told to display the value of **F**—the number of feet you put in. Then it displays **FEET =**. Then it gives the value of **M**. Finally, to finish off the line, it prints the word **METERS**.

Run the program. When it calls for HOW MANY FEET, put in 1000—and don't forget to press [RETURN].

The screen should show this:

```
1000 FEET = 304.8 METERS
```

Run the program three or four more times, each time giving a different number of feet.[1]

Besides offering the first practical use of your Commodore 64, this little program demonstrates an important principle: A computer program usually involves three main activities. First, it takes in the information it needs. Second, it does something useful with that information. Third, it displays on the screen (or in some other way produces) what it has done.

[1] If you use a number that is more than eight numerals long, the computer prints it out using *scientific notation*. For example, one billion—1000000000—would come out **1E+09**. That means 1 times 10 to the ninth power. Or, more simply, 1 and then move the decimal point over nine places. Similarly, 12345000000 would come out **1.2345E+10**. This business comes up only very rarely. You don't really have to understand it—just know when you see it that the computer hasn't gone crazy!

In this example, the prompt-and-**INPUT** sequence accomplished the first part, line **40** "did something," and line **50** put out the result. In other programs you will write or see in books, each of these three aspects may be much more complex and involve many subparts. But in almost all cases you can divide a program into these three activities.

A Variable Table—Understanding What the Computer Is Doing Inside

One way to get a deeper understanding of what the computer does when it carries out your program is to make a chart of what the letters (variables) are equal to at each point along the way. This is called constructing a *variable table.* You will not ordinarily bother with this procedure when writing your own programs. But at this stage of your learning, we think you will find it helpful, if a little dull. Follow along as we go through a variable table for this simple conversion program. In the future, if you run into problems in more complicated programs, you may want to lay out such a table.

Variable Table for Conversion Program

Line	F	M
10	—	—
20	1000	—
30	1000	—
40	1000	304.8
50	1000	304.8

When the computer carries out a program, internally it is busy remembering what the different variables are equal to. In your conversion program, at line **10** the computer has not yet encountered any variables to remember. At line **20**, you input some number the computer remembers as **F**. In this case, we are still assuming you typed in **1000** when you ran the program. There is still no number for **M**. In fact, to the computer, there is no **M** yet. Line **30**, which simply clears the screen, does not change what any of the letters are equal to. However, line **40** is very important. It introduces a new variable and tells the computer how to calculate its value. In this case, the value is 304.8. Line **50** prints out what you told it to, but it does not change any of the variable values.

You can see that in a simple program like this, the computer does not have to do very much internally! We will see some programs later, however, in which the computer is kept very busy remembering and changing the values of many different variables.

Simplifying and Shortening a Program

There is one other thing we can learn from this short little conversion program. It is possible to make it shorter! For one thing, you could combine lines **40** and **50** to be **50 PRINT F "FEET =" F*.3048 "METERS"**. This is possible because, as you will recall from Chapter 4, a **PRINT** instruction displays the result of an arithmetic calculation (as long as it is not written within quotes). Notice, incidentally, that by combining the lines in this way, you no longer need to use the variable **M**.

In fact, you can also combine line **30** with the above by including the clear-screen (quote, reverse-heart figure, unquote) instruction in line **50**. Try it:

Type 50 PRINT "♡" F "FEET =" F*.3048 "METERS"

Delete lines 30 and 40.[2]

List your program.

This new three-line program does everything the old five-line version did. Try it:

```
10 PRINT "HOW MANY FEET"
20 INPUT F
50 PRINT "♡" F "FEET =" F*.3048"METERS"
```

Run the program using 1000 when it calls for a number.

Shortening a program in this way is not necessary. It works fine long or short. But we want at least to show you the shorter version, so you will understand it when you come across it in magazines or books.

Writing the Program to Repeat Itself Automatically

You probably found it slow when you made several conversions from feet to meters to have to type **RUN** and press [RETURN] each time. You can make the program automatically repeat itself by adding a line at the end that tells the computer to go to the beginning.

Type 60 GOTO 10

List your new version of the program.

[2]You may recall from Chapter 5 the two ways to delete a numbered line: 1. Move the cursor to the line you want to delete, delete everything in the line but the line number, and then press [RETURN]; or 2. Just type that line number of the screen and press [RETURN].

```
10 PRINT "HOW MANY FEET"
20 INPUT F
50 PRINT "□" F "FEET =" F*.3048"METERS"
60 GOTO 10
```

This time when you run the program, it does everything it did before. But when it finishes with line **50**, it comes to line **60** and is told **GOTO 10**. Thus, you again see displayed on the screen **HOW MANY FEET**. If you respond, the computer proceeds through the program again. As long as you provide input, the computer makes your conversion and asks for more input until you either turn off the computer or press [RUN/STOP] and [RESTORE]. Let's try it.

Run the program.

When the computer asks you HOW MANY FEET, give it 1000 the first time, then the second time try 50, then 2.76, then 412, and then hold [RUN/STOP] and press [RESTORE].

A Checkbook-Balancing Program

Now let's go on to a new program. The last one was so short, you probably won't want to save it. Just clean it out from the computer's memory.

Type NEW and press [RETURN].

Now, you are ready to begin.

Type the following lines:

```
10 REM CHECKBOOK BAL
20 PRINT "TYPE STARTING BALANCE"
30 INPUT B
40 PRINT "□IF NEXT ITEM IS DEPOSIT TYPE
1; IF CHECK TYPE 2; IF NO MORE TYPE 3"
50 INPUT X
60 IF X=3 THEN GOTO 120
70 PRINT "□TYPE DOLLAR AMOUNT OF ITEM"
80 INPUT AM
90 IF X=1 THEN B=B+AM
100 IF X=2 THEN B=B-AM
110 GOTO 40
120 PRINT "□NEW BALANCE IS $" B
```

List your program and check your typing.

Before you try running this program, let's go through it line by line so that you understand how it works.

Line **10** is a **REM** instruction, which the computer ignores when it runs the program. **CHECKBOOK BAL** is abbreviated so that you can use it as the program name when you save it on tape.

(Remember—when saving, the program's name has to be 16 spaces or shorter.) Line **20** prints a prompt—**TYPE STARTING BALANCE.** This prompt is on the screen when the computer comes to line **30**, **INPUT B**. When you obey the prompt and type your starting checkbook balance (and of course, press [RETURN]), the computer sets **B** equal to that starting balance and goes on to the next line in order.

At this point, the prompt is still on the screen and the computer has in its memory that **B** is equal to some particular number (your starting balance).

Lines **40** and **50** are another prompt-and-**INPUT** sequence that first clears the screen and then asks you to tell the computer whether the next item is a deposit or a check, or whether there are no more items. Here, you type a **1**, **2**, or **3** and press [RETURN].

You come to this prompt several times as the program is carried out. Each time, including, alas, the first time, the prompt asks about the **NEXT ITEM**. It would require a lot of extra programming to make it say "first item" the first time through. Fortunately, it is pretty clear as it is.

Notice two things here:

1. Line **40** begins with the reverse-heart figure so that the screen will be cleared of the prompt from line **20**.
2. It might seem easier to ask you to simply type **DEPOSIT** or **CHECK** or **NO MORE ENTRIES**. Although there is a way you could write the program so this would work, it involves procedures you have not learned yet. It would also mean you would have to type more than just a number each time. Even professional programmers prompt the person at the keyboard to type a single number to indicate a selection from a list of things.

Back to the program. When the computer has finished with line **50**, the prompt from line **40** is still on the screen. The computer has in its memory a number for **B** (the number you put in for your starting balance) and also for **X** (either a **1**, **2**, or **3**, according to whether the first item is a deposit or a check, or there are no more items).

At line **60**, the computer has to make a decision: Is it true that **X=3**? If it is (that is, if you had typed **3** because there were no more items), then the computer would go to line **120** and print the new balance.

The first time through, there should be some items. More likely, **X** is equal to **1** or **2.** In that case, at line **60** the computer finds that the **IF** equation is false and won't carry out the **THEN** instruction. Instead it proceeds directly to the next line.

Lines **70** and **80** are yet another prompt-and-**INPUT** sequence. This time, the idea is to find out the actual dollar amount of the check or deposit under consideration. When you run the

```
10 REM CHECKBOOK BAL
20 PRINT "TYPE STARTING BALANCE"
30 INPUT B
40 PRINT "□IF NEXT ITEM IS DEPOSIT TYPE
1; IF CHECK TYPE 2; IF NO MORE TYPE 3"
50 INPUT X
60 IF X=3 THEN GOTO 120
70 PRINT "□TYPE DOLLAR AMOUNT OF ITEM"
80 INPUT AM
90 IF X=1 THEN B=B+AM
100 IF X=2 THEN B=B-AM
110 GOTO 40
120 PRINT "□NEW BALANCE IS $" B
```

program, type the amount without a dollar sign, and press [RETURN].

All right. The computer has completed everything through line **80**. It has found out your initial balance and set it equal to **B**. It has found out whether your first item was a check or deposit and set **X** equal to either **1** or **2**, accordingly. It now also knows the dollar amount of that check or deposit and has set **AM**—short for amount—equal to that. For the moment, the prompt at line **70** is still on the screen.

Lines **90** and **100** are two **IF . . . THEN . . .** lines. The computer uses these lines to look at what **X** is equal to, and then it acts accordingly. Line **90** checks whether **X** is equal to **1**. Of course, **X** would be equal to **1** only if you had typed **1** in response to the prompt at line **40**. That is, **X** would equal **1** only if this item was a deposit. So when line **90** checks if **X** equals **1**—and it does—it *adds* the amount (**AM**) of that item to the value of **B**. In other words, if the item under consideration is a deposit, then it gets added to your balance.

If, on the other hand, the item was a check, **X** would be equal to **2** and the **IF** in line **90** would *not* be true. In that case, the computer would ignore the **THEN** part of line **90** and go right on to the next line.

Line **100** considers whether **X** is equal to **2**. If it is—that is, if the item is a check—then line **100** subtracts the item amount (**AM**) from the balance (**B**).

At this point, the computer has progressed through line **100**. It has found out your initial balance (**B**), whether the first checkbook item to consider is a check or deposit (**X**), and how much the item is (**AM**). Then it either added it to your balance if it was a deposit or subtracted it from your balance if it was a check.

Line **110** is very simple. It sends the computer back to line **40** so it can go through the whole process again for your next item.

The computer keeps up this cycle (and thus keeps adding deposits and subtracting checks from your balance) until at some point, when it comes to line **40**, you have run out of checks and deposits. Then you type a **3**, indicating you have no more items. When the computer comes to line **60**, the **IF** equation is true—**X** equals **3**. The computer will then go to line **120**.

At line **120**, the computer clears the screen, prints **NEW BALANCE IS $**, and then the value of **B**. **B**, of course, is the new balance—the starting balance plus all your deposits and minus all your checks. Isn't that lovely?

After carrying out line **120**, the computer finds no more lines in its memory, so it tells you it is ready for more instructions.

Let's try running this program. To check if it is working correctly, use the sample checkbook that follows. We already know the final balance for this. Once you understand the program, there will be plenty of time to balance your own checkbook with it.

Run the program.

Remember, when going through the checks and deposits, you are asked first to tell the computer whether the item is a check or deposit. Then, following the next prompt, you tell the computer the amount of the check or deposit. Also, if you make a mistake when putting in an amount, you can change it as long as you have not yet pressed [RETURN] for that entry. Once you have pressed [RETURN], the information is in. There is a way to write a program that allows you to fix mistakes—you'll learn how to do that next. For now, if you make a mistake, just hold [RUN/STOP] and press [RESTORE]. Then run the program again from scratch.

SAMPLE CHECKBOOK:

Starting Balance	127.23
Check	41.90
Deposit	1230.00
Check	54.00
Check	21.25
Deposit	45.00

If you entered all of the numbers correctly, and the program was typed without errors, when you are finished the computer should display this line:

```
NEW BALANCE IS $ 1285.00
```

Programming to Allow Corrections of Input Mistakes

When you were entering all those numbers for balance, checks, and deposits, did you make any mistakes? If you caught the error before pressing [RETURN], there was no problem. You could simply edit the number on the screen and then press [RETURN] when you had the number right. But, if you had already pressed [RETURN], it was too late. To correct the mistake, you had to start the program over from scratch. If you had been doing your actual checkbook, with say 40 entries, and made a mistake around number 35—yikes!

Thus, most programs that require input of a lot of information include a procedure that allows you to check what you put in as you go along.

Type in the following lines:

```
70 PRINT "□TYPE DOLLAR AMOUNT OF ITEM"
80 INPUT AM
82 IF X=1 THEN PRINT "□YOU ENTERED A DEP
OSIT OF $" AM
83 IF X=2 THEN PRINT "□YOU ENTERED A CHE
CK OF $" AM
85 PRINT "IF CORRECT TYPE 0 TO GO ON TO
NEXT ITEM. IF WRONG TYPE 9 & TRY AGAIN."
86 INPUT C
88 IF C=9 THEN GOTO 40
```

List your program and check your typing of lines 82 through 88.

```
10 REM CHECKBOOK BAL
20 PRINT "TYPE STARTING BALANCE"
30 INPUT B
40 PRINT "□IF NEXT ITEM IS DEPOSIT TYPE
1; IF CHECK TYPE 2; IF NO MORE TYPE 3"
50 INPUT X
60 IF X=3 THEN GOTO 120
70 PRINT "□TYPE DOLLAR AMOUNT OF ITEM"
80 INPUT AM
82 IF X=1 THEN PRINT "□YOU ENTERED A DEP
OSIT OF $" AM
83 IF X=2 THEN PRINT "□YOU ENTERED A CHE
CK OF $" AM
85 PRINT "IF CORRECT TYPE 0 TO GO ON TO
NEXT ITEM. IF WRONG TYPE 9 & TRY AGAIN."
86 INPUT C
88 IF C=9 THEN GOTO 40
90 IF X=1 THEN B=B+AM
100 IF X=2 THEN B=B-AM
110 GOTO 40
120 PRINT "□NEW BALANCE IS $" B
```

When the computer carries out this new, improved version of the checkbook balancing program, it proceeds exactly as before through line **80**. At that point, you will have provided the information for your first item. What these new lines do is give you a chance to correct that information if you have made a mistake.

Lines **82** and **83** look at whether you said the item was a check or deposit (that is, whether you typed a **1** or a **2** in response to the prompt about whether the next item is a deposit or a check). In either case, these lines clear the screen of the prompt from line **70** and display the information you typed. If it was a deposit, this is accomplished by the **PRINT** instruction after the **THEN** in line **82**. This calls for the computer to show you the amount you typed

and that you indicated it was a deposit. If it was a check you indicated, then the corresponding **PRINT** instruction for line **83** is activated. At this point, then, you have a chance to review what you have just typed.

Line **85** is a prompt. It tells you what to do if what you typed is correct, and what to do if it is not correct. Line **86** is the actual **INPUT** instruction for this prompt. It assigns whatever number you type at this point to **C.**

Line **88** looks at what you typed at line **86**. If you indicated you had made a mistake—that is, **IF C=9**—the computer then carries out the instruction to go to line **40**. This short-circuits the cycle and prevents the computer from getting to lines **90** and **100**, where it would adjust the balance with your information. Instead, it takes you back to line **40**, where you have a new chance to give the information for that item.

On the other hand, if you had given the correct information, you would have typed a **0** at line **86**. When the computer came to line **88,** it would find it was *not* true that **C=9.** Accordingly, it would ignore the **THEN** instruction and go right on to the next line in order. Thus, when you tell the computer that you typed the information for an item correctly, the computer goes right on and carries out the program in the usual way.

Run the program, using the sample checkbook information again. This time, purposefully make a few mistakes typing in checks and deposits and then correct them after you see them on the screen. Think about how the revised program is handling all this.

Save your program on tape to use in the practice exercises, as well as for your own personal use. You've finally done something useful with your investment!

In the next chapter, we will complete your introduction to the fundamentals by teaching you how to use the **PRINT** command to make tables and simple graphics.

Now do take a minute to lean back in your chair and relax. You are mastering the "cutting edge" of technology—the computer. You deserve a break.

Summary

This chapter illustrates the practical application of what you have learned in the previous chapters and introduces a few new ideas along the way.

First, before you begin a new program, you must clear old program lines out of memory by typing **NEW** and pressing [RETURN].

The first practical example is a program that converts feet to meters. A crucial step in this program involves a new kind of in-

struction, a line that sets a letter equal to some number, or equal to the result of an arithmetic calculation. For example, the line **40 M=F*.3048** tells the computer that **M** is equal to the number that results from multiplying **F** by **.3048**. Instructions that set a letter equal to a number can take several forms—for example, **X=5**, or **Y=5+X**, or even **Y=Y+1**. The last example would be understood by the computer to mean that **Y** is now equal to what it used to be equal to, plus **1**! Note that the use of the equals sign is not quite the same as in ordinary arithmetic.

The conversion example illustrates that most programs involve three aspects: taking in information, doing something with that information, and then giving out the result. In the conversion program example, some number of feet is put in by you, the computer converts that number to the appropriate number of meters, then the computer prints out the result.

To help understand how the computer actually carries out its internal manipulations, you can construct a *variable table*. This shows the value of each variable after the computer has carried out the instruction on each numbered line.

Programs can often be shortened by combining certain numbered lines. For example, in the conversion program the following three lines can be reduced to one:

```
30 PRINT "□"
40 M=F*.3048
50 PRINT F "FEET =" M "METERS"
```

Combines to:

```
50 PRINT "□" F "FEET =" F*.3048 "METERS"
```

Programs can be made to repeat themselves by adding a line at the very end that tells the computer to go to the first line again. In the conversion example, this line is **60 GOTO 10**.

We also consider a longer example of a practical program, one that balances your checkbook. First, prompt-and-**INPUT** lines find out your initial balance, whether the various items are checks or deposits, and how much each item is. Then a number of **IF . . . THEN . . .** lines look at what kind of information you put in. As a result, the computer either adds the item to the balance if it was a deposit, or subtracts it from the balance if it was a check. The next line then sends the computer back to obtain information on the next item. There is also an **IF . . . THEN . . .** line (line **60**) that allows you to tell the computer when you have typed all the items. If you have, then the computer prints out the new balance.

To be able to make a correction if you accidentally put in the wrong information, add program lines that first print out what you put in so you can check it. Then add a prompt-and-**INPUT** sequence to ask you if what you put in is correct. An **IF . . . THEN**

. . . line can then look at your reply. If you indicate the information is not correct, then have the computer go back up the program to permit you to input the information again. If you indicate the information is correct, the computer continues on to the next line in normal order.

Terms and Concepts Introduced in Chapter 7

NEW

Conversion program

LET (or assignment) instruction (for example, **LET M = F*.3048**)

Variable table

Three main parts of a program

Program section that allows user to correct **INPUT** mistakes

Practice Exercises

(Answers and comments in Appendix A.)

1. Balance your own checkbook using the checkbook balance program.
2. Redo the conversion program so that it changes yards to inches (1 yard = 36 inches).
3. Redo the conversion program so that it changes feet to yards (1 foot = .333 yard).
4. Redo the conversion program so that it changes feet to yards *and* to meters.

PART I CHAPTER 8

Designing Your Screen Display

In the last several chapters, you have learned the basics of writing your own computer programs. In the last chapter, we emphasized that programs usually perform three main activities: taking in information, doing something with it, and then "putting out" the result. Before concluding Part I of this book, we want to focus a little more on this last aspect—how to program output.

The Commodore 64 has tremendous graphics capabilities. In Part III (Chapters 12 to 15), you will explore these in some detail, if you choose to. But before going any further, you should know how to produce tables, graphs, and simple designs. These are important even if you are not interested in more complex computer graphics.

In this chapter, we will look at some programming procedures for determining how output gets displayed on the screen. These procedures mostly involve using the **PRINT** command with certain punctuation marks. Then, you will learn some ways to apply these procedures to programming tables, business graphs, and simple designs.

PRINT with No Punctuation Marks

Usually, when the computer encounters a **PRINT** instruction, it displays whatever that instruction calls for *on the first unused line going down the screen.* If it encounters another **PRINT** instruction later in the program, it puts the material in that instruction on the next line down. And so on.

Type NEW and press [RETURN], then clear the screen to give yourself a fresh start. (If you want to save the checkbook program from the last chapter, you'd better do so first!)

Type the following two-line program:

```
10 PRINT "LAYOUT #1"
20 PRINT "LAYOUT #2"
```

When the computer carries out this program, it first comes to line **10** and displays **LAYOUT #1** on the first unused line on the screen. When it comes to line **20**, it displays **LAYOUT #2** on the next screen line. Thus, the screen should show the second phrase directly below the first.

Run your program.

```
LAYOUT #1
LAYOUT #2
```

This principle of printing each new thing on the first available line down is seen even more dramatically in the program below.

Edit line 10 as shown:

```
10 PRINT "LAYOUT"
```

Retype line 20 as shown:

```
20 GOTO 10
```

Line **10** puts the word **LAYOUT** on the first unused screen line. Line **20** sends the computer back to line **10**, where it is again told to print the word **LAYOUT**. The second time, it puts **LAYOUT** on the next screen line down. Then it comes to line **20** again, goes back to line **10**, and again prints the word **LAYOUT**—one more screen line down. This goes on and on, endlessly. The result is a column with the word **LAYOUT** down the left side of the screen.

Run the program. After a few seconds, press the [RUN/STOP] key.

```
LAYOUT
LAYOUT
LAYOUT
LAYOUT
LAYOUT
LAYOUT
LAYOUT
```

```
LAYOUT
LAYOUT
LAYOUT
LAYOUT
LAYOUT
LAYOUT
LAYOUT
LAYOUT
LAYOUT
LAYOUT
LAYOUT
LAYOUT
LAYOUT
LAYOUT

BREAK IN 10
READY
■
```

Of course, you've seen this before with **I CAN PROGRAM** in Chapter 5, but now you will understand it better. The principle is that carrying out a **PRINT** instruction starts a new screen line, and the material in a subsequent **PRINT** instruction is displayed on the next screen line below that.

Because of this principle, you can use a program line that just says **PRINT** (with nothing after it) to leave a line empty. Since you have not said what to print, the computer simply leaves that line blank across the screen.

Type 15 PRINT

List your program.

```
10 PRINT "LAYOUT"
15 PRINT
20 GOTO 10
```

When the computer comes to line **15**, it prints what you have asked for—nothing. It leaves that line on the screen empty. At line **20**, it goes back to line **10** and again prints the word **LAYOUT.** This time, there is a blank, "skipped" line between the first and second printing of **LAYOUT**. The computer comes to line **15** again and again leaves a blank line. Then it goes to line **10** and prints **LAYOUT**, and so on.

Run the program. When the screen is full, press the [RUN/STOP] key.

```
LAYOUT

LAYOUT
```

```
LAYOUT

LAYOUT

LAYOUT

LAYOUT

LAYOUT

LAYOUT

LAYOUT

LAYOUT

LAYOUT

BREAK IN 10
READY
```

There should be a column of the word **LAYOUT** on the left side of the screen with every other line blank.

Maybe you remember from Chapter 5 what happens in an endless **PRINT** and **GOTO** loop. The computer keeps putting in new lines from the bottom, very fast. With the **I CAN PROGRAM** (or the simple two-line **PRINT "LAYOUT"** program you did a couple of minutes ago), it produced a flicker at the bottom. In this case, because of the alternation with the blank line, the whole display appears to be flashing.

PRINT with a Semicolon

When a **PRINT** line ends with a semicolon (;)—outside of quotes—the result of the next **PRINT** instruction *begins exactly where the first one* **left off**. You will see shortly why this might be useful. But first, a demonstration.

Type NEW, press [RETURN], and then type the following:

```
10 PRINT "LAYOUT";
20 PRINT "--#1"
```

List your program.

Line **10** prints **LAYOUT** on the first unused screen line. The computer takes note of the semicolon, which tells it to begin printing the next thing right where it left off printing the last thing,

on the same screen line. Thus, line **20** prints the characters —— **#1** starting at the first available space after **LAYOUT.**

Run the program.

The screen should show this:

```
LAYOUT--#1
```

Again, you can see the effect more dramatically with a **GOTO** loop.

Type 20 GOTO 10

List your program.

```
10 PRINT "LAYOUT";
20 PRINT GOTO 10
```

(Remember, when you type a new program line with the same number as an old program line, the new one replaces the old.)

This program first prints the word **LAYOUT**. Then, when it comes to line **20**, it goes back to line **10**, where it again prints the word **LAYOUT**. This time, it does so in the first free space on the same line. You would see the two of them, **LAYOUTLAYOUT**, if all this weren't happening so fast. When it comes around again to line **10**, it does it again, and continues to do so until you stop it.

Run the program and, once the screen is full, stop it with the [RUN/STOP] key.

If you want each printed word to stand out more clearly, insert a blank space after it.

Edit line 10 to read 10 PRINT "LAYOUT";

List your program.

Run your program and then, once the screen is full, stop it with the [RUN/STOP] key.

Now, each word is separated from the last by a blank space. But the display is still messy because your repetitions "wrap around" in the middle of a word from screen line to screen line.

There are exactly 40 spaces across the screen. You can make your display neater by printing a word of some length that evenly divides up a 40-space screen. Lengths that work out evenly are 1, 2, 4, 5, 8, 10, 20, and 40 spaces. In this case, if you add 1 more space, the total spaces will be 8. Try it.

Edit line 10 by adding another space after LAYOUT

List and run the new program.

```
LAYOUT  LAYOUT  LAYOUT  LAYOUT  LAYOUT
LAYOUT  LAYOUT  LAYOUT  LAYOUT  LAYOUT
LAYOUT  LAYOUT  LAYOUT  LAYOUT  LAYOUT
LAYOUT  LAYOUT  LAYOUT  LAYOUT  LAYOUT
LAYOUT  LAYOUT  LAYOUT  LAYOUT  LAYOUT
LAYOUT  LAYOUT  LAYOUT  LAYOUT  LAYOUT
LAYOUT  LAYOUT  LAYOUT  LAYOUT  LAYOUT
LAYOUT  LAYOUT  LAYOUT  LAYOUT  LAYOUT
LAYOUT  LAYOUT  LAYOUT  LAYOUT  LAYOUT
LAYOUT  LAYOUT  LAYOUT  LAYOUT  LAYOUT
LAYOUT  LAYOUT  LAYOUT  LAYOUT  LAYOUT
LAYOUT  LAYOUT  LAYOUT  LAYOUT  LAYOUT
LAYOUT  LAYOUT  LAYOUT  LAYOUT  LAYOUT
LAYOUT  LAYOUT  LAYOUT  LAYOUT  LAYOUT
LAYOUT  LAYOUT  LAYOUT  LAYOUT  LAYOUT
LAYOUT  LAYOUT  LAYOUT  LAYOUT  LAYOUT
LAYOUT  LAYOUT  LAYOUT  LAYOUT  LAYOUT
LAYOUT  LAYOUT  LAYOUT  LAYOUT  LAYOUT
LAYOUT  LAYOUT  LAYOUT  LAYOUT  LAYOUT
LAYOUT  LAYOUT  LAYOUT  LAYOUT  LAYOUT
LAYOUT  LAYOUT  LAYOUT  LAYOUT  LAYOUT

BREAK IN 10
READY
■
```

This should yield a full screen of five nice, even columns of the word **LAYOUT.**

You can also use semicolons within the same **PRINT** instruction.

Edit line 10 to read 10 PRINT "LAYOUT";"#——1"

List and run your program.

This has the same effect as using two separate **PRINT** instructions, with a semicolon after the first. The screen shows this:

```
LAYOUT#--1
```

(Obviously, it would have been easier to use one line and no semicolon, but real uses for this tool are coming up.)

In previous chapters, whenever we wanted to have one thing displayed right after the other, we put them in the same **PRINT** line, with no special punctuation between them. For example, you may recall this line from the last chapter:

```
50 PRINT "J" F "FEET =" F*.3048 "METERS"
```

You could have written this instead:

```
50 PRINT;"]";F;"FEET =";F*.3048;"METERS"
```

Both ways give the same result. The second is better because there are a number of cases in which, without semicolons, you would be in trouble (for example, when printing the value of two variables together, two numbers together, or a variable and a number together). The reason does not matter—just take our word for it for now. When you want two things in a **PRINT** instruction to come out one right after the other, it is a good programming habit to use semicolons.

PRINT with a Comma

Each screen line on the Commodore 64 is divided into four *print zones* of 10 spaces each. If a **PRINT** instruction ends in a comma, the next thing that gets printed is displayed on the same line, but in the next print zone.

Edit your current line 10 to read 10 PRINT "LAYOUT",

List your program.

The comma at the end of line **10** tells the computer to remember to begin printing the material in the next **PRINT** instruction at the beginning of the first unused print zone of the same screen line.

Run the program.

```
LAYOUT    LAYOUT    LAYOUT    LAYOUT
LAYOUT    LAYOUT    LAYOUT    LAYOUT
LAYOUT    LAYOUT    LAYOUT    LAYOUT
LAYOUT    LAYOUT    LAYOUT    LAYOUT
LAYOUT    LAYOUT    LAYOUT    LAYOUT
LAYOUT    LAYOUT    LAYOUT    LAYOUT
LAYOUT    LAYOUT    LAYOUT    LAYOUT
LAYOUT    LAYOUT    LAYOUT    LAYOUT
LAYOUT    LAYOUT    LAYOUT    LAYOUT
LAYOUT    LAYOUT    LAYOUT    LAYOUT
LAYOUT    LAYOUT    LAYOUT    LAYOUT
LAYOUT    LAYOUT    LAYOUT    LAYOUT
LAYOUT    LAYOUT    LAYOUT    LAYOUT
LAYOUT    LAYOUT    LAYOUT    LAYOUT
LAYOUT    LAYOUT    LAYOUT    LAYOUT
LAYOUT    LAYOUT    LAYOUT    LAYOUT
LAYOUT    LAYOUT    LAYOUT    LAYOUT
LAYOUT    LAYOUT    LAYOUT    LAYOUT
LAYOUT    LAYOUT    LAYOUT    LAYOUT
LAYOUT    LAYOUT    LAYOUT    LAYOUT
LAYOUT    LAYOUT    LAYOUT    LAYOUT

BREAK IN 10
READY
█
```

This time, there are four columns of the word **LAYOUT** across the screen. That's because each time the computer comes to line **10**, it begins printing **LAYOUT** in the next print zone—and there are four print zones across the screen.

As with semicolons, you can use commas to separate material within the same **PRINT** instruction. For example:

Edit line 10 to read 10 PRINT "ZONE 1","ZONE 2","THREE",-4

List and then run the program.

ZONE 1 gets displayed in the first print zone, then the comma tells the computer to start printing **ZONE 2** in the next print zone, and so forth. There are no quotes around **-4** because—remember—quotes are not needed to print a number or the result of a calculation. (You might want to refresh yourself on this by rereading some of Chapter 4.)

PRINT with SPC and TAB

Suppose you want to leave a lot of space between two things you want printed on the same screen line. You could type the first thing, type a lot of spaces, and type the second thing. Or, you could use commas to distribute your items across the four print zones, if that is the spacing you want.

There are still two more ways to accomplish the same thing—**SPC** and **TAB**.

Suppose the first word you want to print takes 4 spaces, and you want 18 spaces before the next word. The zones created by commas won't work. Let's try leaving spaces inside of quotes in the **PRINT** line.

Clear the computer's memory (with NEW), then type the following, leaving 18 spaces between HERE and 18:

```
10 PRINT"HERE                    18 SPACE
S LATER"
```

Now, let's try **SPC**.

Type the following line:

```
20 PRINT"HERE";SPC(18);"18 SPACES LATER"
```

Since you are not using variables or numbers, the semicolons are not really necessary in line **20**. But, as we mentioned earlier, it is a good habit to use them.

Run the program.

The two program lines should produce identical results.

```
HERE                    18 SPACES LATER
HERE                    18 SPACES LATER
```

With **SPC**, the number of spaces goes in parentheses right after **SPC**, and the left parenthesis must be right next to the **C**. In other words, don't leave a space after **SPC**! That's all there is to it.

TAB is another space command. It tells the computer to print at a particular space on the screen line (much as tabs work on a typewriter). For example, **75 PRINT TAB(15) "SOMETHING"** tells the computer to print the word **SOMETHING** beginning on the 15th space of the first unused screen line.

The spaces are numbered from left to right. For some reason, *the numbering begins with space number 0* and continues across the line to space number 39. This is important to remember when using **TAB.**

Type 30 PRINT "HERE"; TAB(22);"18 SPACES LATER"

List your program.

Line **30** prints the word **HERE**. Then the computer jumps to space number 22 (the word **HERE** is in spaces 0, 1, 2, and 3) where it prints the phrase **18 SPACES LATER**.

Run your program.

Box 8–1
If You Own a Printer

If you own a printer, you can make copies on paper of your program listings, tables, graphs, designs, or anything else you would otherwise put on the screen.

There are two steps you must carry out to use the printer:

1. Open up a channel of communication to the printer by giving an **OPEN** command, such as **OPEN 10,4**. The first number is just a reference number—pick any number you want. You will use this number later to refer to the channel of communication you have opened. The second number—always **4**—is the code for "printer." It tells the computer that the channel of communication is being opened up to a printer.
2. Tell the computer to use that line of communication by giving a **CMD** command with the same reference number as the **OPEN** command (for example, **CMD 10**).

Once you have done these two steps, if you tell the computer to list your program, it comes out on the printer instead of on the screen.

Also, anything you tell the computer to print is sent to the printer instead of the screen. It appears on the printer in the same format as on the screen, except that the printer has 80 spaces across. But the commas, the semicolons, and **SPC** all work the same. (Only **TAB** does not work on the printer.) So, if after us-

Making a Table

Now, finally, what is all this good for? How about a table of numbers?

Most often, computer-generated tables display the results of arithmetic calculations. The little program that follows is an example. It lays out a price table showing, for various numbers of items sold, the unit price, the total price, and the total price including 5% tax. (The items could be anything; we just made up the table as an illustration.) To make it more interesting, let's say the cost per item is $2.87 if 12 or fewer are purchased, and the price drops to $2.64 if 13 to 20 are purchased.

Clear the computer's memory and type the following:

```
10 PRINT SPC(15);"PRICE LIST"
20 PRINT
30 PRINT "NUMBER","PER UNIT","TOTAL-NET"
,"WITH-TAX"
40 FOR N=1 TO 19
50 P=2.87
60 IF N>12 THEN P=2.64
70 PRINT N,P,N*P,N*P*1.05
80 NEXT N
```

Now, list your program.

Line **10** prints the title—centered. There are 40 spaces across the screen. The words **PRICE LIST** take up 10 spaces. That leaves 30 spaces, half of which (15) would go to the left of the words to center them.

Line **20** leaves an empty screen line. Line **30** prints the column headings. Notice that since you have only four columns, you can space them (and hence the column headings) very nicely by taking advantage of the comma procedure to use different print zones.

Program lines **40** through **80** do the main work of filling in the table. Line **40** begins a **FOR. . .TO. . .** and **NEXT** loop that will go from line **40** through **80** a total of 19 times, with **N** equal to **1** the first time, and subsequently increased by **1** each time until it gets to **19.** (Since the computer prints a line each time, you have to limit yourself to **19**, or the whole table won't fit on the screen at once.)

Program lines **50** and **60** determine what price will be applicable. Line **50** sets the price—which we are calling **P**—at what it is if there are 12 items or fewer. At line **60,** if there are more than 12 items, then the price is reset at **2.64**.

Line **70** does the real work of making the table. It first prints whatever **N** is equal to (in the **NUMBER** column). Then, because of the comma, it prints the unit price (**P**) in the next print zone of the same line. In the third print zone, it puts the result of multiplying **N** by **P**, the total net price for that many items. This is simply the number of items times the unit price. Finally, in the fourth print zone, it puts the price with tax—that is, it takes the total net (**N*P**) and multiplies it by **1.05** (the cost if you add 5% to the total).

Run the program.

ing **OPEN** and **CMD**, you run a program that produces a table, the table will be printed on the printer instead of the screen.

Actually, you can include the **OPEN** and **CMD** commands right in the program as numbered program lines.

To get the computer back to writing on the screen again, do the following:

1. Tell the computer to stop using that line of communication by typing **PRINT#**, a type of **PRINT** command just for printers. Have it specify the reference number, such as **PRINT#10**, but nothing else.[1]
2. Tell the computer to close off that line of communication with a **CLOSE** command followed by the reference number and the number for printer **(4)** (for example, **CLOSE 10, 4**.)

[1]You can also use **PRINT#** like an ordinary **PRINT** command (for example, **PRINT#10, "PRICE LIST"**) and what follows it will come out on the printer. If you do that, you do not need to use **CMD**. But you cannot **LIST** that way. Furthermore, **PRINT#** closes the line of communication after it is finished printing. To use another **PRINT#** command, you must use **CMD**. This becomes very unwieldy in a program.

Graphs

A picture is worth a thousand words, and a business report is much more effective with a graph. Later, you will learn techniques that will let you make graphs very efficiently with your computer. Now, we will introduce some simple ideas. Although they are a little tedious to use, they are easy to understand.

There is no rule that you must type all of each numbered line in order. Why not type a screenful of numbered **PRINT** lines? On each line, put an opening quote mark and space across so that the closing quote mark is almost at the far right edge. Then design your graph in the space between the two columns of quotes. When you are done, press [RETURN] on each line (the cursor can be anywhere on the line). *Voila!* You have a program that reproduces your graph.

It isn't terribly elegant, but at least you see exactly what you'll be getting as you lay it out—unlike using **SPC, TAB,** commas, and semicolons.

Let's take it step by step.

Clear the computer's memory and type in the program below. The closing quotation mark should be *one space from the right edge of the screen.* Also, be sure to use the space bar rather than the cursor-arrow key to move to the right for the closing quote mark. If you use the cursor-arrow key after an opening quote mark, the computer sets up those symbols you learned about in Chapter 4 that remind the computer to print over or up or down that number of spaces. Press [RETURN] to get to the beginning of the next line.

```
10 PRINT"                                                "
11 PRINT"                                                "
12 PRINT"                                                "
13 PRINT"                                                "
14 PRINT"                                                "
15 PRINT"                                                "
16 PRINT"                                                "
17 PRINT"                                                "
18 PRINT"                                                "
19 PRINT"                                                "
20 PRINT"                                                "
21 PRINT"                                                "
```

Clear the screen and then list your program.

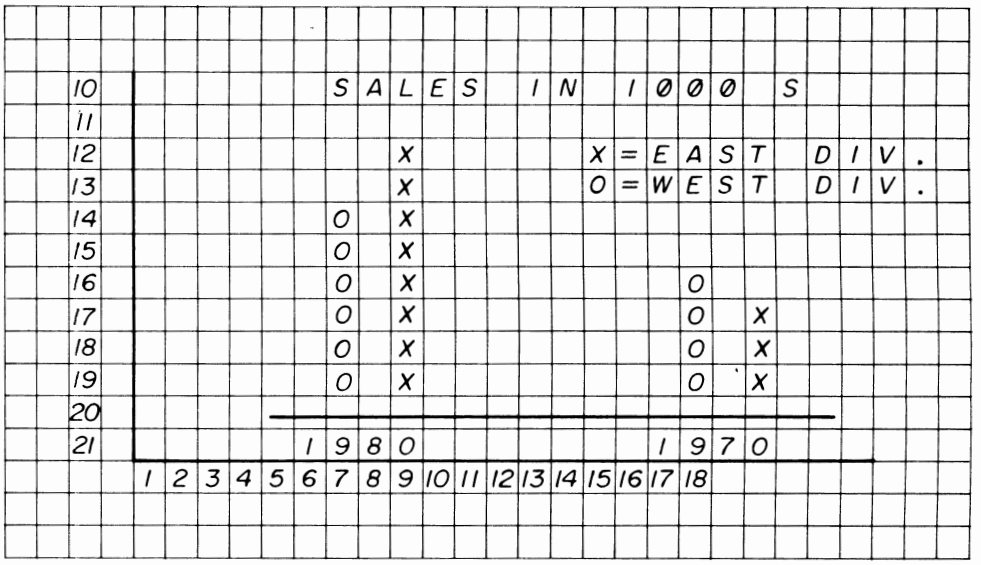
Notice that in the past, we usually left a space between the **T** in **PRINT** and the quote mark. That is not required. We chose not to do so this time to allow more room to make the figure on the screen. Also, we usually number program lines leaving some room between numbers. In this case, however, we would never want to insert program lines because that might mess up the figure, graph, or table that's been drawn in. Finally, we did not put the closing quote mark in the very last space on the right because when a program line ends in that space and you press [RETURN], the next program line—for a very complicated, screwball reason—begins two lines below!

Now that you have laid out your canvas, you can begin creating your art. Using the up-down and right-left cursor-arrow keys to move the cursor around, type whatever you want on the part of the screen enclosed by quotes. (Since you have closed quotes, the cursor moves without leaving those funny symbols.)

Actually, though, rather than work directly on the screen, it is easier to draw your figure on graph paper first. The following is an example of what can be done.

Now, From the graph paper design, put the letters and symbols in the program as shown. Remember, type it in the spaces in the PRINT lines already in your listing. (Don't type new PRINT lines.) Also, it's easier to use the cursor-arrow keys to move around. Avoid using the [INST/DEL] key. To correct errors, just go back and type the new letter over the mistake. To get you started, the S in SALES begins six spaces from the left quote mark.

PLOTTING A GRAPH




```
10 PRINT"       SALES IN 1000'S          "
11 PRINT"                                "
12 PRINT"         X     X=EAST DIV.      "
13 PRINT"         X     O=WEST DIV.      "
14 PRINT"       O X                      "
15 PRINT"       O X                      "
16 PRINT"       O X         O            "
17 PRINT"       O X         O X          "
18 PRINT"       O X         O X          "
19 PRINT"       O X         O X          "
20 PRINT"      ------------------------  "
21 PRINT"        1980        1970        "
```

Move your cursor to the top line and press [RETURN]. Do the same for each line going down the screen.

This is very important. If you don't do it, your program will not be recorded in memory.

Perhaps you wondered about the emphasis on this procedure of first typing the quotes and blank spaces and then typing the design. With any other method, you would probably have run into a problem. For example, as we pointed out, without closing quotes first, the cursor makes those special directional-marker symbols instead of actually moving right or left or up or down.

List your program.

If some numbered line or its part of the design does not show up in the listing, you probably did not press [RETURN] on that line. Do the line again and press [RETURN]. Then list again to make sure it is right.

Run the program.

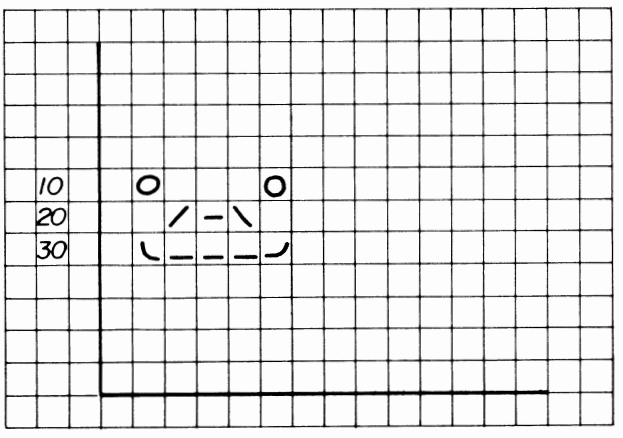
```
SALES IN 1000'S

   X      X=EAST DIV.
   X      O=WEST DIV.
O  X
O  X
O  X          O
O  X          O  X
O  X          O  X
O  X          O  X
-------------------------
 1980          1970
```

Making Designs

To make a picture, you can use the same procedure you just used. First, look at the following design drawn on graph paper.

PLOTTING A GRAPHIC DISPLAY



Use NEW to clear the computer's memory of the old program, then follow the procedure of the previous section to type this design into the following program lines:

```
10 PRINT" O   O                    "
11 PRINT"  /-\                     "
12 PRINT" \___/                    "
```

The symbols in line **10** are just the letter [O]; the symbols in line **11** are [SHIFT] with [N], minus sign, and [SHIFT] with [M]; and the symbols in line **12** are [SHIFT] with [M], the Commodore key with [P], and [SHIFT] with [N].

List the program.

Run the program.

This method of making designs does not allow color, and you can't use the whole screen because of the program line number, the quotes, and the word **PRINT.**

If you want to design extensive business graphs, games, or anything involving complex or color video displays, you will want to learn the graphic procedures taught in Part III of this book. You have, however, learned enough in this chapter to program the kind of simple design or graph that will do quite well for most other practical or fun purposes.

Sit back and relax a minute. This chapter completes Part I of this book. You have learned the fundamentals of using your Commodore 64. We now invite you to build on these fundamentals in the following chapters. The next chapter begins Part II, which is about more advanced programming techniques. It introduces the programming tools for that very important use of computers called *database management*—the big time! So, take a minute to appreciate how much you've learned and how easy it's turned out to be, and then let's get on with it.

Summary

This chapter introduces procedures to lay out what you want displayed on the screen and to apply these procedures to make tables, simple business graphs, and other designs.

The main procedure for determining where things appear on the screen involves the punctuation at the end of a **PRINT** instruction.

If there is no punctuation at the end of a **PRINT** line, the material in the next **PRINT** instruction is printed on the next unused screen line down. A program line that has the word **PRINT** with nothing at all following it (for example, **20 PRINT**) does not put any letters on the screen—it just leaves a blank line.

If there is a semicolon at the end of a **PRINT** instruction (for example, **10 PRINT "LAYOUT";**), the material in the next **PRINT** instruction begins on the same screen line, immediately next to the last character printed.

Each 40-space screen line is divided into four 10-space *print zones*. If there is a comma at the end of a **PRINT** instruction (for example,**10 PRINT "LAYOUT",**), the material in the next **PRINT** instruction begins in the next print zone on the same screen line.

Semicolons and commas may be used within a **PRINT** instruction as well as at the end of one. For example, **10 PRINT "LAYOUT";"——#1"** would result in **LAYOUT——#1** appearing on the screen. **10 PRINT "LAYOUT","——#1"** would result in **LAYOUT ——#1** appearing on the screen.

SPC in a **PRINT** instruction tells the computer to leave a certain number of spaces when it prints. For example, **10 PRINT "LAYOUT";SPC(7); "NUMBER 1"** results in a screen display with exactly seven spaces between the words **LAYOUT** and **NUMBER**. (It is a good habit to use semicolons because for complicated reasons they are sometimes essential.)

The spaces of each screen line are numbered from 0 to 39, from left to right. **TAB** and a number in parentheses means print the following item beginning at a particular numbered space across the screen. For example, **10 PRINT "THIS";TAB(25);"LAYOUT"** results in a screen display in which **THIS** appears at the far left and **LAYOUT** begins at space 25.

You can generate tables using arithmetic. The numbers can be followed by commas to set up four columns in the four print zones automatically. Thus, **60 PRINT A,A+1,A*6,A/B** would display the value of **A** at the left edge of the screen, the value of **A+1** in the second print zone (beginning in the 11th space), the value of **A*6** in the third print zone (beginning in the 21st space), and **A/B** in the fourth print zone (beginning in the 31st space). As the program changes what **A** and **B** are equal to and returns to line **60** to print, the numbers change but the columns remain straight.

Business graphs can be produced by typing in a series of numbered lines with the line number, **PRINT**, an opening quote mark, spaces to almost the other side, and a closing quote mark. You then "draw" your graph, using the keyboard, right on the screen in the blank spaces between the quotes. (This is easier if you have first drawn your graph on graph paper.) When you are finished, be sure to press [RETURN] for each line. Your graph is written into the program.

Simple designs can be programmed with this procedure. It does not, however, permit the use of color. Some techniques for producing more complex graphics are described in Part III of this book.

Terms and Concepts Introduced in Chapter 8

Using **PRINT** without punctuation
Using **PRINT** with a semicolon
Using **PRINT** with a comma
SPC
TAB
Making tables with **PRINT**
Making graphs with **PRINT**
Making designs with **PRINT**

Practice Exercises

(Answers in Appendix A.)

1. Write a program that produces the following on the screen:

```
HERE            AND

     THERE

      AND

   EVERYWHERE
```

2. Write a program that produces a table that gives the inches, feet, and yard equivalents of different numbers of meters, from 1 to 10. The table should come out like this:

METERS	INCHES	FEET	YARDS
1			
2			
3		(Have the computer	
4		fill this in)	

The conversion formulas are:

1 meter = 39.37 inches
1 meter = 3.282 feet
1 meter = 1.094 yards

Use the price table (without the **IF . . . THEN . . .** lines) as a guide.

3. Draw a simple design on graph paper and write a program that makes it appear on the screen.

PART II
More on Programming

PART II CHAPTER 9

Penetrating the Mysteries of Subscripted Variables

In previous chapters, you learned the fundamentals of computer programming. In this chapter, you will begin learning more advanced procedures.

We will start with what are called *subscripted variables.* Then we will explore programs that file and sort information—the basics of *database management,* an awe-inspiring term. Well, rest assured that the only cause for awe is the work database management can do. The ideas behind it are very simple.

What Is a Subscripted Variable?

So far, you have used letters (variable names) to stand for numbers in a variety of situations—**B** for your checkbook balance, for example, or **F** for the number of feet to be converted to meters. Although the word *variables* sounds very mathematical, we don't use them in a mathematical sense. We use them mainly to keep track of information and numbers. Everybody can use that kind of help! If we have a lot of information to keep track of with a lot of different letters, we will need an orderly system of naming the letters. Let's discover one:

The following little program should be easy for you to understand:

Clear the computer's memory (with NEW) and type the following:

```
30 A=7
40 B=8
50 C=4
70 SM=A+B+C
90 PRINT SM
```

List the program.

Line **30** tells the computer to remember that **A** equals **7**. Line **40** sets **B** equal to **8**, and line **50** sets **C** equal to **4**. Line **70** sets **SM** (for "sum") equal to the sum of these three variables. And line **90** prints out **SM**.

Run the program.

The screen should display the result—**19**.

In this example, **A**, **B**, and **C** each stand for a number. They are variable names. Another way of making variable names is to use the same letter with different subscripts after it. The **A**, **B**, and **C** of our example might instead have been written Z_1, Z_2, and Z_3. Speaking out loud, you would call these "Z sub 1," "Z sub 2," and "Z sub 3." As you probably figured out, they are called *subscripted* because something is written *(scripted)* slightly below *(sub)* the regular writing line.

However, your dear old Commodore 64 (like most computers) cannot easily display a number slightly below the line. So, the standard computer procedure is to put the subscript in parentheses after the letter or name—for example, **Z(1)**, **Z(2)**, and **Z(3)**.

Subscripted variables work like the variables you are already used to. Each stands for a number. Thus, you can substitute subscripted variables for ordinary variables, and everything will work out exactly the same.

Edit your program by substituting Z(1) for A, Z(2) for B, and Z(3) for C. When you are finished, the program should look like this:

```
30 Z(1)=7
40 Z(2)=8
50 Z(3)=4
70 SM=Z(1)+Z(2)+Z(3)
90 PRINT SM
```

List and then run your program.

The result should again be **19**.

So that's subscripted variables. Big deal, huh? And what are they good for? We're getting to that.

The DIM Instruction

There is one special requirement for subscripted variables. If you are going to use more than 10 subscripts with the same letter, you have to let the computer know in advance so it can set aside enough room in its memory. You do this with a numbered program line that begins with **DIM** (for dimension) and then gives the letter to be subscripted followed by the number of subscripts you will use with it (in parentheses)—for example, **10 DIM X(50)**.

Our example program only involves three subscripts for the letter **Z**. So a **DIM** line is not really necessary. Nevertheless, it is a good habit to use **DIM** when you use subscripted variables. In this case, it would look like this:

Type 10 DIM Z(3)

List your program.

```
10 DIM Z(3)
30 Z(1)=7
40 Z(2)=8
50 Z(3)=4
70 SM=Z(1)+Z(2)+Z(3)
90 PRINT SM
```

Run your program.

The result should be unchanged. You could have dimensioned **Z** for more than 3. **DIM Z(200)** would have worked fine, except that you would be reserving excessive space in memory. When you dimension, remember that the Commodore 64 has lots of memory, and if you dimension too small a number (or use more than 10 subscripts for the same letter and forget to dimension at all), you will get an error message instead of the output you want when you run your program.

Variables within Variables

Here's the payoff for all this: Although subscripted variables can do everything ordinary variables can do, the reverse is not true!

A subscripted variable's most important special ability is what we might call "putting a variable within a variable." That is, you can use another variable to stand for the *subscript* of a variable so that the value of the subscript can change. Instead of **Z(1)**, **Z(2)**, and **Z(3)**, we could write **Z(J)**. That means the value of **J** could vary among **1**, **2**, and **3**. When **J** equals **2**, **Z(J)** is the same as **Z(2)**, but when **J** equals **3**, then **Z(J)** is the same as **Z(3)**. Clever? It becomes clear when you see it used in the next program.

Type the following program lines:

```
20 SM=0
60 FOR K=1 TO 3
70 SM=SM+Z(K)
80 NEXT K
```

List your program.

The program should now look like this:

```
10 DIM Z(3)
20 SM=0
30 Z(1)=7
40 Z(2)=8
50 Z(3)=4
60 FOR K=1 TO 3
70 SM=SM+Z(K)
80 NEXT K
90 PRINT SM
```

```
10 DIM Z(3)
20 SM=0
30 Z(1)=7
40 Z(2)=8
50 Z(3)=4
60 FOR K=1 TO 3
70 SM=SM+Z(K)
80 NEXT K
90 PRINT SM
```

Line **20** sets **SM** equal to **0**. (You'll see the point of this in a minute.) Line **60** begins a **FOR . . . TO . . .** and **NEXT** loop that, the first time through, sets **K** equal to **1**. At line **70**, the computer sets **SM** equal to its previous value (which is **0**) *plus* whatever **Z(1)** equals. That is, it adds **0** plus **7** (which is what **Z(1)** equals), and **SM** now equals **7**.

Now, you can see why we put in line **20**—to start **SM** out equal to **0** before we added things to it. Actually, in this case, your Commodore 64 would have assumed that, since you'd never used **SM** before, it is equal to **0**. But it is a good programming habit never to use a variable that has not been set equal to something. Among other reasons, this helps you keep track of what you are doing.

Line **80** is **NEXT K**. Thus, the computer goes back to line **60**, where it makes **K** equal to **2** and goes on to the next line. This time, when it gets to line **70**, it sets **SM** equal to the previous value of **SM** (which is now **7**) *plus* what **Z(2)** equals. That is, it adds **7** and **8**. After passing line **70** this second time, **SM** equals **15**.

Then the computer comes to line **80** the second time. It again goes back up to line **60**. This time, it sets **K** one number higher still—to **3**—and proceeds as usual to line **70**. There, it resets **SM** to the total of the preceding value (which we just saw is **15**), adds **Z(3)** (which is **4**), and resets **SM** again, at **19**.

The third time the computer comes to line **80**, it does not go back up to line **60**. This is because line **60** said to continue for **K** equaling **1**, to **K** equaling **3**. (You will recall from earlier chapters that this is the meaning of **FOR K=1 TO 3**.) The limit has been reached, and there is no need to go back again. The computer automatically goes on to the next line in order, which is line **90**. There it prints **SM**, which is now **19**.

Run the program.

The result should be **19**.

As a review, study the variable table for this little program.

Variable Table for Example Program

LINE #	SM	Z (1)	Z (2)	Z (3)	K
10	—	—	—	—	—
20	0	—	—	—	—
30	0	7	—	—	—
40	0	7	8	—	—
50	0	7	8	4	—
60	0	7	8	4	1
70	7	7	8	4	1
80	7	7	8	4	1
60	7	7	8	4	2
70	15	7	8	4	2

LINE #	SM	Z (1)	Z (2)	Z (3)	K
8Ø	15	7	8	4	2
6Ø	15	7	8	4	3
7Ø	19	7	8	4	3
8Ø	19	7	8	4	3
9Ø	19	7	8	4	3

In this example, subscripted variables made your program longer. From now on, they will make your programs shorter. Sometimes, without subscripted variables a program would be just about impossible.

Applying Subscripted Variables—the Idea Behind Database Management

Subscripted variables make it easy to keep track of information—data—in a systematic way. For example, suppose you want to keep track of customers. You might have one variable—say, **R**—that is the customer's region of the country; one—perhaps **C**—that is the customer's credit rating; one that represents which product line that customer prefers—say, **P**; one that indicates which salesman usually deals with that customer—perhaps **S**; and so on.

This works fine for one customer. But what if you have several hundred? The first customer's information uses up **R**, **C**, **P**, and **S**. The second uses, say, **H**, **I**, **J**, and **K**. Soon, it's alphabet soup. What letters go with what information and which customer?

With subscripted variables, **R(1)** could be the region for the first customer; **C(1)**, the credit rating for the first customer; **P(1)**, the product line for the first customer; and **S(1)**, the salesman for the first customer. Thus—see the beauty of it?—**R(2)** would be the region for the second customer; **C(2)**, the credit rating for the second customer, and so forth. The same letter always stands for the same type of information, and the particular subscript tells you which customer that information applies to!

This kind of orderliness makes it possible to do all kinds of systematic record keeping, as you can imagine. You could ask the computer for information of any kind on any particular customer, or for lists of customers that fall in certain categories—for example, all customers using a particular product line, for all customers of a particular salesman, or for all customers with a certain credit rating buying from a particular salesman. Anything!

With subscripted variables, it is also possible to create any kind of ordering or classification pattern—for example, you could create a list of customers laid out from best to worst credit rating,

or a table of the percent of customers of each salesman buying each product line. And so on.

Using your computer this way is called *database management.* A "database" is your information, and the programs you write (or buy) "manage" it by filing data and sorting through it in various systematic ways. Database management is a fancy term for a computerized filing system. Without subscripted variables, it would be extremely cumbersome.

An Example of a Database Management Program

The following program keeps track of a coin collection. For simplicity's sake, we have left out some program lines, and we will only keep track of five coins. But the program listed and the number of coins are sufficient to show you how a database program actually works.

Clear the computer's memory and type the following:

```
10 DIM FV(5),Y(5),MC(5)
20 FOR K=1 TO 5
30 PRINT "TYPE IN FACE VALUE, YEAR, AND
 MINT CODE, FOR COIN #";K
40 INPUT FV(K),Y(K),MC(K)
50 NEXT K
60 PRINT "TYPE 1 TO LIST ALL COINS, 2 TO
LIST COINS OF A PARTICULAR FACE VALUE,"
70 PRINT "3 FOR A PARTICULAR YEAR, OR 4
FOR A PARTICULAR MINT CODE"
80 INPUT L
90 IF L=1 THEN GOTO 150
100 IF L=2 THEN GOTO 200
```

(Remember, we're leaving out some lines—line **150** for what to do if **L=1**, and those that would follow from that, plus all the lines that deal with what to do if **L=3** or **L=4**. They would go here.)

```
200 PRINT "LIST COINS OF WHICH VALUE"
210 INPUT WV
220 PRINT "FACE VL","YEAR","MINT CODE"
230 FOR J=1 TO 5
240 IF FV(J)=WV THEN PRINT FV(J),Y(J),MC(J)
250 NEXT J
260 PRINT
270 GOTO 60
```

List and check your typing. (With a program this long, you will probably get a couple of error messages before you get your typing perfect. That's normal.)

```
10 DIM FV(5),Y(5),MC(5)
20 FOR K=1 TO 5
30 PRINT "TYPE IN FACE VALUE, YEAR, AND
 MINT CODE, FOR COIN #";K
40 INPUT FV(K),Y(K),MC(K)
50 NEXT K
60 PRINT "TYPE 1 TO LIST ALL COINS, 2 TO
LIST COINS OF A PARTICULAR FACE VALUE,"
70 PRINT "3 FOR A PARTICULAR YEAR, OR 4
FOR A PARTICULAR MINT CODE"
80 INPUT L
90 IF L=1 THEN GOTO 150
100 IF L=2 THEN GOTO 200
200 PRINT "LIST COINS OF WHICH VALUE"
210 INPUT WV
220 PRINT "FACE VL","YEAR","MINT CODE"
230 FOR J=1 TO 5
240 IF FV(J)=WV THEN PRINT FV(J),Y(J),MC(J)
250 NEXT J
260 PRINT
270 GOTO 60
```

Now, let's analyze this program line by line. Line **10** sets aside space in memory for three subscripted variables of five subscripts each.

Lines **20** through **50** are a **FOR . . . TO . . .** and **NEXT** loop that the computer goes through five times. The first time, **K** equals **1**, the second time **K** equals **2**, and so forth, up to **5**.

Each time through the loop, the computer comes to lines **30** and **40**, which is a prompt-and-**INPUT** sequence. The prompt asks you to type in information about one of the coins. It even tells you which one by putting what **K** equals at the end of the prompt. Thus, the first time through, since **K** equals **1**, the end of the prompt appears on the screen as **FOR COIN # 1**.[1] The second time through the loop, the end of the prompt appears as **FOR COIN # 2**, and so forth. Nice, huh?

The point of this prompt-and-**INPUT** sequence is that the computer gets the information about your coins—the database—so that it can file the information for future use.

The prompt asks for three pieces of information about each coin—its face value (**.10** for a dime, **.25** for a quarter, etc.), its year, and its mint code.

One way to handle typing the three different pieces of information is to press [RETURN] after each number. That is, for each coin, type the coin's face value, then press [RETURN]; type the year, then press [RETURN]; type the mint code number, then press [RETURN]. (If you follow this procedure, after pressing [RE-

[1] You may wonder about the space before the number. The Commodore 64 always puts a plus or minus sign in front of a number—but the plus sign is left invisible. It just takes up a space!

TURN] for the face value and for the year, the computer will display **??** to let you know it is still waiting for more information about this coin.)

There is another method for putting in three numbers for a single **INPUT** line: type the first number, then type a comma, type the second number, then a comma, type the third number, and *then* press [RETURN].

40 INPUT FV(K),Y(K),MC(K) makes these variables equal to the numbers you type. The first number you type is assigned to **FV(1)**. This is because **FV(K)** is the first variable named, and **K** at that point equals **1**. The second number you type, the year of the first coin, is assigned to **Y(1)**. And so on.

The second time through the loop, you are entering information **FOR COIN # 2**. The computer sets **FV(2)** equal to the first number you type, **Y(2)** equal to the second number you type, and so on. This procedure continues until you have given the required information for all five coins. The computer will then have internally filed this information under the five subscripts of each of the three variable names. A rather tidy filing system!

The next section of the program allows you to tell the computer how you want it to give back the information.

Lines **60**, **70**, and **80** are two prompt lines and an **INPUT** line. Here's where you tell the computer what kind of listing you want. Because the prompt was so long, it was necessary to use two **PRINT** lines—the maximum length for any numbered program line is 80 spaces. But since the two **PRINT** instructions are one right after the other, the material in the second will come out right below the material in the first. Therefore, it appears as one prompt.

This one prompt gives you the choice of telling the computer to list all the coins or to list just those coins of a particular face value, a particular year, or a particular mint code. When you type the appropriate number (**1** for **LIST ALL COINS**, **2** for **LIST COINS OF A PARTICULAR VALUE**, etc.), line **80** sets **L** equal to that number.

Lines **90** and **100** check what **L** is equal to. If it is equal to **1**, the computer goes to line **150**, where (if we had given you the whole program), there would be program lines to print out a table of the information for all the coins. To simplify your typing, we are using as our example only what happens when **L** equals **2**. (If **L** equals **3** or **4**, the **IF . . . THEN . . .** lines (perhaps lines **110** and **120**) would send the computer to appropriate sections of the program to print out lists for particular years or mint codes. The same principles would apply when **L** equals **3** or **4** as when **L** equals **2**.

So, what happens if **L** equals **2**—that is, if you typed a **2** at the last prompt, indicating you wanted a list of coins of a particular face value? The equation **L=2** in line **100** would be true, and the computer would then go to line **200**.

Line **200** is a prompt that first clears the screen (to get rid of the old prompt from lines **60** and **70**) and then asks the next logical question: If you want a list of coins of a particular face value, which face value is it going to be? Dimes? Quarters? Line **210** then sets **WV** (short for "which value?") equal to whatever you type at this point.

By the time the computer has finished with line **210**, you've done three things. You've put in the information for all the coins (at lines **20** through **50**). You've told the computer you want a listing of just those coins of a particular value (by your response to the prompt of lines **60** and **70**, which was registered by line **80**). And, finally, you've told the computer what specific face value of coin you want included in that listing (by your response to the prompt of line **200**, registered in line **210**).

Of course, on a particular day you might only be adding data to the files, or only asking for lists, or even deleting or editing data. This time, we are trying out everything the program can do.

The computer must act on your desires. It sorts the information to select just those coins you want and prints out a table of them.

Line **220** clears the screen of the last prompt and then prints headings for a table of coins. Didn't think of that, huh? No matter which coins you select to be displayed in a table, you will want headings for the table. The layout of the headings is very simple: Each heading begins a new print zone on the screen, as indicated by the commas separating the quoted headings. (Remember, each screen line is divided into four print zones of 10 spaces each, as we discussed in Chapter 8.)

230 FOR J=1 TO 5 begins a loop that the computer goes through five times (once for each coin). **J** (we could have picked any letter) equals **1** the first time, **2** the second time, and so forth.

The important line in this loop is line **240**. It compares each coin's actual face value to the number you typed. That is, line **240** begins by checking whether **WV** (the value of coins you want listed) is the same as **FV(J)**. As the computer makes its five trips through the loop and **J** goes from **1** to **5**, **FV(J)** becomes the value of each coin and is checked against **WV**.

If on any particular time through this loop it's true that **WV** is equal to a coin's face value, **FV(J)**, then the computer is instructed to print that coin's face value, year, and mint code. The output for each variable is in the same format as the headings, since, like the headings, the three variables are separated by commas.

The result? Only those coins that have a face value that matches the value you asked for are listed on the screen, with all their data under tidy headings.

After the computer has completed its five times through the **230**-through-**250** loop, it proceeds to the next line in order. Line **260** just prints an empty line to leave space at the end of the table. Line **270** tells the computer to go back to line **60**. At line **60**, the computer gives you a chance to list your coins in some other way, by starting again with the prompt that asks you what category of listing you want.

Let's review. You learned in Chapter 7 that most programs have three parts: taking in information, doing something with it, and giving out the result. This pattern is also true for a database management program. However, for this kind of program (as with many others), the first aspect—taking in the information—has two subparts: taking in the database—the information to be kept on file, and taking in your request for what you want done with it—your selection of the kind of output wanted on that particular running of the program.

This coin collection example illustrates this pattern. Lines **20** through **50** allow you to put in the database—in this case the face value, year, and mint code for each of five coins. The next group of lines (**60 to 210**) allows you to tell the computer what kind of listing you want, beginning with the general category of listing you want. If you choose a listing other than for all the coins, it asks what kind of coin you want listed—for example, which particular face value.

The last few program lines combine the second two main parts of a program. The computer is instructed to go internally through the coins, one by one, checking each coin to see whether it fits the category you want. This is the "doing-something-with-it" part. Then, it displays the coins that fit the category on the screen. That's the third, or output, part.

An outline of the program would look like this:

I. Input
   A. Database (coin information)
   B. Operator's desires for output listings (what kind of listing in general, then more specifically)

II. Doing something internally with the information (checking each coin against kind of listing desired)

III. Output
   (a printed table of those coins that fit the desired category)

In order to actually run this program, you need information on some coins. Here is a sample coin collection you can use to try out this program.

Coin #	Amount	Year	Mint
1	.Ø5	1943	1
2	.Ø1	1977	4
3	.1Ø	1982	Ø
4	.25	1914	2
5	.1Ø	1935	2

Run the program. (If it doesn't work properly, and you want to stop it while it is waiting for input, remember to use [RUN/STOP] and [RESTORE].)

When you get the prompt to type the information for the coins, use the data we gave you in the table. Remember, you must put in three pieces of information for each coin, so either use [RETURN] three times or separate the items with commas.

When the computer asks how you want the coins listed, be sure to type 2 for a listing of coins of a particular face value, since you haven't yet done the parts of the program that would handle anything else.

When the computer asks what face value to list the coins by, try any value you like. Then on subsequent times through the program, use different values until you have tried them all.

The screen should show nice, neat, short tables of coins of the specified face value.

If you use **.10** for the value of coins you want listed, the result on the screen should look like this:

```
FACE VL    YEAR        MINT CODE
 .1          1982        Ø
 .1          1935        2
```

Notice, incidentally, that the value you typed in as **.1Ø** is shown on the table as **.1**. This is because the computer only uses as many decimal places as it needs—and, of course, .1 = .1Ø. It is a little confusing if you are not used to it, but there is no easy way in BASIC to get around it. There is no problem with nickles, pennies, or quarters, since **.Ø5**, **.Ø1**, and **.25** have to have two decimal places to be accurate. But if you had half dollars ($.5Ø), the computer would have shown their value as **.5**.

When you have tried this program enough, stop it by pressing [RUN/STOP] and [RESTORE] at the same time.

To start up the program again, type **GOTO 6Ø** and press [RETURN]. It picks up right where you left off—and it even remembers all the coin information you typed in earlier! If, however, you start up the program again using the **RUN** command, all the coin information you typed will be cleared out of memory, and you will

have to put it in again. Moreover, if you save your program on tape or disk, the information will be lost when you load it back. (Want to reread those last two sentences?)

All of this is not so bad when you have only five coins with three pieces of information about each. You just type it again when you run it again. But suppose you had 500 coins with 10 pieces of information about each?

Writing Information Permanently into a Program—READ and DATA

One way to avoid having to type your information each time you use a program of this kind is to include the information—the database—permanently in the program itself. (This does not mean you cannot ever edit or delete any data: it just means you won't have to type from scratch again.)

Permanent input is provided most efficiently with **READ** and **DATA** program lines. You put the information in a **DATA** program line—for example, **1200 DATA 12.3. READ** then works very much like **INPUT**, except that it reads **DATA** lines for its information rather than taking data from what you type.

A program line such as **25 INPUT X** tells the computer to stop and wait for you to type a number. It sets **X** equal to that number and goes on to the next line. Similarly, when the computer finds a line such as **25 READ X**, it looks through the program for a **DATA** line, sets **X** equal to the first number in that **DATA** line, and then goes on. If it has previously read one or more numbers in a **DATA** line, it takes the first unused number.

Again, a **READ** instruction searches for the first number in the first (by numerical order, of course) **DATA** line anywhere in the program. The next time the computer comes to a **READ** instruction, it remembers what numbers it has already used from **DATA** lines and automatically goes to the first *unused* number in the first **DATA** line. If none is left in that line, it goes to the next **DATA** line.

You can use as many **DATA** lines as you like, with as much information in them as you like, each number separated from the next by a comma. (The only limit is that the total length of any program line cannot be more than 80 spaces.) Although they can go anywhere, **DATA** lines usually are put at the very end of a program.

Let's modify your coin collection program to use the **READ**-and-**DATA** procedure instead of the **INPUT** procedure you used before.

Edit line 40 to read 40 READ FV(K),Y(K),MC(K)

Add the following program lines:

```
1000 DATA .05,1943,1,.01,1977,4,.10
1010 DATA 1982,0,.25,1914,2,.10,1935,2
```

Actually, you could have made one long **DATA** line with all the numbers on it, since the data did not take up 80 spaces. Or you could have made 15 **DATA** lines, each with one number. Or, for the sake of remembering what you are doing, you might have wanted to use five **DATA** lines, one for each coin. As long as the information is in order, the computer doesn't care how many or how few **DATA** lines you break the information into.

Delete line 30. Remember, you can just type 30 on a new line and press [RETURN].

Line **30** was a prompt for the **INPUT** instruction at line **40**, which is now a **READ**-and-**DATA** procedure. The prompt is no longer needed and would be confusing if it appeared on the screen.

List your program and check it for typographical errors.

The program should now look like this:

```
10 DIM FV(5),Y(5),MC(5)
20 FOR K=1 TO 5
40 READ FV(K),Y(K),MC(K)
50 NEXT K
60 PRINT "TYPE 1 TO LIST ALL COINS, 2 TO
LIST COINS OF A PARTICULAR FACE VALUE,"
70 PRINT "3 FOR A PARTICULAR YEAR, OR 4
FOR A PARTICULAR MINT CODE"
80 INPUT L
90 IF L=1 THEN GOTO 150
100 IF L=2 THEN GOTO 200
200 PRINT "LIST COINS OF WHICH VALUE"
210 INPUT WV
220 PRINT "FACE VL","YEAR","MINT CODE"
230 FOR J=1 TO 5
240 IF FV(J)=WV THEN PRINT FV(J),Y(J),MC(J)
250 NEXT J
260 PRINT
270 GOTO 60
1000 DATA .05,1943,1,.01,1977,4,.10
1010 DATA 1982,0,.25,1914,2,.10,1935,2
```

Run your program.

It should work exactly the same as before, except that you no longer have to put in the information. Also, if you stop the program (with the [RUN/STOP] and [RESTORE] keys), you can start it up again at any time with **RUN** and your coin collection is still remembered. Most important, you can save your program on tape, and later load it back from the tape. The information will still be in it.

The **READ**-and-**DATA** procedure also makes it easier to correct mistakes and add and delete entries. You just edit the **DATA** lines. Furthermore, you can see the data for your collection when you list the program.

In general, when you write programs to keep track of your collections, inventories, mailing lists, and other kinds of information to be used repeatedly, use the **READ**-and-**DATA** procedure. The **INPUT** procedure is better for information that changes each time you use the program (such as checks and deposits for balancing your checkbook each month).

In the next chapter, you will learn about variables that stand for words, letters, symbols, and the like. These are called *string variables.*

Congratulations. You have just been initiated into the Secret Society of Database Managers!

Summary

This chapter considers subscripted variables and their application in programs that file and sort information.

A variable is a letter or name that stands for a number. A subscripted variable is a letter or name, *with a number after it in parentheses,* that stands for a number. For example, ordinary variables such as **A**, **B**, or **C** might stand for the numbers **7**, **8**, and **4**, respectively. In the same way, **Z(1)**, **Z(2)**, and **Z(3)** could be used to stand for these numbers.

If any variable, however, is to have more than 10 subscripts (the numbers in parentheses), you must tell the computer how much memory space to set aside for it. Do this with a **DIM** (for "dimension") instruction. For example, **5 DIM AA(2Ø)** sets aside space for up to 20 subscripted variables with the main name **AA**.

Subscripted variables permit a "variable within a variable." For example, **Z(K)** would be considered **Z(1)** when **K** equals **1**, or **Z(2)** when **K** equals **2**, and so on.

Subscripted variables simplify the writing of programs that file and sort information—what is called *database management.* For example, two different kinds of information about the same customer can be kept track of by using different main variable names (such as **R** for region of the country and **C** for credit rating) with the same subscript (for example, **R(8)** and **C(8)** for customer #8).

A program for a small coin collection illustrates how the three main parts of any program apply to database management. *Input of information* has two aspects: taking in the information to be kept on file, and asking each time the program is run what specific kind of listing to produce (such as all the coins of a particular value). *Doing something internally with the information* consists of going through all the items on file and checking each to see if

it should be included in the listing requested (such as whether each coin is of the face value you specified). Finally, *output of results* involves printing out the requested list.

READ and **DATA** program lines allow you to include information directly in the program instead of having to retype the information each time you run the program. The information is included in one or more **DATA** lines. For example, **READ 1000 DATA .05,1943,1,.01,1977,4,.10** works like **INPUT**, except that instead of taking in what you type, it takes numbers from the **DATA** lines in the program, beginning with the first number in the first **DATA** line.

Terms and Concepts Introduced in Chapter 9

Subscripted variable

DIM

"Variable within a variable"

Database management program

READ and **DATA**

Four parts of a database management program

Practice Exercises

(Answers and comments in Appendix A.)

1. Complete the coin collection program. That is, add the lines needed to make it produce listings for all coins, for particular years, and for particular mint codes.

2. Fix the program so it can handle 12 coins and put the following additional coins into the program (in **DATA** lines):

COIN #	AMOUNT	YEAR	MINT #
6	.25	1971	0
7	.05	1926	1
8	.05	1947	3
9	.01	1951	3
10	.01	1932	2
11	.50	1925	4
12	.10	1977	1

PART II CHAPTER 10

String Variables—Keeping Track of Words and Symbols

In the previous chapter, you learned about subscripted variables and how to use them in programs that file and sort information. Variables are letters or names that stand for numbers. Subscripted variables are much the same thing, except that after the letter or name they have a subscript—a number in parentheses.

In this chapter, you will learn about *string variables*. These are letters or names that stand for words or symbols—any "string" of characters—instead of numbers. You will also learn about subscripted string variables, which—you guessed it—are string variables that have a subscript after them. Then, you will learn about two other important programming tools: random numbers and delay loops. Finally, you will learn how to use subscripted string variables, along with random numbers and delay loops, to put together a versatile educational program.

Introduction to String Variables

If a string variable stands for certain letters or other characters, how do you know it's acting as a stand-in? The answer is that there's a dollar sign **($)** at the end of the variable. String variables are so named because what they stand for is like a string of beads—a series of characters, one after the other. For example, **X$** could stand for **HI!**, **PL$** could stand for **1-2-3-GO**, or **PICT$** could stand for \\|//. (Numbers can appear in string variables, but they are not calculated; they are treated as just another type of symbol or character.)

Clear the computer's memory (with NEW) and type the following lines:

```
10 A$="COMMO"
40 PRINT A$
```

List your program.

Line **10** sets **A$** equal to the string of letters **COMMO**. Notice that when you set a string variable equal to a series of letters or characters, you must put quote marks around the string, just as when you are printing a string of characters. Line **40** instructs the computer to print the value of **A$**.

Run your program.

The letters **COMMO** should be displayed on the screen.

Stringing Strings

You can have one string displayed right after the other by using the plus sign.

Type the following two new lines:

```
20 B$="DORE 64"
30 C$=A$+B$
```

Edit line 40 to read 40 PRINT C$

List your program.

```
10 A$="COMMO"
20 B$="DORE 64"
30 C$=A$+B$
40 PRINT C$
```

Line **10**, as you have seen, sets **A$** equal to the character string **COMMO**. Line **20** sets **B$** equal to the string **DORE 64**. Line **30** directs the computer to set **C$** equal to the string that results from putting **B$** right after **A$**. That is, **C$** equals **COMMO** plus **DORE 64**. Line **40** prints **C$**. (Technically, putting two strings together like this is called *concatenating* them—but you really don't have to remember that word!)

Run your program.

Selecting Out Part of a String—MID$, Substrings, and String Slicing

You can pick out a part of a long string by typing **MID$** and then putting three things in parentheses: the name of the string

variable, a number indicating the space in the string where the part you want begins, and a number indicating how many spaces there are in the part you want.

Clear the computer's memory and type the following lines:

```
10 A$="ABCDEFGHIJK"
20 PRINT MID$(A$,3,2)
```

List your program.

Line **10** sets **A$** equal to the string **ABCDEFGHIJK**. Line **20** then tells the computer to take that part of **A$** that begins with the third character (**3**) and continues for a total of two characters (**2**).

Run your program.

The result should be CD.

Now, let's consider a more interesting example of string slicing.

Clear the computer's memory and type the following lines:

```
10 OD$="-----FIRST SECONDTHIRD FOURTH"
20 PRINT "TYPE # OF PLACE FINISHED"
30 INPUT N
40 PRINT MID$(OD$,6*N,6)
```

List your program.

Line **10** sets the string variable **OD$** (you could have used any letter or name) equal to a long string beginning with five dashes. You'll see why there are dashes in a minute. The next six spaces include the word **FIRST** with a blank at the end, the next six are the word **SECOND**, the next six spell out **THIRD** with a blank at the end, and the last six spell out **FOURTH**. Thus, each word has been made a six-space segment.

Lines **20** and **30** are a prompt-and-**INPUT** sequence that sets **N** equal to the number you type for **TYPE # OF PLACE FINISHED** (as, for example, in a horse race).

Line **40** is the important line here. It tells the computer what part of **OD$** to print. **MID$** has three things in parentheses. The first thing tells the computer that it is to slice out a section of the variable **OD$**. The second indicates that the desired slice begins in the space number that results from multiplying **N** by **6**. (Clever, huh?) The third number indicates that the slice continues for a total of **6** characters.

If you had input **1** for **N**, then **N*6** would be **1** times **6**, or **6**. In that case, the computer would take the part of the string beginning with the sixth character, which is the **F** in **FIRST**. (Now you see why the five dashes—so we could type **1** for **FIRST** and

have **FIRST** begin on the sixth space.) The computer would go on until it had six characters—which would come out **FIRST** . Notice that the blank counts as a character.

Suppose you had typed **4** at lines **20** and **30** so that **N** was equal to **4**. In that case, line **40** would print that section of **OD$** that begins with space number 24 (the result of **N*6**), and the following six spaces. If you count it out by hand, you will see this would result in **FOURTH** being printed.

Run the program several times, typing different numbers each time to see how this works.

This procedure is useful if you use certain character strings repeatedly and want to refer to them by numbers—the names of the days of the week, for example, or states, or models, and so forth. You will also find this procedure useful in creating graphics.

Subscripted String Variables

Again, subscripted string variables work exactly the same as the ordinary subscripted variables you learned about in the last chapter. Except, of course, that instead of standing for numbers, they stand for strings of characters.

For example, we could rewrite the program example at the beginning of this chapter as follows:

Clear the computer's memory and type these lines:

```
5 DIM S$(3)
10 S$(1)="COMMO"
20 S$(2)="DORE 64"
30 S$(3)=S$(1)+S$(2)
40 PRINT S$(3)
```

List the program.

Notice that, as with the numeric subscripted variables, you always begin with a **DIM** instruction to let the computer know how much space to set aside. Otherwise, the subscripted variables here work exactly the same as the nonsubscripted string variables **A$**, **B$**, and **C$** we just used.

Run the program.

The screen should show the same result as before:

```
COMMODORE 64
```

As we saw in the last chapter, subscripted variables can do everything ordinary variables can do, and much more. Subscripted *string* variables have all the same advantages—including being very useful in database management.

You may have noticed in the last chapter that our programs for filing and sorting information involved only information that

was expressed in numbers—such as a coin's value, year, and mint code. With subscripted string variables, you can do all the same things with letters, words, and designs. For example, you could have used **MC$(1)** to stand for the actual name of the mint of the first coin (such as "Denver" or "Philadelphia") instead of code numbers.

Subscripted string variables are used in programs to organize information such as names and addresses on mailing lists, lists of titles (like book titles or record titles), and vocabularies, indexes, or glossaries ("Give me all the books with 'Bliss' in their title; all references to 'vacations' in the index; all recipes using artichoke hearts"). Using string variables makes database management very flexible.

Rather than belabor database management any further, we will illustrate the use of subscripted string variables with an example of a new kind of program—a program that helps you learn something. First, we will introduce some new programming tools to write that program. These tools are important in programming in general. These new tools are *random numbers* and *delay loops*.

Random Numbers, RND(1), and INT

When you flip a coin, whether it lands heads or tails is said to be *random*. It is something you cannot predict in advance. Random results are also produced by throwing dice, spinning a roulette wheel, or recording the time between clicks of a Geiger counter.

Randomness is obviously central to all kinds of games, from determining the number of spaces in a Monopoly® move to deciding who kicks off in a football game. Nearly all computer games use randomness in some way.

Randomness is also important for many of the jobs for which computer programs can be helpful. This includes randomization procedures in scientific research and statistics, the selection of random samples of information from a large database, and, as you will see later in this chapter, educational programs.

The key to using randomness in computer programs is **RND(1)**. Your Commodore 64 has a long list of random numbers built right into it. **RND(1)** takes numbers from that list.

The numbers in the list are between **0** and **1** (that is, the lowest number is **.000000000** and the highest number is **.999999999**).

Type PRINT RND(1) and press [RETURN]. (Don't worry about the program in memory; operating in command mode doesn't affect it, and we're finished with it anyway.)

A decimal number less than 1 should appear on the screen.

Let's get a longer list of them.

Clear the computer's memory and type the following lines:

```
10 PRINT RND(1)
20 GOTO 10
```

Run your program. When the screen is full, press [RUN/STOP].

Most of the time, though, what you want is not a random number between **0** and **1**. You want something such as "between 1 and 10." To accomplish that, multiply **RND(1)** by whatever number you want to be the upper limit. For example, **RND(1)*10** would multiply whatever **RND(1)** produced by **10**. Thus, you would end up with a number between **0** and **10**. Try it:

Edit line 10 to read 10 PRINT RND(1)*10

List and then run your program. When the screen is full, press [RUN/STOP].

This should produce a column of decimal numbers between **0** and **10**.

When people say "pick a number," they don't usually mean a number like 2.3988876512. They mean whole numbers, like 1, 2, 3, and so on.

The procedure for converting a decimal number to a whole number is quite simple. You use **INT** (an abbreviation for *integer*, which means "whole number"). When you put **INT** before a decimal number, it converts the decimal number to a simple whole number by chopping off whatever comes after the decimal point. Try it:

Type PRINT INT(23.876) and press [RETURN].

The screen should show **23**.

Type PRINT INT(7.200786)

The screen should show **7**.

Type PRINT INT(67.99899)

The screen should show **67**.

Notice that **INT** does not round off the number. It just takes any decimal number and drops the decimals to make it a whole number.[1]

Accordingly, if you want to make your random numbers come out as whole numbers, just put **INT** in front of what you already had.

[1]Actually, it always "rounds down." Thus, if you give it a negative number, such as **-6.84**, it would come out **-7**.

Edit line 10 to read 10 PRINT INT(RND(1)*10)

List your program.

This time, when you run this little program, it produces a list of random numbers from **0** to **9**—but with the decimal places lopped off. (The highest number would be **9** because *almost* **10** can be no higher than something like **9.99999**, which is **9** when you drop off the decimals.)

Run the program and, when the screen is full, press [RUN/STOP].

But didn't we want random whole numbers from **1** to **10**? What about the poor person who says "10"? How do we incorporate that into the program?

All that's necessary is to add **1** to this mess.

Edit line 10 to read 10 PRINT 1 + INT(RND(1)*10)

List and then run the program.

There were a lot of steps to generate random numbers from **1** to **10**, but they were all easy and ended up expressed on one program line:

1. Begin with **RND(1)**.
2. Multiply **RND(1)** by the number that is the upper limit of the range you want.
3. Use **INT** to make whole numbers.
4. Add **1** to make it include your upper limit.

If you like formulas, the most general rule would be that random whole numbers ranging from **A** to **B** are produced by the program line **A + INT(RND(1)*(B − A + 1))**.

If you don't like formulas—that's fine. With a little practice, you will have no trouble getting the random numbers you want from the four simple rules we described previously.

Delay Loops

Even though sometimes you are not in a hurry, the computer always is.

For example, when you do a program with a **GOTO** loop, it might be nice to actually *see* what is happening on the screen instead of having things flash by. More important, there are many situations in which you want to move slowly from one screen display to another (as in an animated business graph), to pause a minute (as when you want a person to have some time to digest one thing before going on to the next), or to time the appearance of something for a precise interval (as in many games).

These and other applications are usually handled by what is called a *delay loop*. This is simply a **FOR...TO...** and **NEXT** loop that the computer goes through a large number of times, just to take up time. Here is an example of a delay loop:

Type the following program lines:

```
12 FOR H=1 TO 100
13 NEXT H
```

List your program.

```
10 PRINT 1+INT(RND(1)*10)
12 FOR H=1 TO 100
13 NEXT H
20 GOTO 10
```

The four-line program you now have is the same as what you had before. It still produces random whole numbers between **1** and **10**—they are just displayed a little more slowly. That's because the computer must pass lines **12** and **13** between producing a random number at line **10** and getting to the **GOTO** instruction at line **20**, where it is sent back immediately to produce the next random number. Each time the computer comes to lines **12** and **13**, it has to go through the little loop 100 times. The loop does absolutely nothing. It just makes the computer jog in place a while. In fact, the computer is so fast that going through a loop like this 100 times takes only a fraction of a second—but it does slow things down a little. Try it:

Run the program.

Try it again with a bigger delay.

Edit line 12 to read 12 FOR H=1 TO 1000

List and run your program.

This should go more slowly.

Try several different numbers in line **12** to get a feel for how long each loop delays the computer.

A Flashcard Program

Now you're ready to apply some of what you have learned in this chapter to something practical—an educational program that you should be able to adapt to your own purposes right away.

The program below is sometimes called a *flashcard program* because it does essentially what flashcards do—it gives you a question, you give what you think is the answer, and then you find out whether you are correct or not.

Clear the computer's memory and type the following lines.

```
10 DIM ENG$(5),FR$(5)
20 FOR M=1 TO 5
30 READ ENG$(M),FR$(M)
40 NEXT M
50 R=1+INT(RND(1)*5)
60 PRINT "?";ENG$(R)
70 PRINT "TYPE IT IN FRENCH"
80 INPUT T$
90 IF T$=FR$(R) THEN GOTO 120
100 PRINT "SORRY, ANSWER IS ";FR$(R)
110 GOTO 130
120 PRINT "*****CORRECT*****"
130 FOR D=1 TO 1000
140 NEXT D
150 GOTO 50
160 DATA DOOR,PORTE,CHAIR,CHAISE
170 DATA NOSE,NEZ,FOOT,PIED,APPLE,POMME
```

List your program.

Line **10** tells the computer to put aside space for up to five subscripted variables with the main name **ENG$** and for up to five others with the main name **FR$**.

Lines **20** through **40** are a loop that the computer goes through five times, the first time setting **M** (which is going to be the subscript of **FR$** and **ENG$**) equal to **1**, the second time equal to **2**, and so on. Line **30** tells the computer to read from **DATA** lines to get the values for the string variables **ENG$(M)** and **FR$(M)**. Thus, the first time, it sets **ENG$(1)** equal to **DOOR** and **FR$(1)** equal to **PORTE**; the second time it sets **ENG$(2)** equal to **CHAIR** and **FR$(2)** equal to **CHAISE**; and so on.

So far, this program is very much like the last version of the coin collection program in the last chapter. You have used a loop to read the information into subscripted variables from **DATA** program lines. This part of the program takes in the database.

Line **50** selects a random number between **1** and **5** and sets the variable **R** (we could have used any variable name) equal to that random number.

To review this again, here is how the line works, going from the inside out:

1. **RND(1)** produces a random number between **0** and **1**.
2. It is multiplied by **5** (**RND(1)*5**) to yield a number between **0** and **5**.
3. **INT** makes the random number a whole number, so you have **INT(RND(1)*5)**.
4. Now, add **1**. Thus, instead of a whole number from **0** to **4**, you end up with a whole number from **1** to **5**.
5. Finally, in order to use it later on, make some letter—we'll be creative and use **R**—equal to your random number.

```
10 DIM ENG$(5),FR$(5)
20 FOR M=1 TO 5
30 READ ENG$(M),FR$(M)
40 NEXT M
50 R=1+INT(RND(1)*5)
60 PRINT "□";ENG$(R)
70 PRINT "TYPE IT IN FRENCH"
80 INPUT T$
90 IF T$=FR$(R) THEN GOTO 120
100 PRINT "SORRY, ANSWER IS ";FR$(R)
110 GOTO 130
120 PRINT "*****CORRECT*****"
130 FOR D=1 TO 1000
140 NEXT D
150 GOTO 50
160 DATA DOOR,PORTE,CHAIR,CHAISE
170 DATA NOSE,NEZ,FOOT,PIED,APPLE,POMME
```

As you will see in a minute, since the program involves a **GOTO** loop, you will come to line **50** several times (as often as you want). Since the computer picks a new random number each time, the value of **R** may be different each time—sometimes **1**, sometimes **2**, sometimes **3**, sometimes **4**, and sometimes **5**.

Line **60** clears the screen and then displays a randomly selected English word (from the five we put in) by taking the word equal to **ENG$(R)**. Very clever!

Line **70** is a prompt that asks you to type what you think is the correct French translation. Line **80** sets **T$** equal to your guess.

Line **90** checks whether you are right. It asks the computer if your guess (**T$**) is equal to the right answer. The right answer is the French word that is equal to the variable **FR$(R)**, since the equivalent English and French words have the same subscript.

If your guess does equal the right answer, then the computer will go to line **120**, where it prints *****CORRECT*****.

If your guess is *not* correct—that is, if it is not true that **T$ = FR$(R)**—the computer ignores the **THEN** instruction. It goes right on to the next line, line **100**, where it kindly tells you **SORRY, ANSWER IS** and then what **FR$(R)** is equal to.

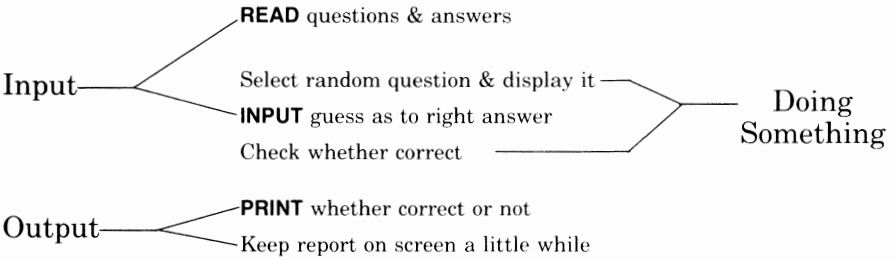
If you had the wrong answer, you would then come to line **110**, which would skip you down to line **130**. If you had the right answer, you would have gone to line **120**, and then automatically have gone right on to line **130**. Either way, right or wrong, the computer eventually comes to line **130**.

Lines **130** and **140** are a delay loop. Each time the computer comes to these lines, it has to go through this loop 1000 times before it can proceed. In the present case, you are delaying in order to leave on the screen the ******CORRECT****** (to make you feel good) or the message that you were wrong (so you can look at the right answer). Either way, in a little while the computer comes to line **150**, goes back to line **50**, selects a new **R**, goes to line **60**, clears the screen, and shows you a new English word with the prompt from line **70**. The computer goes through all this as many times as you are willing to respond at this point.

This program is typical of programs that help you memorize things (such as states and their capitals, lines of a script, or any set of questions and answers—as well as a foreign-language vocabulary). First it reads the information (in this case, the pairs of English and French words). Second, it selects a random number (in this case between **1** and **5**), which it uses to pick a question (in this case a random choice from a list of five English words). Third, you are asked to give the correct answer (the French equivalent). Fourth, the computer compares your answer to the right one. Fifth, if you are correct, it tells you so; if not, it tells you the right answer. Sixth, there is a delay loop that permits you to appreciate the result of your guess. Finally, the computer goes back to the beginning and gives you a new question.

In terms of the fundamental three-part program structure we have emphasized in this book, programs of this kind can be understood as follows. The program *takes in the information* in two ways (very much like database management programs): through the flashcards (in this case, the English and French word pairs) obtained with the **READ**-and-**DATA** procedure, and through your typed answer. The *doing-something-with-the-information* aspect involves selecting a random question (in this case, the English word to be translated) and evaluating whether your guess is correct. The *output* consists of showing you on the screen whether you are right or wrong—and doing so for a considerable length of time!

The program could be outlined as follows:



Run this program, and see how long it takes you to learn the five French words. If you already know them, make mistakes on purpose to see how the program handles wrong answers!

When you are finished, stop the program with [RUN/STOP] and [RESTORE].

Now, *readez vous la summary.*

In the next chapter, we will conclude Part II of this book by considering the few remaining topics you need to know to be a competent programming admiral of your Commodore.

You've absorbed a lot, although you may be wondering if you've really been absorbing, or just drowning. That's all right—like any skill, learning to use a computer comes in both steep climbs and plateaus. Think of how much you've already learned! The rest will be yours with practice, as you apply it.

Summary

This chapter introduces string variables and some additional programming tools.

A string variable has a dollar sign at the end of the variable name and stands for a series (or string) of characters. (For example, the program line **10 A$="COMMO"** sets the variable **A$** equal to the character string **COMMO**.)

Two strings can be made into a single string using the plus sign. (For example, if **A$="COMMO"** and **B$="DORE 64"**, then **A$+B$** equals **COMMODORE 64**.)

You can slice out part of a string with **MID$**. For example, if **A$="ABCDEFGHIKJK"**, then **MID$(A$,3,2)** would tell the computer to slice out the part beginning with the third character and continuing for a total of two characters. Thus, **MID$(A$,3,2)** would yield **CD**. The second and third terms in parentheses can also be variables or arithmetic expressions—for example, **MID$(A$,3*N,2)**.

Subscripted string variables are just like subscripted numeric variables, except that instead of standing for numbers, they stand for character strings. With subscripted string variables, you can file and sort words and symbols, as well as numbers, in database management. They will also be helpful in doing advanced graphics. In this chapter, we apply them in a program that helps you memorize a foreign-language vocabulary.

RND(1) tells the computer to pick a random number between **0** and **1**. To change the range, multiply the random number by the highest number you want in the range. (For example, **RND(1)*5** yields a random number between **0** and **5**.) To get whole-number random numbers, use **INT**, which rounds numbers down to whole numbers. (For example, **INT(2.87)** produces **2**; and **INT(RND(1)*5)** yields a random number that is either **0**, **1**, **2**, **3**, or **4**.) Finally, you can add a number to move the whole range up. (For example, **1+RND(1)*5** would, finally, produce whole random numbers from **1** to **5**.)

A delay loop delays the computer while it is carrying out a program. It consists of a **FOR . . . TO . . .** and **NEXT** loop with no lines in between (for example, the loop **130 FOR D=1 TO 1000** and **140 NEXT D**).

Programs that give you answers to questions (such as the French translation of English words) are similar to flashcards. The computer shows you the question, you give the answer, and the computer tells you if you are right or not. Then the cards are shuffled and you try a new one.

A program of this kind can be analyzed in terms of the standard three parts of any program. The *input* includes the question-and-answer pairs that are part of the program, along with your guesses, which you type at the appropriate times. *Doing something with the information* includes the computer's selection of a random question, and its evaluation of your guess as right or wrong. Finally, *output* includes the computer's report as to whether you were right or wrong and an adequate length of display time so that you can think about the result.

Terms and Concepts Introduced in Chapter 10

String variable

Combining strings (concatenation)

Substrings and string slicing

MID$

Subscripted string variable

RND(1)

INT

Delay loop

Flashcard program

Practice Exercises

(Answers and comments in Appendix A.)

1. Write a program in which, when you give it the number of a month of the year, it gives you the two-letter abbreviation of that month. For example, if you input **1**, it prints **JA**. If you input **2**, it prints out **FE**, and so forth. (Use **JN** for June, **JL** for July, **MA** for March, and **MY** for May.) Use the **PLACE FINISHED** program as a guide.

2. Write programs to produce 10 random whole numbers as follows:

a. from **1** to **5**

b. from **1** to **8**

c. from **Ø** to **2**

d. from **Ø** to **25**

e. from **1** to **4**

3. Make a flashcard program that helps you memorize which sound frequency goes with which note of one octave of the C major musical scale.

The pairs are as follows:

NOTE	FREQUENCY
C	262
D	294
E	33Ø
F	349
G	392
A	44Ø
B	494
C	523

PART II CHAPTER 11

A Few More Fine Points of BASIC Programming

In the last chapter, you learned about string variables, subscripted string variables, flashcard programs, random numbers, and delay loops.

There's still more to learn—in fact, far more than we could possibly cover in a single book. You do not, however, have to know all the professional's fancy details and advanced procedures to do most of the kinds of programs you will probably want to do. Indeed, with what you have already learned, there is little you could not do. Perhaps your program won't be as elegant as the professional's, but it will do the job.

Before we leave our introduction to programming, there are three or four more fine points we should cover. If you go on to Part III (on graphics and sound), you will learn some procedures that are useful in general programming as well. Part IV discusses the steps involved in actually writing original, longer programs. In *this* chapter, you will complete your beginning course by studying subroutines, character codes, and various procedures for shortening programs. You'll also work on an example of a game program.

Programs within Programs—Subroutines, GOSUB, and RETURN

A *subroutine* is a miniprogram or subprogram within a larger program. It is a set of numbered lines you want done several times within the main program.

Clear the computer's memory (with NEW) and type the following:

```
10 PRINT "TYPE IN 2 NUMBERS"
20 INPUT X,Y
30 GOSUB 200
40 X=X+1
50 GOSUB 200
60 Y=Y+1
70 GOSUB 200
```

```
10 PRINT "TYPE IN 2 NUMBERS"
20 INPUT X,Y
30 GOSUB 200
40 X=X+1
50 GOSUB 200
60 Y=Y+1
70 GOSUB 200
80 END
200 J=X+Y
210 IF J>7 THEN J=INT(X*Y/.17)
220 PRINT "J =";J
230 RETURN
```

List your program.

In a minute, this program is going to seem quite straightforward (if it doesn't already). Don't worry about what all the arithmetic means—you can completely ignore lines **200**, **210**, and **220**. This is just an example to show you how subroutines work.

Lines **10** and **20** are a prompt-and-**INPUT** sequence that asks you to type two numbers and then sets **X** and **Y** equal to what you type. Line **30** tells the computer to go to a subroutine that starts at line **200**. The computer then does everything called for in lines **200**, **210**, and **220**. Then it comes to line **230**. **RETURN** is a special instruction that always goes at the end of a subroutine. Whenever you use **GOSUB** to go to certain program lines, those program lines must end with a **RETURN** line. Otherwise, you'll get an error message. (Incidentally, don't confuse a **RETURN** program line, which is typed letter by letter in the usual way, with the [RETURN] key.)

When the computer comes to the **RETURN** line, it does what it is told to do—it returns to the main program, *continuing where it left off*. In this program, since this time it left from line **30**, it goes back to the program line that comes right after line **30**—which is line **40**.

Line **40** sets **X** equal to the old value of **X** (the first number you typed in response to the prompt)—plus **1**. If you had typed **15**, **X** would now be set equal to **16**.

At line **50**, the computer is sent again to the subroutine, where it goes through all the steps again, this time using the new value of **X**. At line **230**, it again returns to where it came from—this time going back to the line immediately after line **50**.

At line **60**, the variable **Y** is set equal to what you typed for it, plus **1**. Then, at line **70**, it's off to the subroutine again, and when it's finished, it goes back to line **80**.

Line **80** says **END**. That tells the computer that the program is finished.

Usually, you don't need to put an **END** line at the end of a program—although it never hurts. In this case, however, it is necessary. Otherwise, the computer would go right on from line **70** to the next line in order—which is line **200**, the start of the subroutine. In the first place, we don't want to use the subroutine again. More important, if the computer did go into the subroutine program lines, it would eventually come to the **RETURN** line. This would make no sense to the computer, since it had not come from a **GOSUB** line. You would get an error message.

Run the program. Use any two numbers you like for the input.

Try running the program a few more times with different input.

You probably noticed that **GOSUB** is actually much the same as **GOTO**. The difference is that the program lines in a subroutine always end with **RETURN**. After carrying out the instructions in line **200** and subsequent lines, the computer comes to the line that says **RETURN**, and it goes back to where it just came from before the subroutine.

You can see the special advantage of **GOSUB** in the present example. Suppose that, instead of a **GOSUB** instruction, line **30** tells the computer to **GOTO 200**. That would work fine. The computer would go to line **200** exactly as if the line had used **GOSUB**. At line **230**, instead of **RETURN**, you would need another **GOTO** line to send the computer back to where it came from. Line **230** would become **GOTO 40**. This would produce exactly the same result as the **GOSUB** procedure.

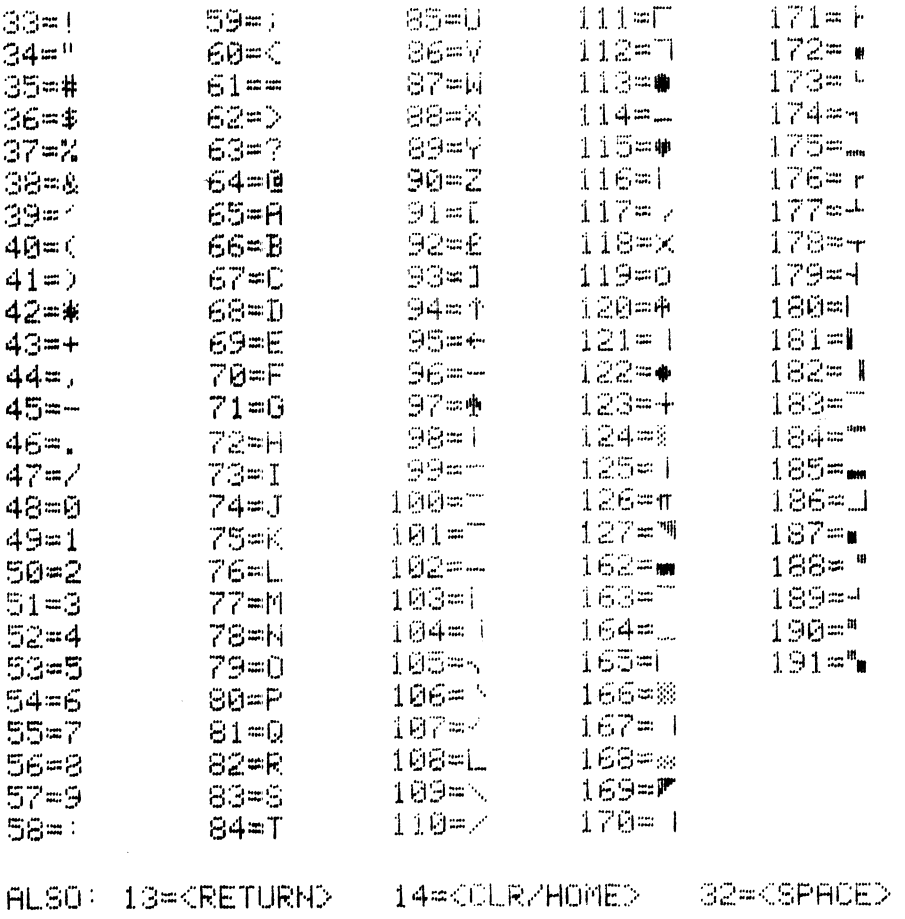
But what would happen the next time you used this subroutine? Later in the program, at line **50**, the computer is sent to the same subroutine. Using **GOTO 200** again would work fine. The problem would be getting back! Line **230** would still be **GOTO 40**. Instead of going back to line **60**, the computer would end up at line **40** again. In fact, the computer could never get any further in the program!

The subroutine procedure allows you to use a set of program lines several times in the course of a program, each time sending you back to the point where you were in the program before the subroutine. As you progress in programming, you will find the **GOSUB**-and-**RETURN** procedure to be one of the handiest little tools in BASIC.

Numbers That Stand for Letters—Character Codes, CHR$, and ASC

The Commodore 64 has a code number for every letter (and every other character the keyboard can produce). A list of those codes is shown in the following table.

Character Code Used with CHR$ and ASC



CHR$** converts a code into the letter it stands for. Thus, **CHR$(73) is the letter **I**. Try it.

Type PRINT CHR$(73) and press [RETURN].

For each of the following, type and then press [RETURN]:

```
PRINT CHR$(94)

PRINT CHR$(34)

PRINT CHR$(28+10)
```

You can also use variables with **CHR$**.

Clear the computer's memory and type the following:

```
10 FOR Q=65 TO 90
20 PRINT CHR$(Q);
30 NEXT Q
```

List the program.

Notice the semicolon at the end of line **20**. It makes each subsequent printing of what **CHR$** equals appear right next to the one before.

Run the program.

One of many practical uses for **CHR$** is that it permits you to print things you can't put within quotes. For example, you cannot put a quote mark within quotes because the computer would think you were closing quotes! But you *can* tell the computer to print a quote mark by using **PRINT CHR$(34)**.

Type PRINT "HE SAID, ";CHR$(34);"HI!";CHR$(34)

Press [RETURN].

This should come out like this:

```
HE SAID, "HI!"
```

Now let's flip the coin and get to know **ASC**. **ASC** does the opposite of **CHR$**. It converts a character to its code number.

For each of the following, type and then press [RETURN]:

```
10 PRINT ASC("/")
20 PRINT ASC("T")
40 PRINT ASC(" ")
```

This last example should give you **32**—the code for a blank space!

You can also use **ASC** with variables.

Clear the computer's memory and type the following:

```
10 A$=">"
20 PRINT ASC(A$)
```

List and run your program.

This should give you **62**, since that's the code for >.

The character codes—with **CHR$** and **ASC**—have many programming applications (believe it or not). They are useful for prompts in which the person inputs words. They are also necessary for many specialized programs you will encounter in other books, if you go further with programming. And, as you might expect, they come up in various graphics applications.

An Example—An Intuition-Test Program

The program below illustrates one use for subroutines and character codes. It also introduces a programming pattern that is often used in computer games.

Clear the computer's memory and type the following:

```
10 REM INTUITION TEST
20 N=1+INT(RND(1)*4)
30 PRINT "GUESS A NUMBER 1 TO 4"
40 INPUT GN
50 IF GN<>N THEN PRINT"SORRY, IT WAS";N
60 IF GN=N THEN GOSUB 1000
70 CC=65+INT(RND(1)*26)
80 L$=CHR$(CC)
90 PRINT "GUESS A LETTER"
100 INPUT G$
110 IF L$<>G$ THEN PRINT"SORRY,ITS ";L$
120 IF L$=G$ THEN GOSUB 1000
130 GOTO 20
1000 PRINT "**********"
1010 PRINT "******AMAZING******"
1020 PRINT "**********"
1030 RETURN
```

List your program.

Line **10** is simply a remark that, as you will recall from Chapter 5, the computer ignores when it runs the program. Line **20** selects a random whole number from **1** to **4**. (To review *once* more, **RND(1)** picks a random number between **0** and **1**. Multiplying it by **4** makes it a random number between **0** and **4**. The **INT** function knocks off the decimal places, making it a whole number from **0** to **3**. Adding **1** makes it a random number that is either **1**, **2**, **3**, or **4**.) Then **N** is set equal to that random number.

Line **30** is a prompt that asks you to guess a number from **1** to **4** and type it. Line **40** sets **GN** equal to your guess.

Lines **50** and **60** compare your guess to the right answer. If you are wrong—that is, if your guess, **GN** (for "**GUESS A NUMBER**"), does not equal the random number **N**, then the comparison **GN<>N** is true (<> is the symbol for "not equal to"). Thus, the computer prints that it is **SORRY** and tells you the right answer, which is the value of **N**.

If you are correct—that is, if at line **60** it is true that your guess, **GN**, equals the random number **N**—then the computer is told to **GOSUB 1000**.

The subroutine prints a fancy display, and then at line **1030** it returns you to the line just following the one that sent you to the subroutine—that is, it goes to line **70**. If you had been wrong, the **IF** equation in line **60** would have been false (that is, it would not be true that **GN=N**). Thus, you would have gone directly on to line **70** without doing the subroutine. (Remember, line **50** already told you on the screen that you were wrong.)

Either way, you come to line **70**. This line sets **CC** equal to a random whole number between **65** and **90**. (You should think through why it comes out that way.)

The numbers from **65** to **90** just happen to be the character codes for **A** through **Z**! Thus, line **80** sets the string variable **L$** equal to the letter for which **CC** is the code. Hence, you get a random letter.

Lines **90** through **120** duplicate lines **30** through **60**, except that you are guessing a letter instead of a number.

At line **130**, the computer is told to **GOTO 20** to start all over again.

Notice, incidentally, that although there is a subroutine at the end of this program, we did not need to include an **END** program line. Why? There is no way the computer can get to the subroutine except through one of the **GOSUB** instructions. That's because, no matter what, the computer always ends up at line **130**, just before line **1000**, and line **130** always sends the computer back to line **20** to start over again. The program never really ends at all. Eventually, you will have to stop it with [RUN/STOP] and [RESTORE].

Run the program and keep making guesses until you get at least a couple of them right.

Some BASIC Shortcuts

There are several ways to make a BASIC program more compact. These shortcuts help the experienced programmer save a little time typing a program. They also save some space in memory.

In general, especially when you are learning, it is better to write out programs as clearly as possible. But as you get better at programming, you will appreciate some of these shortcuts. You should at least be aware of them, since you will run into these procedures if you look at Commodore 64 programs in magazines or in other books. Consider this a "for-future-reference" section.

Let's type a little program first in the ordinary way, and then see some of the ways it could be abbreviated without changing what it does.

Clear the computer's memory with the NEW procedure, and type out the following program. (Leave 9 spaces between the reverse-heart figure and CHARACTERS in line 10.)

```
10 PRINT "♥         CHARACTERS & CODES"
20 FOR C=35 TO 118
30 PRINT CHR$(C);" =";C,
40 NEXT C
```

List your program.

Line **10** clears the screen and then prints the title, centered, on the first screen line. Line **20** begins a **FOR...TO...** and **NEXT** loop that, the first time, sets **C** equal to **35** and proceeds to line

40, where it is sent back to line **20**. When the computer comes to line **20** the second time, it sets **C** equal to **36**. This process continues until **C** has gone all the way up to **118**.

The key line is line **30**. It prints the actual character for which **C** is the code, then an equals sign, then the number that **C** stands for. The first time, for example, since **C** equals **35**, the screen shows **# = 35**. (You will recall from Chapter 8 that, because the **PRINT** instruction ends with a comma, the next time it is carried out, printing begins in the next print zone on the same line—that is, the 10th space across the screen.)

Run the program.

Now, for the shortcuts.

One way to shorten this program is to use abbreviations for the various BASIC terms. The following table gives the abbreviations for the BASIC vocabulary you have learned so far.

Abbreviations of BASIC Terms

CHR$=C	GOTO=G	NEW=NONE	RND=R
CONT=C	IF=NONE	NEXT=N	RUN=R
DATA=D	INPUT=NONE	POKE=P	SAVE=S
DIM=D	INT=NONE	PRINT=?	SPC(=S
END=E	LET=L	READ=R	STEP=ST
FOR=F	LIST=L	REM=NONE	TAB(=T
GET=G	LOAD=L	RESTORE=RE	THEN=T
GOSUB=GO	MID$=M	RETURN=RE	VERIFY=V

You can see that the abbreviation for **PRINT** is the question mark (**?**), and that some terms don't have any abbreviations at all. Otherwise, the abbreviation is usually the first letter and then the graphic symbol you get when you hold [SHIFT] and press the key for the second letter. Isn't that clever? To get the abbreviation for **GOTO**, for example, type **G** and then hold [SHIFT] and press the [O] key. This produces **G/** on the screen. If you substitute abbreviations for all the terms in the program you just ran, it would look like this':

```
10 ? "          CHARACTERS & CODES"
20 F C=35 TO 118
30 ? C CHR$(C);" =";C,
40 N C
```

Edit or retype your program to look as shown:

Do *not* list your program yet! (If you list your program, the abbreviations will be converted to full words. In a moment you can try that, but not quite yet.)

As you can see, the program is a bit hard to follow by looking at it. But it is certainly shorter, and it *is* the same program.

Run the program.

Now that you have seen the program run in this form, list it.

See? The abbreviations are gone.

Another way to shorten your program is to combine lines where possible. Instead of numbering each line, put one line after the other by simply putting a *colon* (:) between two instructions and by omitting the line numbers. If, however, you have a **GOTO** or **GOSUB** in your program—which always refer to a destination line by number—you had better leave that destination line as a separate numbered line!

Clear the computer's memory (with NEW) and retype your program so that it comes out as follows:

```
10 ? "□           CHARACTERS & CODES":FT
C=35 TO 118:? C;CHR$(C);" =";C,:N  C
```

This is still the same program!

Run your program.

In general, a colon between two lines works just like consecutively numbered lines. One exception is that following an **IF . . . THEN . . .** instruction, the computer goes to the line after the colon only when the **IF** comparison is true. Otherwise, the computer goes to the next *numbered* line.

There are still a few more things you can do to shorten a program. For one thing, in most cases, the Commodore 64 does not require spaces between things in a program line. Also, you can eliminate all the spaces at the beginning of the first instruction (which are there to center the title) by using **SPC**. Notice, incidentally, that the abbreviation for **SPC** includes the left parenthesis, so that **SPC(10)** is abbreviated **S 10**).

Finally, the Commodore 64 does not require the variable name in a **NEXT** instruction. Instead of **NEXT C**, you could just say **NEXT**, and the computer knows it must be **NEXT C**.

Retype your program to look like this:

```
10?"□";S 10);"CHARACTERS & CODES":FTC=35
TO118:?C;CHR$(C);" =";C,:N
```

Although it hardly looks like the original, it is actually the same.

You can see why we recommend sticking to writing programs in the unabbreviated form! But it's good to know about all these abbreviations because when you look at Commodore 64 programs—even in the *User's Guide*—you'll see many if not all of these shortcuts.

Run the last version of the program.

In the next few chapters (Part III), you will enter the exciting world of Commodore 64 graphics and sound. If you are really not interested in that, go right on to Part IV. But you'll be missing a lot of fun!

Summary

In this chapter, you learn some additional fine points of BASIC programming.

A subroutine is a kind of miniprogram that is used one or more times in carrying out the main program. It consists of two or more program lines, usually put near the end of the program. **GOSUB** sends the computer to the subroutine. It is similar to **GOTO**, except that the last line of a subroutine always says **RETURN**, which tells the computer to go back to the line immediately following the one that sent it to the subroutine. An **END** line (for example, **80 END**) at the end of the main part of a program prevents the computer from inappropriately going on to the (usually higher numbered) subroutine lines.

Every character has a special number code. For example, the code for the letter **A** is **65** and the code for the graphic symbol ⊥ is **167**. A list of all the codes is given in the chapter for your convenience. **CHR$** converts a code number to the character it stands for. For example, the **CHR$(65)** command would produce **A**. **ASC** converts a character to its code number. For example, the **ASC("A")** command would produce **65**.

The use of subroutines and character codes is illustrated in an "Intuition-Test" program. In this program, a random number is selected internally by the computer. You type a guess as to what it is, and then the computer compares your guess to the number it has selected. If you're correct, the computer is sent to a four-line subroutine that prints a fancy **AMAZING**. It then goes back to the rest of the main program, which is just like the first part except that now you guess a random letter. The letter is selected internally by the computer by first generating a random number between **65** and **90** and then using **CHR$** to convert the number to a letter.

Programs can be abbreviated in several ways. Once you know them well, these shortcuts make typing programs a bit faster. Abbreviations also save space in memory, if that's a concern. On the other hand, abbreviated programs are much harder to understand when you look at them. Thus, we don't advise beginners to use many of these procedures. But you should know about them, since many programs in books and magazines use them.

One way to shorten a program is to use abbreviations for the BASIC terms. The abbreviation for **PRINT** is the question mark (**?**). For most other terms, it is the first letter of the term and then the graphic symbol you get when you press the key for the second letter of the term while holding down the [SHIFT] key. Thus, for example, the abbreviation for **GOTO** is **G** .

Another shortcut is to combine lines by putting one after the other, without numbers, separated by a colon. (Of course, you can't combine lines when you need the numbers, as with destinations for **GOTO** commands.) Thus, **10 INPUT A$**, **20 PRINT A$**, and **30 GOTO 10** combines to become **10 INPUT A$:PRINT A$:GOTO 10**. The short version works exactly the same as the long one. (An exception is that following an **IF . . . THEN . . .** instruction, the computer goes to the instruction after the colon only when the **IF** comparison is true. Otherwise, the computer goes to the next *numbered* line.)

You can also shorten a program by eliminating spaces between things in a program line, using **SPC** instead of typing out the spaces, and by leaving the variable name off a **NEXT** instruction.

Terms and Concepts Introduced in Chapter 11

Subroutine

GOSUB

RETURN

Character code

CHR$

ASC

Abbreviated BASIC instructions

Combining program lines

Other procedures for shortening programs

Practice Exercises

(Answers and comments in Appendix A.)

1. Write a program, based on the Intuition Test, that has you guess the numbers **1** or **2**, then a number from **1** to **5**, then a number from **1** to **10**, then back to the numbers **1** or **2**. When you are correct, the program should send the computer to the following subroutine:

```
1000 FOR X= 1 TO 2000
1010 PRINT "□"
1020 PRINT "□"
1030 PRINT "CORRECT"
1040 NEXT X
1050 RETURN
```

If your answer is wrong, there should be a subroutine that says you are wrong, gives the correct answer, and keeps it on the screen a little while before going on to the next item to guess.

2. Write a program that prints the character code when you type a letter.

3. Shorten as much as possible the three-line version of the feet-to-meters conversion program in Chapter 7.

PART III
Show Business: Graphics and Sound

PART III CHAPTER 12

POKEing Your Computer with Color and Design

This chapter begins our discussion of how to make sophisticated graphics with the Commodore 64. You will first learn the important **POKE** command. Then you'll learn how to apply the command to change the color of the screen and border and make designs.

POKE and Changing Screen and Border Colors

POKE directly changes the internal functioning of the computer. Everything the computer does is controlled by some piece of electronics located at a particular place in the computer's innards. Each place has its own identifying number or *address*.

If you know the address, and what code number to "poke" into that address, you can make the computer do all kinds of wonderful things that are not otherwise available—especially with graphics and sound.

For example, the address for the place in the computer that controls the color of the screen is **53281**.

Please, don't waste your time trying to memorize this or any other **POKE** address. There is no obvious logic to the address numbers, and there are a fair number of them. When you want one, look it up. There's a list of the important **POKE** addresses in Appendix B.

To change the screen color, you also have to know the code number to poke into address **53281**. There are 16 possible screen colors. Their codes are shown in the following table.

Codes for Screen and Border Colors

CODE	COLOR	CODE	COLOR
0	BLACK	8	ORANGE
1	WHITE	9	BROWN
2	RED	10	DARK ORANGE
3	"CYAN" (slightly greenish light-blue)	11	DARK GRAY
		12	MID-GRAY
4	PURPLE	13	MINT GREEN
5	GREEN	14	LIGHT BLUE (the normal character color)
6	BLUE (the normal screen color)		
7	YELLOW	15	LIGHT GRAY

Again, there is little point in memorizing these. You can always look them up. If you do a lot of graphics, you'll memorize them without trying!

From now on, we will sometimes refer to the location in the computer's innards as a "**POKE** address," or just "address," and to the number you poke into that address as a "code number," or just "code."

Now let's poke. To make the screen white, type **POKE**, the address that controls screen color (**53281**), a comma, and the code for white (**1**).

Type POKE 53281,1 and press [RETURN].

The screen should turn white, but the letters and border should remain the same color as before.

To get the normal blue screen color back, either poke in the code for blue or use the [RUN/STOP] and [RESTORE] keys. Clearing the screen won't do it.

Hold down [RUN/STOP] and press [RESTORE]. (Remember, you sometimes have to tap the [RESTORE] key rather smartly while holding down the [RUN/STOP] key to make it work.)

These two keys set all the **POKE** addresses back to their normal settings. The screen color should be back to normal. As with programs, the [RUN/STOP]-and-[RESTORE] combination is your passport out of whatever difficulties you have gotten into. It clears the screen, unpokes everything, and stops running any program. Fortunately, however, it doesn't erase the program in memory.

Now, back to screen colors.

Type each of the following, pressing [RETURN] after each:

```
POKE 53281,0

POKE 53281,10

POKE 53281,14
```

The last line made the screen the same color as the letters, so the letters seem to disappear!

Hold down [RUN/STOP] and press [RESTORE].

You can change the color of the screen border in the same way. Its **POKE** address is **53280**.

Type POKE 53280,7 and press [RETURN].

This should give you a yellow border.

Type POKE 53280,6 and press [RETURN].

This should make the border the same color as the rest of the screen—a rather nice effect!

The next step is to put **POKE** commands in a program. Here is an example that displays all the different screen colors:

Return the screen to normal with [RUN/STOP] and [RESTORE], and clear the computer's memory (with NEW).

Type the following lines:

```
10 FOR C=0 TO 15
30 POKE 53281,C
50 FOR D=1 TO 200
60 NEXT D
80 NEXT C
```

Makes **C** a variable—for different screen color codes

Pokes different screen colors

Delay loop

List your program.

Line **10** starts a loop that begins with **C** equal to **0** and continues until **C** equals **15**. Line **30** pokes whatever **C** equals into the screen-color-control address (**53281**). The first time, it pokes in **0**, making the screen black; the next time, it pokes in **1**, making the screen white; and so on, through all the colors.

Lines **50** and **60** are just a delay loop to slow the whole thing down. Otherwise, the screen colors would change too fast.

Run your program.

Notice that the screen ends up light gray—the last color code (**15**) is for this color. Leave it that way for now.

Now, do the same with the border colors.

Edit line 30 to read 30 POKE 53280,C

List and then run your program.

Notice this time that the border is gray when the program is done.

Hold down [RUN/STOP] and press [RESTORE] to get back to normal.

If you put another loop inside the main loop you now have, you can get all the combinations of screen and border colors.

Type the following lines:

```
20 FOR S=1 TO 15
40 POKE 53281,S
70 NEXT S
```

List your program.

```
10 FOR C=0 TO 15
20 FOR S=1 TO 15
30 POKE 53280,C
40 POKE 53281,S
50 FOR D=1 TO 200
60 NEXT D
70 NEXT S
80 NEXT C
```

This time the computer has to go through the inside loop (with the **S**) 15 times before it can go through the outside loop (with the **C**) once. Thus, you'll see the first border color with all the screen colors, then the next border color with all the screen colors, and so forth.

Run your program.

Poking Characters onto the Screen

Think of the screen as a kind of checkerboard with 40 boxes across and 25 boxes down—a total of 1000 boxes in all. The Commodore 64 has a **POKE** address to control what character gets displayed in every one of those boxes. Then it has a second **POKE** address for every one of those thousand boxes to control the *color* in which that character gets displayed. You can use these **POKE** addresses to make any character appear in any color any place you want on the screen.

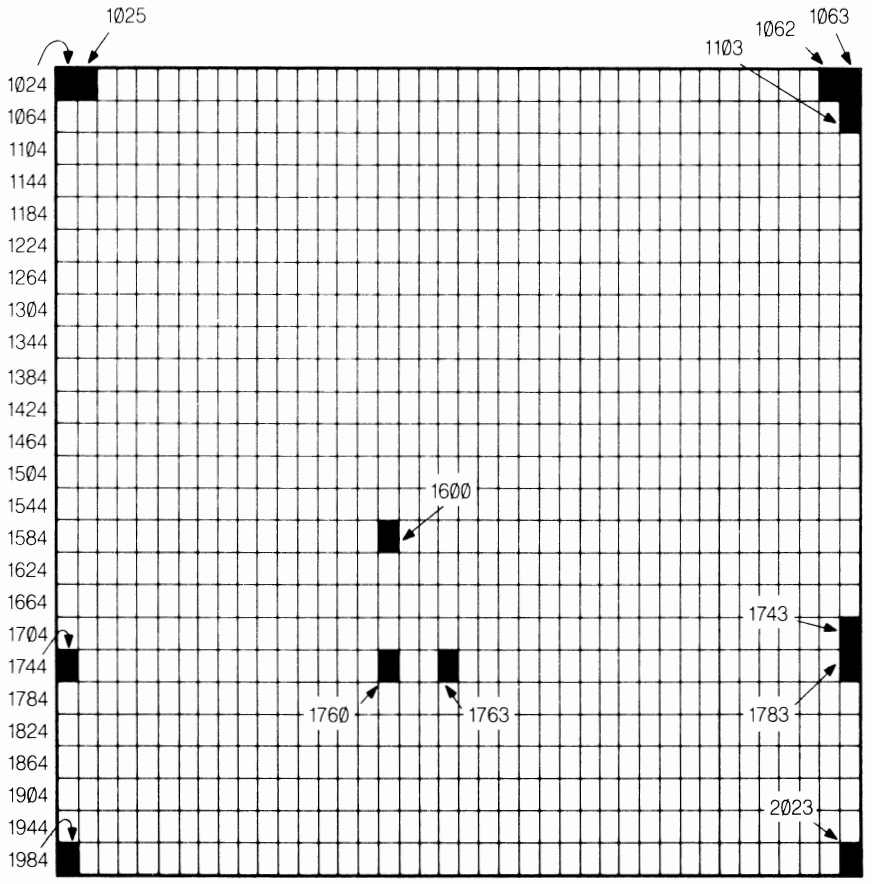
Of course, the simplest way to make something appear on the screen is just to type it directly from the keyboard. But when you just type a character, the computer doesn't remember it for future use. Another possibility is to use **PRINT** in a program. We saw in Chapter 8 that you can make programs for some fairly interesting designs using **PRINT**. But the procedure was cumbersome, and you couldn't use color. Furthermore, the method in Chapter 8 will not work for animation or keyboard-controlled movement of screen designs (which we'll get to in the next chapter).

The **POKE** approach to graphics also seems unwieldy at first, but there are ways—and you'll learn some of them—to make it pretty easy. Besides, if you want to do any but the simplest graphs or designs, you have little choice!

The idea is that you can poke just what you want, just where you want.

The **POKE** numbers for the various screen locations are shown in the following figure.

POKE ADDRESSES FOR PUTTING CHARACTERS ON THE SCREEN



Notice that the **POKE** numbers begin with **1024** in the top left corner and continue across the screen to **1063** at the top right corner. The next screen line down begins with **1064** and continues to **1103**. It continues like this until you get to the end of the last screen line, which is **2023**. (You cannot poke characters into the border area—you can only change the color of the border.)

The **POKE** addresses for the *colors* of characters for each box are each exactly **54272** higher than the corresponding address for characters for that box. Thus, for example, if the screen location address for characters for a particular box is **20000**, then the color address for that box would be **56272**—that is, **2000** plus 54272.

Before you begin poking these screen-control addresses, you need to know what code numbers to poke into them. The code for characters is called the *screen display code*. Every character has

its own number (see the table displaying the screen display codes). For example, the code number for **N** is **14**. The code for **I** is **66**.

Screen Display Code

```
 0=@   20=T   40=(   60=<   80=┐  100=_   120=▒
 1=A   21=U   41=)   61=═   81=●  101=▌   121=▄
 2=B   22=V   42=*   62=>   82=▁  102=▒   122=┘
 3=C   23=W   43=+   63=?   83=♥  103=▕   123=▖
 4=D   24=X   44=,   64=─   84=▌  104=▒   124=▝
 5=E   25=Y   45=-   65=♠   85=╱  105=◤   125=┘
 6=F   26=Z   46=.   66=▏   86=╳  106=▕   126=▔
 7=G   27=[   47=/   67=╴   87=○  107=├   127=▚
 8=H   28=£   48=0   68=▔   88=♣  108=▗
 9=I   29=]   49=1   69=▔   89=▕  109=└
10=J   30=↑   50=2   70=▁   90=♦  110=┐
11=K   31=←   51=3   71=▏   91=┼  111=▁
12=L   32=    52=4   72=▕   92=▒  112=┌
13=M   33=!   53=5   73=╲   93=▕  113=┴
14=N   34="   54=6   74=╲   94=π  114=┬
15=O   35=#   55=7   75=╱   95=┤  115=┤
16=P   36=$   56=8   76=└   96=   116=▏
17=Q   37=%   57=9   77=╲   97=▌  117=▌
18=R   38=&   58=:   78=╱   98=▄  118=▐
19=S   39='   59=;   79=┌   99=▔  119=▔
FOR REVERSE OF ANY CHARACTER, ADD 128
--MOST USEFUL IS REVERSE SPACE:160=
```

Notice that the screen display code is *not* the same as the character code you learned about in Chapter 11. The idea is the same, and even *some* of the code numbers are the same, but most are not. We don't know why the Commodore 64 doesn't use just one code. But, it's not that big a deal. Just look up whatever kind of code you want as you need it. These are not numbers you would be likely to memorize, anyway.

The code for color, fortunately, is very simple—it uses the same **0** to **15** code numbers you used for screen and border color.

Now, let's poke around a little on the screen.

The bottom left corner of the screen is controlled by **POKE** address **1984**, and the code for the club symbol (as on a playing card) is **88**. The color address for that box is just **1984** *plus* **54272**.

Press [RUN/STOP] and [RESTORE], then clear the screen (using [SHIFT] and [CLR/HOME]).

Type POKE 1984,88 and press [RETURN].

Nothing should have happened.[1]

Now, type POKE 1984+54272,Ø and press [RETURN].

Now there should be a black club at the bottom left corner of the screen.

Type POKE 16ØØ,1Ø2 and press [RETURN].

Type POKE 16ØØ+54272,7 and press [RETURN].

This should produce a yellow checkerboard figure (or shaded box) near the middle of the screen.

Poking in Designs

Clear the computer's memory (with NEW) and clear the screen.

Type the following lines:

```
100 POKE 1760,95
101 POKE 1760+54272,7
110 POKE 1761,98
111 POKE 1761+54272,7
120 POKE 1762,98
121 POKE 1762+54272,7
130 POKE 1763,105
131 POKE 1763+54272,7
```

List your program.

In each pair of lines in this program, the first puts a particular character on a particular place on the screen and the second makes it appear in a particular color.

If you look at the figure of screen **POKE** addresses on page 155, you'll see that address **1760** controls a box in the middle, about three-quarters of the way down the screen. Addresses **1761**, **1762**, and **1763** are, of course, the addresses for screen locations just to the right of **1760**. Thus, this program puts characters into these four locations. By using the table of screen display codes, look up the code numbers used in this little program. See if you can figure out the pattern that is going to appear. What color will it be?

Run your program.

[1] If you have an older model Commodore 64, you will actually get a white or light blue club on the screen at this point. That's okay—ignore it and continue with the rest of the instructions. The older machines automatically produce characters on the screen—usually in white—when you poke them in, without your having to tell the computer what color to make them. If you *do* designate a color—with the second **POKE** command, as described in this chapter—the character comes out in that color, exactly the same as with the newer machines. There is no disadvantage if you have the older machine. The procedures described in this chapter work perfectly on your computer. The point of this footnote is to make sure you are not confused by the little bonus of getting a character to appear in white before you have poked in a color.

This should produce the same little boat you produced from the keyboard in Chapter 3—only this time in yellow.

One advantage of poking in designs is that you can take advantage of the consecutive numbering of the addresses. For example, here's how to put the boat on waves:

Type the following lines:

```
60 FOR W=1744 TO 1783
70 POKE W,73
71 POKE W+54272,1
80 NEXT W
```

List your program.

```
60 FOR W=1744 TO 1783
70 POKE W,73
71 POKE W+54272,1
80 NEXT W
100 POKE 1760,95
101 POKE 1760+54272,7
110 POKE 1761,98
111 POKE 1761+54272,7
120 POKE 1762,98
121 POKE 1762+54272,7
130 POKE 1763,105
131 POKE 1763+54272,7
```

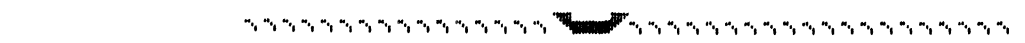
The new lines, lines **60** through **80**, are a **FOR . . . TO . . .** and **NEXT** loop that first sets **W** equal to **1744**, then to **1745**, and so forth, until it gets up to **1783**. Each time through the loop, line **70** pokes address number **W** with the screen display code **73**. Addresses **1744** to **1783** are for one 40-space screen line, the one with the boat on it. Thus, you are poking the **73** character into an entire screen line. Character **73** looks something like a wave. You are creating an ocean surface for the boat.

Line **71** pokes in the color—in this case white—for each character. You can see how handy just adding **54272** is!

The program then pokes in the boat design, as before. Notice you put the waves first and the boat second in the program. Otherwise, the waves would cover the boat.

Clear the screen (with [SHIFT] and [CLR/HOME]).

Run your program.



Similarly, you can color in an entire section of the screen, using the character that is a full box (actually, it is the reverse character of a blank space).

Type the following lines:

```
10 PRINT "□"
30 FOR S=1024 TO 1743
40 POKE S,160
41 POKE S+54272,14
50 NEXT S
```

List your program.

```
10 PRINT "□"
30 FOR S=1024 TO 1743
40 POKE S,160
41 POKE S+54272,14
50 NEXT S
60 FOR W=1744 TO 1783
70 POKE W,73
71 POKE W+54272,1
80 NEXT W
100 POKE 1760,95
101 POKE 1760+54272,7
110 POKE 1761,98
111 POKE 1761+54272,7
120 POKE 1762,98
121 POKE 1762+54272,7
130 POKE 1763,105
131 POKE 1763+54272,7
```

This time, we start out with line **10**, which clears the screen. Now that you are getting into designs that involve more of the screen, it is a good idea to have the computer clear the screen at the outset. That avoids any mixup of the design with what was on the screen before.

As for lines **30** to **50**, they make a sky! **1024** through **1738** are the **POKE** addresses for all the screen locations above the screen line that has your boat and the waves. Screen display code **160** is a full box. Line **41** makes these full boxes all come out light blue (it is a clear day). Thus, this program makes the whole upper part of the screen come out light blue.

Run your program.

You will notice one little problem! As soon as the computer finishes poking in your pretty picture, it has to go and spoil it with this: **READY**.

This is a general problem with screen graphics. Any time you make a design, when the program is finished, the design gets moved three screen lines and **READY** appears. The solution is to not let the program end!

Type 200 GOTO 200

List your program.

```
10 PRINT "□"
30 FOR S=1024 TO 1743
40 POKE S,160
41 POKE S+54272,14
50 NEXT S
60 FOR W=1744 TO 1783
70 POKE W,73
71 POKE W+54272,1
80 NEXT W
100 POKE 1760,95
101 POKE 1760+54272,7
110 POKE 1761,98
111 POKE 1761+54272,7
120 POKE 1762,98
121 POKE 1762+54272,7
130 POKE 1763,105
131 POKE 1763+54272,7
200 GOTO 200

READY.
```

Line **200** is a one-line, endless **GOTO** loop. Once the computer gets to this, it just keeps running in circles forever. While it is doing this, your picture stays on the screen in all its glory, without the cursor or **READY** or anything else. When you want to stop it, just press the [RUN/STOP] key.

Run your program.

When you get tired of looking at the picture, press the [RUN/STOP] key.

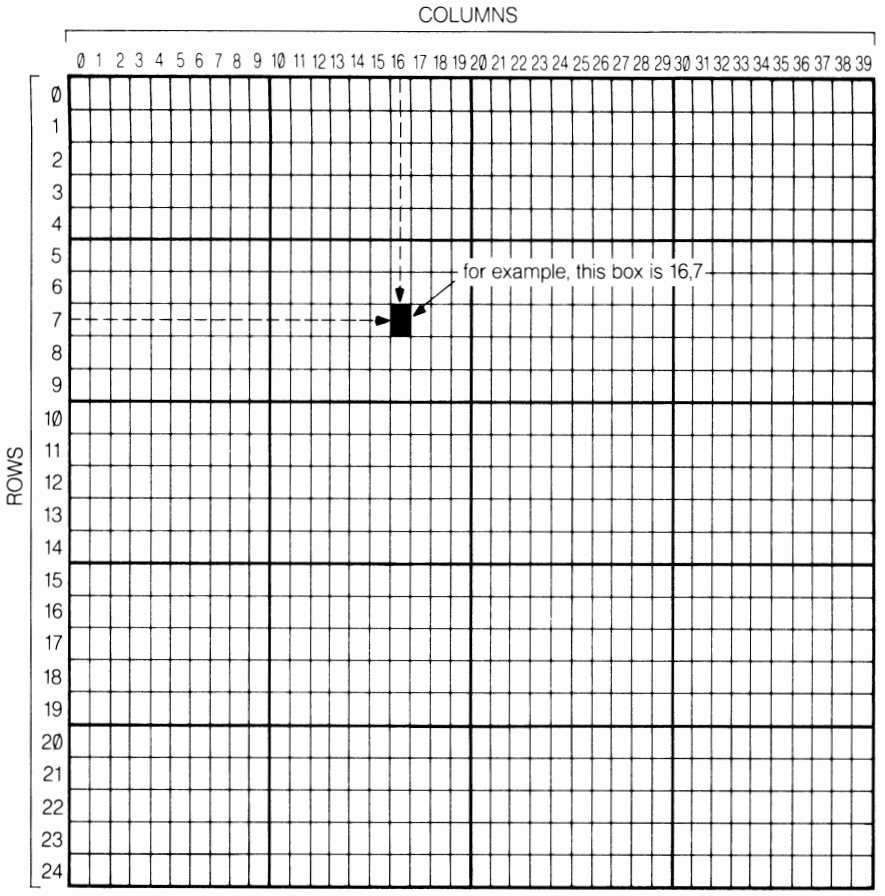
Simplifying the Process

The little boat is a simple design. As you can see from what you have done already, it would be rather tedious to program a complex design that involves, say, 50 different spaces on the screen! For each, you would have to write separate **POKE** instructions for its character and color. Because of the numbering system of the **POKE** addresses, you would probably have to look up almost every screen location address as you were producing your design.

We promised you a way to simplify this process. It cannot be made entirely simple. (By the time you are done with this book, you will look at those graphics in video games with a whole new appreciation for the work that went into them!) Some things can be done, though, to speed up the process.

The main thing that helps is to consider the screen a chart, like a piece of graph paper, with the columns across numbered from Ø to 39 and the rows down numbered from Ø to 24—as shown in the following figure.

SCREEN COLUMN AND ROW NUMBERS

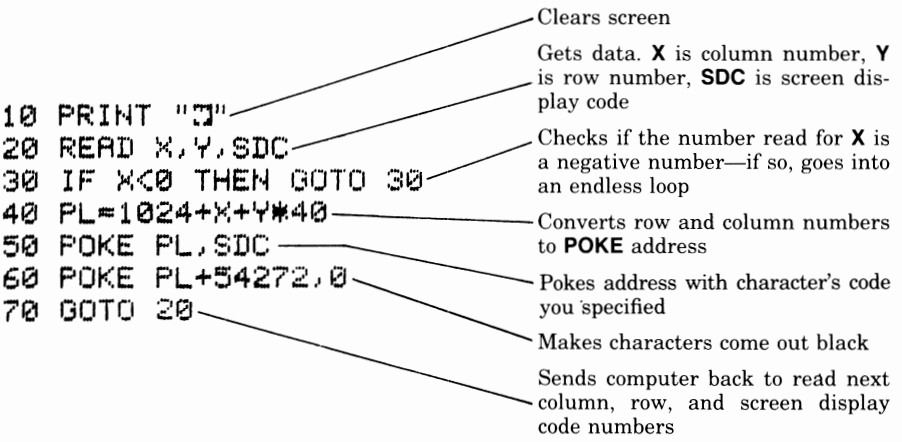


If you could use the column and row numbers for your **POKE** location, it would mean shorter numbers. More important, it would be much easier to keep track of what you were doing and to make systematic repetitions up and down or right and left. It would also be much easier to plot mathematical and statistical charts. (See Box 12-1 later in the chapter.) It would also be a help if you did not have to make two whole **POKE** program lines for each thing you wanted to display.

Both of these shortcuts are provided in one technique. Instead of making **POKE** commands, you make one or more **DATA** lines. (Remember this from Chapters 9 and 10?) On the **DATA** lines, you put three numbers for each screen place you want to poke a character: the column number, row number, and the screen display code number for the character you want. Then the program can go through a loop that reads each set of three numbers, plugs the row and column numbers into a simple arithmetic formula to convert them to a **POKE** address number, pokes that address with the screen display code number you specified, and pokes the corresponding color address (which it gets by adding **54272**) with the color you indicate.

You will also need a way to tell the computer when it has finished. This requires two things. First, you add a **DATA** line with three numbers that would never be column, row, or screen display code numbers (for example, negative numbers). Then, write into the program a check (with an **IF...THEN...** line) which, when the program has read those numbers, puts the program into an endless loop (to keep the picture on the screen).

Clear the computer's memory and type the following program lines:



List your program.

Line **10** clears the screen so that none of your picture gets mixed up with old program lines. Line **20** tells the computer to read from a **DATA** line (not in the program yet). Remember, the computer always reads the first numbers it has not used yet. If this is the first time through, it reads the first three numbers in the first **DATA** line; the second time, the next three numbers; and so forth. If it cannot find enough numbers on the first **DATA** line, it takes them from the next. (If you need to refresh yourself on this, see Chapter 9.)

The three numbers read are assigned to **X** (for the column number across the screen), **Y** (for the row number down the screen), and **SDC** (for the character's screen display code number).

Line **30** checks whether **X** (the first number read in line **20**) is a negative number. Until the end of the data, **X** is a column number (from **0** to **39**), so this **IF...THEN...** line is false. Only when you are finished does **X** equal a negative number (because you have put a negative number at the end of the last **DATA** line). At that point, the comparison in line **30** is true and the computer will then go to line **30**—where it already is. That's right. When the computer has read the last number, it starts telling itself to go to the line it is already on. That keeps anything you have designed on the screen until you press [RUN/STOP].

Line **40** converts the column and row numbers to a screen location **POKE** address, which it then remembers as **PL**. This is done

by taking the lowest possible screen number, **1024**, adding the number for the column, and then adding 40 for each row down. If you try this out, you'll see that this little formula does the proper conversion for you.

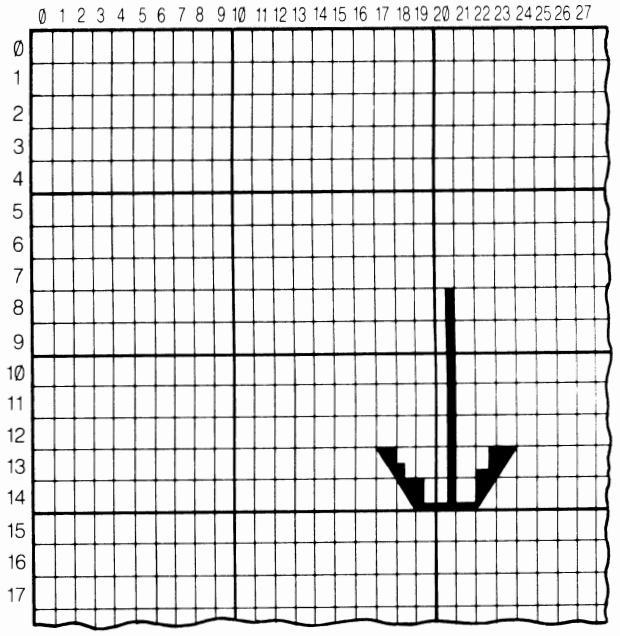
Line **50** does the actual poking into that address of the character you indicated.

Line **60** neatly takes each screen location **POKE** address calculated in line **40** and adds **54272** so that it pokes the corresponding color control address for that screen location. You could use any color here. We chose black. Of course, by using this sytem, all your characters come out the same color. We will consider at the end of the chapter a way you can make each character a different color and still use this basic system of row and column numbers.

Finally, at line **70**, the computer is sent back to line **20** to start the process again, this time with the next three numbers in the **DATA** lines.

Now, let's put in some **DATA** lines so that you can actually try this program. The following figure is a picture drawn on graph paper of a somewhat bigger boat than we would have tried with the ordinary **POKE** procedure.

BOAT FIGURE DRAWN WITH COMMODORE CHARACTERS



Now, type in the following DATA lines:

```
1000 DATA 17,13,95,18,14,95,18,13,123
1010 DATA 19,14,76,20,8,103,20,9,103
1020 DATA 20,10,103,20,11,103,20,12,103
```

```
1030 DATA 20,13,103,20,14,122,21,14,122
1040 DATA 22,13,122,22,14,105,23,13,105
1050 DATA -1,-1,-1
```

List your program and correct any typing errors. (This will be some good practice in correcting methods!)

```
10 PRINT "□"
20 READ X,Y,SDC
30 IF X<0 THEN GOTO 30
40 PL=1024+X+Y*40
50 POKE PL,SDC
60 POKE PL+54272,0
70 GOTO 20
1000 DATA 17,13,95,18,14,95,18,13,123
1010 DATA 19,14,76,20,8,103,20,9,103
1020 DATA 20,10,103,20,11,103,20,12,103
1030 DATA 20,13,103,20,14,122,21,14,122
1040 DATA 22,13,122,22,14,105,23,13,105
1050 DATA -1,-1,-1
```

Run your program. (When you get tired of looking at it, press [RUN/STOP].)

Save your program if you have a Datassette. You will be using it in the next chapter.

This whole approach for making a design is still a bit tedious. If you do many graphics, though, you quickly get used to the row-and-column-number procedure. In fact, before long you'll not only move rapidly from a drawing on graph paper to **DATA** lines, but also design things in your head and put them directly into **DATA** lines!

You could also make multicolor pictures using this procedure. You just include a color code in the **DATA** lines along with column, row, and screen display character codes. Line **20** would become **20 READ X,Y,SDC,CC**, and line **60** would become **60 POKE L+54272,CC**.

In the next chapter, you will see how all this can be applied to animation and keyboard-controlled animation. Here's where you start saving some quarters at the video arcade.

Summary

In this chapter, you become thoroughly acquainted with the **POKE** command. Each **POKE** directly changes some internal operation of the computer. Every such internal operation has a number or *address* that identifies it. For example, **POKE** address **53281** controls total screen color. However, you must know the code number to poke into that address to make that address do something. For example, the code numbers for screen color are **0** for

black, **1** for white, **2** for red, and so forth—as shown in the chart on page 152. Thus, the command **POKE 53281,0** would turn the screen black.

Address **52380** controls the color of the border around the main part of the screen. It uses the same color codes as for the main part of the screen.

The various **POKE** addresses and code numbers do not follow any obvious pattern. When you want to use them, just look them up in tables such as those in this chapter, in Appendix B, and in the quick-reference table for graphics and sound in Appendix H.

The screen is divided into a 40 (across) by 25 (down) pattern of 1000 little boxes. Each box has its own **POKE** address. For poking in a particular character you want to appear, the address numbers begin at **1024** in the upper left corner and continue, left to right, across each screen line. To put a character on the screen, poke the address corresponding to the desired screen location with the *screen display code* number for the desired character. Then you must poke the address that is the sum of that address plus **54272**, followed by the color code number. For example, **34** is the screen display code number for #. To put a white # in the upper left corner of the screen, you would tell the computer **POKE 1024,34** and **POKE 1024+54272,1**. A table of screen display code numbers is given on page 156. (Note that the screen display code is not the same as the character code you learned to use in Chapter 11.) The color code numbers are the same as the ones you use for screen and border color given in the table on page 152.

Because the **POKE** addresses for screen locations follow a logical sequence across the screen, you can write fairly simple programs to put a character all the way across the screen. For example, the lines **10 FOR P=1024 TO 1063**, **20 POKE P,63**, **21 POKE P+54272,7**, **30 NEXT P** would put a question mark (the screen display code **63** character) in yellow (color code **7**) across the top line of the screen (which goes from **1024** to **1063**). You could also use such a loop with more than 40 repetitions to put a character in a whole section of the screen.

When writing a program to poke designs on the screen, start with a clear-screen program line. This gets rid of anything already on the screen that might interfere with your design. Finish the program with an endless **GOTO** loop—such as **200 GOTO 200**—so the program keeps running until you press [RUN/STOP]. This keeps your full design on the screen by not letting the computer put up its **READY** indicator.

If you want to make a design that is more complex, the procedure we have just described is rather slow. A quicker method is to think of the screen as a chart 40 columns across and 25 rows down. First, draw your design on graph paper. Then, for each box you have drawn in, put the column and row number and that character's screen display code number in a program **DATA** line. Finally, the **DATA** lines go into a program that reads them, con-

Box 12-1
Graphs for Business and Math

Do you want to show your employees a picture of who is contributing most, plot different formulas for your math teacher, or see for yourself—graphically—just how sales have gone during the last year?

The system of poking characters onto the screen lends itself very well to making graphs. Most graphs are just "**X** by **Y**" plots of something. In math, for example, you might want to plot a number on the across axis (**X**), and what it is squared on the up and down axis (**Y**). This kind of basic graph requires simply poking a character into a series of addresses that are determined by the **X** and **Y** values of the formula. There are two steps involved in doing this: getting your **X** and **Y** values within the range of 40 across and 25 high, and converting those numbers to **POKE** addresses with a formula that starts with the lower left corner screen address, adds the **X** value, and *subtracts* (because you want to go *up* the screen) **Y** times **40**. The following program illustrates this:

```
10 PRINT "J"
20 FOR CLM=0 TO 39
30 X=CLM/10
40 Y=INT(X*X)
50 POKE 1984+CLM-40*Y,42
51 POKE 1984+CLM-40*Y+54272,14
60 NEXT CLM
70 GOTO 70
```

Line **10** gives you a clear screen. Line **20** starts a loop that goes through all 40 columns across. Line **30** sets **X** equal to a multiple of the number of columns, one that will make **Y** always less than **25**. Line **40** makes **Y** equal to **X** squared and converts it to a whole number. Line **50** converts **X** and **Y** to a **POKE** address and pokes in the code number for the asterisk. Line **51** makes asterisks light blue. Line **60** sends the computer back to line **20**. Line **80** is an endless loop to keep the full graph on screen.

verts the column and row numbers to the appropriate **POKE** addresses, pokes those addresses with the screen display code numbers specified, and pokes all the screen color addresses for those locations with a color you give (in a single program line).

Graphs for business or educational purposes are more often bar graphs. You make a bar by having the computer systematically poke in addresses going up along a single line for the number of spaces that correspond to what you want to show. For example, the program below makes a bar graph of one year of monthly sales. You just type the number of each month's sales (in thousands, with a maximum of 25 so you do not exceed screen size). The program converts those numbers into the height of the bars. The computer is sent through a loop for each month. Each time through the loop it puts in one more box higher up. The number of times through the loop is determined by the number of sales you put in. The following is what the program looks like:

```
10 DIM SA(12)
20 FOR MO=1 TO 12
30 PRINT "  SALES FOR MONTH NUMBER";MO
40 INPUT SA(MO)
50 NEXT MO
60 PRINT "□"
70 FOR MO=1 TO 12
80 FOR HT=1 TO SA(MO)
90 POKE 1983+3*MO-40*HT,160
91 POKE 1983+3*MO-40*HT+54272,7
100 NEXT HT
110 NEXT MO
120 GOTO 120
```

Line **10** tells the computer the dimension of variable **SA**. Line **20** starts the loop to get 12 months' worth of sales data. Line **30** is a prompt line. Line **40** takes in your sales info for that month. Remember, this number can't be more than 25, which is the height of the screen. Sales are usually of some multiple, such as thousands of pancakes or pairs of alligator shoes. Line **50** sends the computer back to line **20**. Line **60** makes a clean screen for the bar graph. Line **70** starts a large loop to poke in the bars, one month at a time. Line **80** starts the loop to make one bar, of height **HT**. Line **90** pokes one full box of bar each time this line is passed. Each row of boxes will be in the column for the number of the month times **3** (to spread out the bars) and will keep getting poked, going up that column as many times as there were sales for that month. Line **91** makes the bars yellow. Line **120** is an endless loop to keep the graph on the screen without **READY** showing up.

Concepts and Terms Introduced in Chapter 12

POKE

Address

Code number

Using **POKE** with address and code numbers

Poking screen and border colors

Screen and border color codes

[RUN/STOP] and [RESTORE] to return to normal screen and border colors

Poking color characters onto the screen

Screen location **POKE** addresses for characters

Screen display code for characters

Screen location **POKE** addresses for color

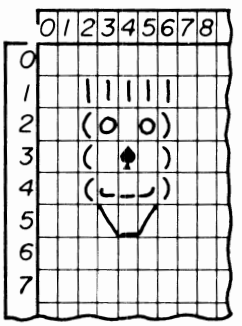
Putting a character onto an entire screen line or section of the screen

Using a 40 by 25 pattern and graph paper to make screen designs

Practice Exercises

(Answer and comment in Appendix A.)

Use the program on p. 162 to put the following design in yellow on the top left corner of the screen. (When you are finished, save your program; you will use it in the practice exercises in Chapter 13.)



PART III CHAPTER 13

Animation and Controlling Animation from the Keyboard

In the last chapter, you learned how to use **POKE** to change screen and border colors and to create designs on the screen. In this chapter, you will learn how to animate screen designs and control their movement from the keyboard.

Although you will be animating very simple designs, the programming procedures you will learn apply to the most complex, interesting designs possible. For example, in the next chapter, you will apply these procedures to what is called *sprite* animation.

Animation by Changing Part of a Design

Load, or retype (from p. 164), your program for the boat picture from the end of the last chapter.

Edit line 30 to read 30 IF X<0 THEN GOTO 80

Type the following lines:

```
80 POKE 1364,95
90 POKE 1364,103
100 GOTO 80
```

List and check your typing.

```
10 PRINT "□"
20 READ X,Y,SDC
30 IF X<0 THEN GOTO 80
40 PL=1024+X+Y*40
50 POKE PL,SDC
60 POKE PL+54272,0
70 GOTO 20
80 POKE 1364,95
90 POKE 1364,103
100 GOTO 80
1000 DATA 17,13,95,18,14,95,18,13,123
```

```
10 PRINT "J"
20 READ X,Y,SDC
30 IF X<0 THEN GOTO 80
40 PL=1024+X+Y*40
50 POKE PL,SDC
60 POKE PL+54272,0
70 GOTO 20
80 POKE 1364,95
90 POKE 1364,103
100 GOTO 80
1000 DATA 17,13,95,18,14,95,18,13,123
1010 DATA 19,14,76,20,8,103,20,9,103
1020 DATA 20,10,103,20,11,103,20,12,103
1030 DATA 20,13,103,20,14,122,21,14,122
1040 DATA 22,13,122,22,14,105,23,13,105
1050 DATA -1,-1,-1
```

You will recall from the last chapter that this program takes information that you provide in **DATA** lines about your design and converts that information into appropriate **POKE** commands. The special advantage of this approach is that you provide the screen location information in the form of column and row numbers instead of those unwieldy **POKE** addresses. It also saves the trouble of writing lots of **POKE** commands.

In this version, you have changed one line and added three more. First, the program makes the boat, just as before. Then, instead of going into an endless loop at line **30**, it is told to go to line **80**. Here's where the fun begins.

Line **80** is a new **POKE** instruction. Since you are going to be changing something at only one location, it is easier in this case to look up its actual address than to route the computer back through the formula for converting a row and column number to a **POKE** address. The address in this **POKE** command is for the top of the ship's mast. (Figure it out yourself, as a little exercise.) The screen display code number **95** is a triangle character—a sort of flag. In other words, the mast will have a flag at the top.

Line **90** pokes the usual mast character back in again. Line **100** sends the computer back to the line that pokes the flag.

Thus, the flag flashes on and off, looking as though it is waving in the wind.

Run the program.

With a little imagination, this procedure can be used to create all kinds of dramatic effects. One person we know programmed an entire animated band. The pianist, for example, was made to look as though he was playing by changing his hand from "." to "," over the "piano keyboard."

Animating Motion Across the Screen, and RESTORE

Now, let's see if we can make the ship move across the screen.

Delete lines 80, 90, and 100. (We will just do one kind of animation at a time while you are learning.)

Add the following lines:

```
8 FOR A=0 TO 10
45 PL=PL+A
80 RESTORE
90 NEXT A
```

List your program.

```
8 FOR A=0 TO 10
10 PRINT "□"
20 READ X,Y,SDC
30 IF X<0 THEN GOTO 80
40 PL=1024+X+Y*40
45 PL=PL+A
50 POKE PL,SDC
60 POKE PL+54272,0
70 GOTO 20
80 RESTORE
90 NEXT A
1000 DATA 17,13,95,18,14,95,18,13,123
1010 DATA 19,14,76,20,8,103,20,9,103
1020 DATA 20,10,103,20,11,103,20,12,103
1030 DATA 20,13,103,20,14,122,21,14,122
1040 DATA 22,13,122,22,14,105,23,13,105
1050 DATA -1,-1,-1
```

This time, the program begins with line **8**, which puts almost all of the previous program in a loop that ends with line **100**. As you will recall from previous chapters, the procedure in a **FOR . . . TO . . .** and **NEXT** loop is that the first time through the program, the computer sets the variable in it equal to the first number (in this case, to **0**). Then it proceeds normally through the program until it gets to the **NEXT** instruction. **NEXT** sends the computer directly back to the **FOR . . . TO . . .** line and sets the variable one number higher (in this case, to **1**). It continues in this way until the variable has reached the limit set in the **FOR . . . TO . . .** instruction (in this case, **10**).

Line **10** clears the screen, and line **20** reads the values for **X**, **Y**, and **SDC** from the **DATA** lines. Line **30** checks whether **X** is a negative number (which, in this program, is the way you indicate the end of the **DATA** list). If the list is not finished, the computer goes on.

Line **40** calculates the **POKE** address equivalent, called the **PL**, of the column and row number you specified.

Line **45** takes the **POKE** address for the screen location and adds the value of **A**. The first time through, since **A** equals **0**, it has no effect. The second time through, since **A** equals **1**, each **POKE** address becomes one number higher. One number higher is one space farther to the right (except when the design comes to

```
8 FOR A=0 TO 10
10 PRINT "□"
20 READ X,Y,SDC
30 IF X<0 THEN GOTO 80
40 PL=1024+X+Y*40
45 PL=PL+A
50 POKE PL,SDC
60 POKE PL+54272,0
70 GOTO 20
80 RESTORE
90 NEXT A
1000 DATA 17,13,95,18,14,95,18,13,123
1010 DATA 19,14,76,20,8,103,20,9,103
1020 DATA 20,10,103,20,11,103,20,12,103
1030 DATA 20,13,103,20,14,122,21,14,122
1040 DATA 22,13,122,22,14,105,23,13,105
1050 DATA -1,-1,-1
```

the right edge of the screen line, and then one number higher would mean going down to the next line on the far *left* side of the screen). Hence, each time through the loop, the figure moves to the right.

The only other thing new in this procedure is the problem of getting the computer to read the **DATA** lines a second time. When the computer has taken in all the data and poked in the design, it finally finds a negative number in the last **DATA** line. Usually, that indicates that everything is done, and the computer goes into an endless loop. This time, we want the computer to go through the larger loop again so that it can redraw the picture one space over to the right.

The key to accomplishing this is **80 RESTORE**. The **RESTORE** command should not be confused with the [RESTORE] key. **RESTORE** is another typed program-line instruction, like **GOTO** or **PRINT**. It tells the computer to start reading **DATA** lines *again,* beginning from the first **DATA** line.

In this case, the negative numbers at the end of the data and the **IF . . . THEN . . .** line do not tell the computer to go into an endless loop and effectively stop the program. Instead, they cue it to read again, thanks to **80 RESTORE**.

Line **90** ends the **A** loop, sending the computer back to line **8**, where it starts over with **A** being one number higher.

Run the program.

The boat "moves" across the screen—in the sense that it is repeatedly redrawn one space farther to the right. Unfortunately, with a figure this large, it takes the computer a little while to draw it each time. Thus, the effect is of a boat constantly reappearing piecemeal from the fog, moved over a little bit. This would not be a problem with a smaller figure. Nor will it be when you learn (in the next chapter) how to make sprites. But bear with this for now. You are learning the *principles* of animation. You will use them with more *panache* later on.

You can use this same loop procedure to move the figure from right to left—for example, with the line **8 FOR A=0 TO −10 STEP −1**. Or it could be made to go farther across the screen, with something like **8 FOR A=−10 TO 10**. Whatever you do, be careful not to let the figure go so far in either direction that it runs off the edge of the screen! (It isn't the end of the world though—because the world *is* round, and your boat will appear at the other side! If only Columbus had had a computer to show his crew. . . .)

You can also make this design move up and down by having the variable change what **Y** (the row number) is set at—for example, with the line **35 Y=Y+A**. You can make it move more slowly by putting in a delay loop each time it moves over one place. Or you can make it move faster by having the changing variable increase by two each time—for example, **8 FOR A=0 TO 10 STEP 2**.

One difficulty with this whole approach is that each time the design is redrawn, the screen is cleared. This is necessary because otherwise the new boat would be superimposed over the old one, leaving a kind of trail of flotsam and jetsam as it proceeded across the screen. Try it.

Delete line 10 and run the program.

Clearing the screen avoids this problem.

Type line 10 back in: 10 PRINT "♡"

On the other hand, clearing the screen also wipes out any background you might want to put in (such as clouds or sun in the sky, or other boats). There are two solutions to this. First, you can put a border of blank spaces along the edges of the figure itself. Thus, when the design is redrawn one space over to the right, the invisible blank space covers what was before the left edge of the ship. Or you can have the background also repoked each time you go through the loop—but without changing its location. This whole problem, however, does not arise with the sprite graphics you will learn in the next chapter. Sprite designs are produced in such a way that they automatically leave no trail.

Controlling Animation from the Keyboard, and GET

Often, you want the person at the keyboard to be able to move a screen design around at will. For example, you may want to move an arrow that points to various parts of a graph, or a character or vehicle in some kind of game.

This is fairly straightforward, now that you understand ordinary animation. With the animation procedure, the design was repeatedly redrawn one space farther over by adding to the **POKE** addresses a number being changed by a **FOR . . . TO . . .** and **NEXT** loop. To move the design by pressing keys on the keyboard, you must arrange for the pressing of certain keys, of your choice, to signal the computer to redraw the design, adding or subtracting **1** to the **POKE** addresses.

One way you have learned to signal the computer while the program is running is the **INPUT** procedure. But this is pretty slow, since you must press [RETURN] after each entry.

There is a better way (for this purpose). It is a new, typed programming command: **GET**. **GET** takes information and assigns it to a variable, just as **INPUT** does. But **GET** takes the information **directly** from what you type—immediately. It doesn't wait for you to press [RETURN].

GET, however, only allows you to give the computer one key's worth of information at a time. There is no chance to consider what you have typed before it gets used. Thus, for most purposes, **INPUT** is better, but for *immediate* communication—as in what you are doing here—**GET** has the edge.

One problem with **GET** is that it is *too* fast. If you don't type something, the computer goes on anyway, making the variable in the **GET** instruction equal to **Ø**. You can get around that. After a **GET** line, add an **IF . . . THEN . . .** line that says if the **GET** variable equals **Ø**, then go back to get **GET** line.[1] A typical sequence would look like this:

```
500 GET K
510 IF K=0 THEN GOTO 500
```

Applying this to your boat program, we would use **GET** like this:

```
8 A=0
10 PRINT "□"
90 GET K
100 IF K=0 THEN GOTO 90
110 IF K=8 THEN A=A-1
120 IF K=9 THEN A=A+1
130 GOTO 10
```

Type the following lines:

```
8 A=0
10 PRINT "□"
20 READ X,Y,SDC
30 IF X<0 THEN GOTO 80
40 PL=1024+X+Y*40
45 PL=PL+A
50 POKE PL,SDC
60 POKE PL+54272,0
70 GOTO 20
80 RESTORE
90 GET K
100 IF K=0 THEN GOTO 90
110 IF K=8 THEN A=A-1
120 IF K=9 THEN A=A+1
130 GOTO 10
1000 DATA 17,13,95,18,14,95,18,13,123
1010 DATA 19,14,76,20,8,103,20,9,103
1020 DATA 20,10,103,20,11,103,20,12,103
1030 DATA 20,13,103,20,14,122,21,14,122
1040 DATA 22,13,122,22,14,105,23,13,105
1050 DATA -1,-1,-1
```

List your program.

[1]Often, you will want to get a key that has some letter or symbol on it instead of a number. In that case, you must get it as a string variable. If you don't press a key fast enough, it is set equal to an "empty string" (signified by ""). A sequence to avoid this would look like this: **500 GET K$,IF K$=”” THEN GOTO 500.**

When you run this program, the computer pokes in the ship as usual. Then, when it runs out of data, line **30** sends the computer to line **80.** There, **RESTORE** sets it up to read through the data again. But before it does so, it comes to line **90**, the **GET** line.

Until now, the whole thing has happened almost instantaneously from the moment you typed **RUN.** So, as soon as you see the picture, **GET** is ready to set **K**, (we could have picked any variable name) equal to any number you type. To make it easy to remember what's going on, use the [8] and [9] keys because on their tops they have parentheses that point left and right. If you press the [8] key, line 11 sets **A** one number lower than what it was before. (The first time, **A** equals **0**, so the [8] key would make it **−1**.) If you press the [9] key, line **120** sets **A** one number higher than before.

The result is that the next time the pieces of the figure are poked in at line **50**, **A** is added to each **POKE** address. Thus, if **A** is one number higher than the last time, the whole figure moves to the right. If it is one number lower, the whole figure moves to the left. You can keep tapping the key to keep it moving in a particular direction, or back and forth. Again, if you move too far in one direction, you will hit the edge of the screen and come around the other side.

Run the program.

When you see the design, press the [8] or [9] key to make it move one way or the other. Play around with it a while.

When you have tried it enough and feel you understand it, press the [RUN/STOP] key to stop the program.

These principles apply to making a figure move up and down. You could even designate four keys—one for each direction—and according to what was pressed, add **1** to **A** (to make the boat go right), subtract **1** (to make it go left), add **40** (to make it go down one full screen line), or subtract **40** (to make it go up one full screen line).

Congratulations. You have learned the fundamentals of computer animation and keyboard-controlled computer animation. Video-game designers make fortunes playing around like this. Of course, their programs are a little more complicated, but they're nothing you couldn't learn to do.

In the next chapter, you will learn how to use these procedures with a very impressive feature of the Commodore 64—sprites. These are designs that, once you program them in, are movable, expandable, and interactable—with relatively little programming work. The sprite procedure makes complex animation very easy to program.

Summary

In this chapter, you learn how to make your designs move either by themselves or at your command from the keyboard.

One way to create animation is to change part of the design. The program pokes in the basic design (such as a ship with an empty mast). Then it pokes in a different character for a particular screen location (such as a flag-like character in the location at the top of the mast). Then, with a loop, it alternates between poking back in the original character for that location and poking in the new one. This makes the character appear to flash (or look like a flag waving in the wind!).

To move a design across the screen, first set up a **FOR . . . TO . . .** and **NEXT** loop that systematically increases some variable by **1** each time the computer goes through the loop. Then have it add to the **POKE** addresses the value of the variable. When the program pokes in the design, the design is redrawn one space over. The process repeats each time through the loop and the design moves across the screen.

To have the computer redraw the figure (using the procedure you learned in Chapter 12) again and again, it must read the same **DATA** lines each time. That is, each time through the loop, the computer has to know to read the same information again. This is done with the **RESTORE** instruction, which tells the computer to start reading **DATA** lines from the beginning.

To assure that the old drawings of the design are not left on the screen trailing behind your moving design, the program must clear the screen before each redrawing of the design. An alternative to this is to put an "invisible border" of blank spaces at the side edges of the design so that when it is redrawn one space over, the blanks cover the leftover part of the old design.

It is also possible for the person at the keyboard to control the movement of the design. To do this, the program must allow for the pressing of designated keys to signal the computer to add **1** (to move right) or subtract **1** (to move left) to all the **POKE** addresses of a design, and then redraw it. To do this, use **GET. GET** sets a variable equal to whatever key you press without waiting for you to hit [RETURN]. For example, **90 GET K** sets **K** equal to **9** if you press the [9] key. But **GET** is so fast that you have to add a special **IF . . . THEN . . .** line to keep the program going back to the **GET** line until you have pressed a key and made **K** some value other than **0**. For example, **100 IF K=0 THEN GOTO 90** does the job.

Once the program gets your number, it evaluates it and either adds **1** to the variable or subtracts **1** from it. It then adds the variable to all the **POKE** addresses for the design. (You could also designate keys for moving the figure up and down, by subtracting or adding **40** to the **POKE** addresses.)

Terms and Concepts Introduced in Chapter 13

Animation by making a character appear and disappear

Moving a design across the screen

RESTORE

Keyboard-controlled movement of a design

GET

Practice Exercises

(Answers and comments in Appendix A.)

1. Load (or retype) the program to make the face from the practice exercise in Chapter 12. See if you can make the left eye *wink*.

2. Make the face move across the screen.

3. Change the program so that you can make the face move right or left with the [8] and [9] keys.

4. Change the program so that you can also move the face up or down with the [6] and [7] keys.

PART III CHAPTER 14

Commodore 64 Sprites—Ghosts That Move in the Night

Or in the day.

A *sprite* is a little design or mosaic picture, made up of 24 (horizontal) by 21 (vertical) dots on the screen. Once you design a sprite you can move it around on the screen, lengthen or fatten it, or even have several at once. Sprites are pretty darn wonderful.

As you can see from the last chapter, a lot of intense work is required to animate something as complicated as the average video game. The Commodore 64 eliminates much of that work because it has a special section of its internal operation set aside for animating small figures—these things called sprites—which are crucial to most games. Although there is some effort involved in producing your first sprite, you'll never have to do it again. The effort is far less than would otherwise be necessary to accomplish all that a sprite can do. Sprites are one of the features of the Commodore 64 that make it a marvel of space-age technology. So—hang on. This may be the toughest chapter to understand, but it is the only chapter you don't need to understand perfectly to use. You'll see what we mean soon.

First, we'll give you a general picture of what is going on in the machine. Then we'll show you some very mechanical steps to make and move sprites, which you can use immediately and understand in a general way without knowing much about those deep mysteries of machine language!

In particular, we'll give you a program that you don't need to grasp fully. You may not understand it all, but it will make designing sprites child's play.

Overview of Steps in Designing a Sprite

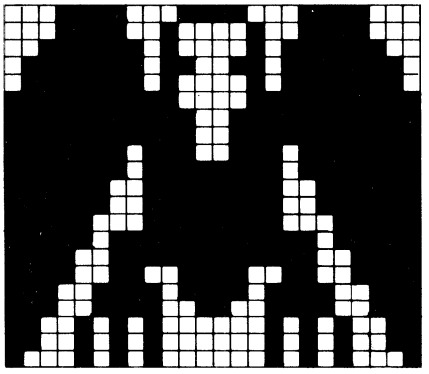
There are basically five steps involved in producing a sprite:

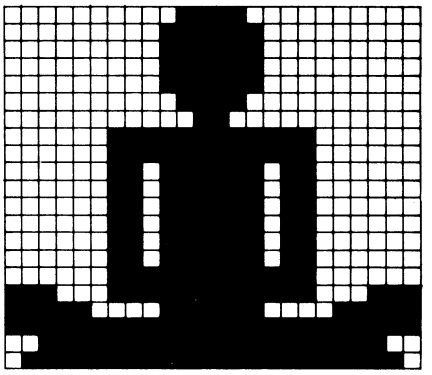
1. *Let the computer know what place in its memory will hold the sprite design information.*

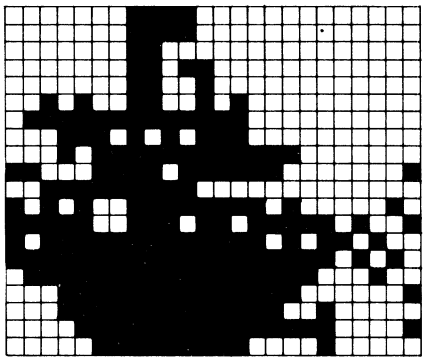
   This requires just one **POKE** command per sprite. In a minute, we will give you the **POKE** addresses and the code numbers to put in them.

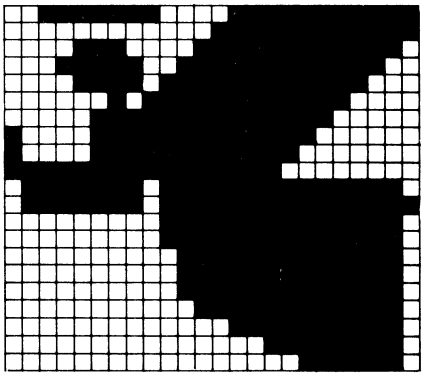
2. *Put in the description of the sprite.*

   You first design your sprite on a piece of graph paper, with 24 columns across the top and 21 rows down. You cannot use any fancy characters. Each box is either completely filled in or completely empty. That's all. Thus, it is more like a mosaic than the figures you made before. The following figures are some examples.









A sprite is actually quite small. Each box in the sprite is the size of one of the dots that makes up the normal Commodore 64 characters. The whole sprite occupies a space the size of about three characters across and three screen lines down. Thus, you can make pretty precise figures. (Any fancy design characters would be too tiny to see, anyway.) Later you will see that, once you put the sprite design into the computer's memory, you can make it either twice as wide or twice as high, or both, but you will still have only 24 by 21 boxes, so the fineness of detail will not increase.

According to the Commodore 64 instruction manual, once you have drawn your sprite on graph paper, things get complicated. You must convert your design into numbers the computer can understand—a messy business because the computer uses binary numbers. Then you have to carefully poke them all in the right place, in a particular, somewhat complex order.

Fortunately, however, you will not have to learn this new numbering system or convert your picture to numbers at all, or even have to laboriously poke in all those numbers. Our "Handy-Dandy Sprite Maker" program (which we will reveal shortly) will solve all your sprite-making problems!

3. *Tell the computer what color to make the sprite.*

   This requires a single **POKE** for each sprite and uses the standard color code you are already familiar with from the last two chapters.

4. *Tell the computer where to put the sprite on the screen.*

   The screen, in this case, is considered to be a *very* fine-grained checkerboard with—count 'em (or better yet, don't count 'em)—320 columns across and 200 rows down. This is *not* the same layout as for poking characters on the screen. For every normal character space, there are 8 of these fine-grained boxes across and 8 down (64 in all). Figure it out: 40 across times 8 equals 320; 25 down times 8 equals 200. In addition, there are some spaces off the edges (both sides and top and bottom) so that you can mover your sprite partially or completely off the screen. Your sprite is, on this fine grained checkerboard. a figure of 24 by 21. All of this should be pretty clearly illustrated in the next figure.

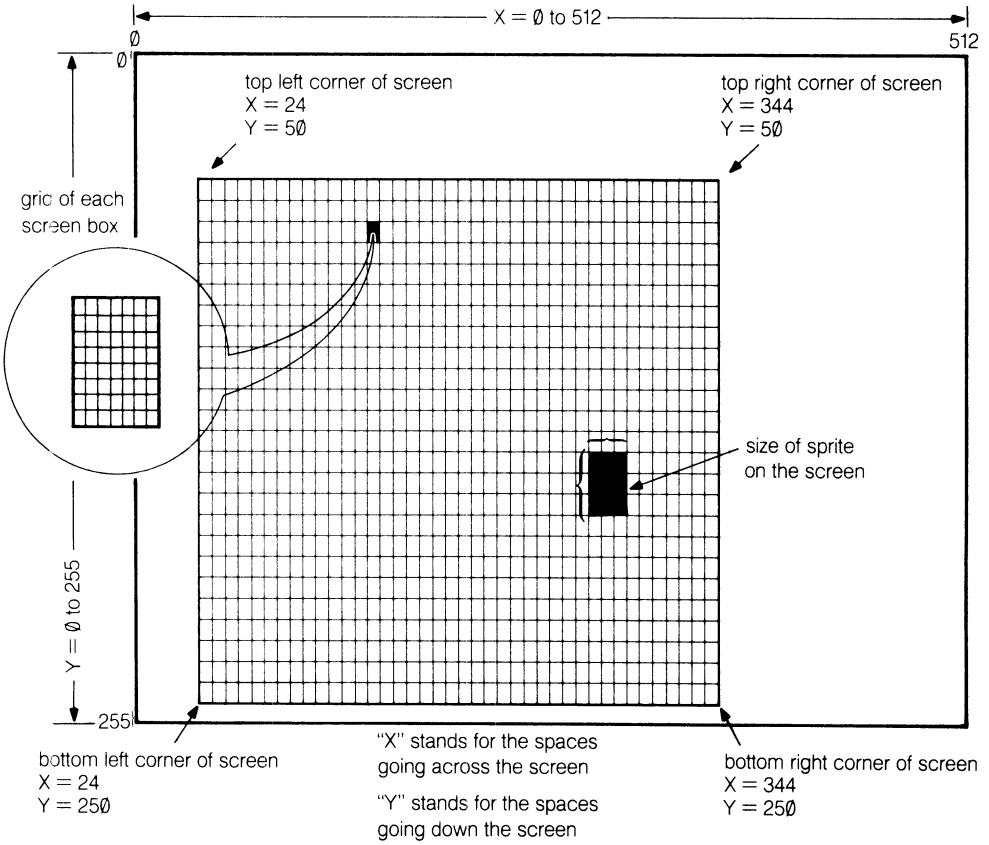
   On this fine-grained checkerboard of a screen, tell the computer just where you want your sprite to appear. On the right middle? Half-way down the left side? At the very bottom right corner?

   The column and row numbers you pick determine where the top left corner of the sprite will appear.

   So far, this should not be too difficult. It gets complicated, however, when you try to poke this information into the computer. Unfortunately, more than two **POKE** addresses are involved, and the relations among them are not straightforward.[1]

[1] Actually, poking in the up-down direction number is pretty straightforward, but because any particular **POKE** address can take only code numbers up to **255**, you run into a

SPRITE LOCATION LAYOUT



The "Handy-Dandy Sprite Maker" comes to the rescue. All we will ask you to do is type the column and row number, and the program will convert this to a form the computer can understand.

5. *Turn the sprite on.*

That is, tell the computer to actually produce on the screen the sprite you have described. This just takes a single **POKE** line.

problem with the across number. The Commodore 64 solves this by having you poke in the column number if you are on the left side of the screen. If you are on the right, you must poke in two different things: an adjusted version of the column number (the actual number minus **255)** into the usual column-number **POKE** address, and a special code into a second address, to show you want this to apply to the right side of the screen.

"The Handy-Dandy Sprite Maker"—for One Sprite

Clear the computer's memory (with NEW) and type the following lines:

```
10 PRINT"□"
20 F=53248
30 POKE 2040,192
40 GOSUB 600
50 POKE F+39,0
60 X=180
70 Y=210
80 GOSUB 400
90 POKE F+21,1
100 END
```

- Clears screen
- First of main sprite-making addresses
- Technical item that tells computer where sprite design information is in the computer's memory
- To sprite design subroutine, not in the program yet
- Sets sprite color—in this case to **0**, for black
- **X** is column number to place sprite on the screen
- **Y** is row number
- To subroutine that converts column and row numbers to appropriate **POKE** addresses and codes
- Turns sprite on

```
400 J=X
410 G=0
420 IF X>255 THEN G=G+1
430 POKE F+16,G
440 IF X>255 THEN J=X-255
450 POKE F,J
460 POKE F+1,Y
470 RETURN
```

- Lines **400** through **470** are a subroutine that converts column and row numbers to appropriate **POKE** addresses and codes; the details of how it works are not important

List and check your typing carefully.

This is the entire program for defining a sprite except for the actual drawing of the sprite, which you will do in a moment. First, let's explain how this main part of the program works.

Line **10** clears the screen. Line **20** sets **F** equal to the number **53248.** This is the first **POKE** address for the part of the computer that handles a lot (but not all) of sprite making. Setting a variable equal to this number saves typing in some large numbers. For example, in line **450** we just have to say **POKE F,X.** The computer will convert that to **POKE 53248,X.**

Of course, if you were using this number only once, it would not be worth having a separate program line to make it equal to a variable. But you can see that it is used quite a bit. For example, at line **50** we need to poke address number **53287.** But since this is just **39** higher than the address that starts the sprite section of memory, you can just say **POKE F+39.** The program would

work just as well without this procedure, except that the typing would be harder![2]

Line **30** tells the computer what section of its memory will be used to store the sprite design information. Since you are only making one sprite, you poke **2040,192.** The addresses and code numbers are different for more sprites (we'll get to those later). 30 is a technical line you do not need to understand. Just be sure you have it in the program, or you won't get a sprite!

```
10 PRINT"□"
20 F=53248
30 POKE 2040,192
40 GOSUB 600
50 POKE F+39,0
60 X=180
70 Y=210
80 GOSUB 400
90 POKE F+21,1
100 END
```

Line **40** sends the computer to the big subroutine you don't have yet, which will start at line **600.** (A subroutine, remember, is a kind of miniprogram that you use one or more times within the main program. When the computer finishes a subroutine, it returns to where it came from in the main program.) Although you have not yet written this subroutine—we will show you how to do that in just a moment—you should know that it will be the part of the program that actually gives the computer the design of the sprite.

Line **50** tells the computer what color to make the sprite. The **POKE** location for the color of the first sprite is **F+39** (which is actually **53287** if you wanted to write it all out). You can make your sprite any color you want. For the code number that goes after the **POKE** address for sprite color, use the usual **0** through **15** color code you used in previous chapters. (See the color code table in Chapter 12.) This example uses **0,** the code for black.

Lines **60** and **70** give the column and row numbers for the screen location of the top left corner of the sprite. Remember, the **X** numbers go from left to right (counting the off-screen numbers, from **0** to **511**). And the **Y** numbers go from top to bottom (from **0** to **255**), as shown in the second figure in this chapter. In this example, **X** equals **180** and **Y** equals **210.** That puts your sprite about halfway across and near the bottom of the screen.

Line **80** sends the computer to a subroutine (lines **400** to **470**) that accomplishes the technical business of getting your **X** and **Y** values into the computer's memory in a way it can understand. It is not necessary to understand the details of this subroutine, either.[3]

Line **90** turns on the sprite. **F+21** is the **POKE** address and **1** is the code for turning on the first sprite (which is the only one you've got with this version of the program). **0** would turn off the sprite.

[2]For those of you who have been trying to follow the description of how to make sprites in the Commodore 64 *User's Guide* or *Programmer's Reference Manual:* They use the same procedure of setting **53248** equal to a variable, except that they use **V.** Obviously, it doesn't matter what letter you use, as long as you are consistent. The "VIC Chip" is the name that both reference books use to label the place that the sprite business starts; therefore, they use the letter **V** to label the number. We think it is easier to remember it as the first **POKE** address of the sprite area, so we label it **F** for first.

[3]If you do try to understand it, you may be puzzled by what seem like unnecessary steps. That's because later we will use this same subroutine for animation, keyboard-controlled animation, and—using a modified version—for making two or more sprites.

Line **100** tells the computer the program is finished. Without this line, the computer would go on to the next line which is where the subroutines begin. To go to that again now would lead to an error message instead of a sprite.

Now, let's get on with the program. What remains is the subroutine (and some **DATA** lines associated with it) referred to in line **40**. It actually puts in the sprite design.

Type the following lines:

```
600 P=12287
610 FOR A=1 TO 21
620 READ D$
630 FOR B=0 TO 2
640 K=0
650 FOR C=1 TO 8
660 IF MID$(D$,B*8+C,1)="H" THEN K=K+2↑(
8-C)
670 NEXT C
680 P=P+1
690 POKE P,K
700 NEXT B
710 NEXT A
720 RETURN
```

When you type the following lines, put 24 spaces between quotes. There should be four blank spaces between the last quote and the edge of the screen.

```
1000 DATA "                        "
1010 DATA "                        "
1020 DATA "                        "
1030 DATA "                        "
1040 DATA "                        "
1050 DATA "                        "
1060 DATA "                        "
1070 DATA "                        "
1080 DATA "                        "
1090 DATA "                        "
1100 DATA "                        "
1110 DATA "                        "
1120 DATA "                        "
1130 DATA "                        "
1140 DATA "                        "
1150 DATA "                        "
1160 DATA "                        "
1170 DATA "                        "
1180 DATA "                        "
1190 DATA "                        "
1200 DATA "                        "
```

Type LIST 600–720 and press [RETURN].

Your entire program is too long to see on the screen all at once. If you listed it, the computer would flash it all by and end up showing you just the last 22 lines. You can make the computer list just part of a program by using the format shown above.

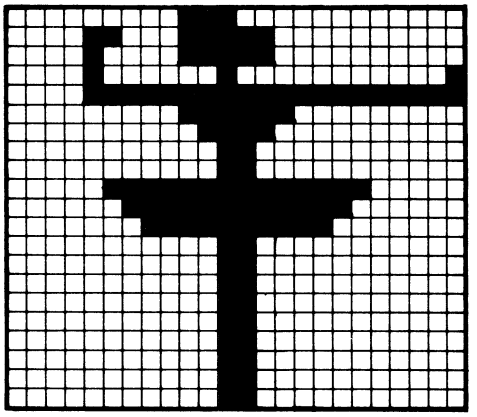
Now, what does this subroutine do? Basically, you will fill in the blanks in the **DATA** lines in the same way you filled in the blanks in **PRINT** lines for the business graph you made in Chapter 8.

```
600 P=12287
610 FOR A=1 TO 21
620 READ D$
630 FOR B=0 TO 2
640 K=0
650 FOR C=1 TO 8
660 IF MID$(D$,B*8+C,1)="H" THEN K=K+2↑(
8-C)
670 NEXT C
680 P=P+1
690 POKE P,K
700 NEXT B
710 NEXT A
720 RETURN
```

Let's look a little bit at how the subroutine works to give you not all the details, but a general idea.

Line **600** sets **P** equal to the first of one series of **POKE** addresses used in sprite design. Line **610** starts a **FOR . . . TO . . .** and **NEXT** loop that the computer goes through 21 times—the vertical dimension of the sprite. Each time, at line **720**, it reads one **DATA** line and sets **D$** equal to what it finds on that line. Also, each time it goes through this loop it goes through a couple of smaller loops (beginning at line **630**). As it goes through these smaller loops, it analyzes the string **D$** it got from your **DATA** line, converts it into code numbers the computer can use, and pokes those numbers into the right addresses.

When the computer has gone through the larger loop (the one that ends at **710 NEXT A**) 21 times, it has finished putting your sprite design into memory. Then it goes on to line **720**, where it returns to the main program.



Now, let's actually draw the picture into these data lines. First, draw your design on graph paper. Remember, it will be quite small on the screen, and each box can be either full or empty—you cannot use particular characters. When you fill in your design in the **DATA** lines, however, you have to use some character to show you

want a particular place filled in. This program uses the letter **H** for that purpose.[4]

When you fill in the **DATA** lines with your design, you'll have an easier time if you avoid using the [INST/DEL] key. Use the cursor-arrow keys to your heart's delight—the quotes are already in and you are working on typed spaces. Type over any mistakes with a space.

Now, let's try drawing in a sprite. Soon enough, you will be making your own. This time, we have designed one for you. (This time, it won't be a ship! The Commodore goes on leave sometimes—perhaps to the ballet?)

LIST 1000–1200

First count the lines. Are there 21? Then count the blank spaces. Are there 24 between quotes in each line? Then type H characters into the blank space in your lines 1000 to 1200 so that it ends up looking like the following program. (You don't have to copy it exactly, as long as there are 21 lines total, each with only Hs in it, and exactly 24 spaces across within the quotes.)

```
1000 DATA "          HHH           "
1010 DATA "    HH    HHHHH         "
1020 DATA "    H     HHHHH         "
1030 DATA "    H       H           "
1040 DATA "    HHHHHHHHHHHHHHHHHHHHHHHH"
1050 DATA "          HHHHHHH       "
1060 DATA "           HHHH         "
1070 DATA "             HH         "
1080 DATA "             HH         "
1090 DATA "       HHHHHHHHHHHHHHH  "
1100 DATA "        HHHHHHHHHHHH    "
1110 DATA "         HHHHHHHHHH     "
1120 DATA "             HH         "
1130 DATA "             HH         "
1140 DATA "             HH         "
1150 DATA "             HH         "
1160 DATA "             HH         "
1170 DATA "             HH         "
1180 DATA "             HH         "
1190 DATA "             HH         "
1200 DATA "             HH         "
```

[4] If you prefer some other character, such as the shaded box— and [+]—or the diamond—[SHIFT] and [Z]—feel free to use it. You'll just be pressing two keys instead of one. If you do it, be sure to change the character within quotes in line **760**! Also, do not try to use reverse characters for this purpose. Remember, no matter what character you use here, the sprite comes out the same—each box is either full or empty. We picked **H** because it fills up a lot of the space and you don't have to use the [SHIFT] or key to get it.

When you are finished putting in the Hs (or along the way, when you are finished with each line), be sure to take the cursor and go down each line pressing [RETURN]—otherwise, your drawing will not be registered in memory, and you will have to do it again.

Type LIST 1000–1200 to see that everything you typed came out right.

Now, it should work.

Run your program.

It takes 10 seconds or more for anything to happen. If, after half a minute or so, it has not worked, stop it (with [RUN/STOP]). Then list your program, section by section, and check the typing in each.

Using the "Handy-Dandy Sprite Maker"

When you design your own sprites, copy the previous program exactly as it is typed, with the following options to suit your creativity:

1. You can change the color by putting a different color code number at the end of line **50**. Try white.

Type LIST 10–100 (Notice that the sprite stays on the screen. That doesn't hurt anything. If you want to get it off, use [RUN/STOP] and [RESTORE].)

Edit line 50 to read 50 POKE F+39,1

Type LIST 10–100

Run your program.

2. You can change where on the screen your sprite is located by setting **X** and **Y** in line **60** and **70** equal to whatever you want. (See the chart of the sprite screen layout on page 000.) Try putting your dancer near the top right of the screen.

Type LIST 10–100

Edit lines 60 and 70 as follows:

```
60 X=280
70 Y=60
```

Type LIST 10–100

```
10 PRINT"J"
20 F=53248
30 POKE 2040,192
40 GOSUB 600
```

```
50 POKE F+39,1
60 X=280
70 Y=60
80 GOSUB 400
90 POKE F+21,1
100 END
```

3. You can change the size of the sprite by adding a line or two. **POKE** address **F+23** makes it taller, **F+29** makes it fatter. The code number to poke in is **1**. Try it.

Type 85 POKE F+23,1

List lines 10 to 100, and then run the program.

Your dancer just got taller. Now, try fleshing her out a bit.

Type 86 POKE F+29,1

List lines 10 to 100.

```
10 PRINT"J"
20 F=53248
30 POKE 2040,192
40 GOSUB 600
50 POKE F+39,1
60 X=280
70 Y=60
80 GOSUB 400
85 POKE F+23,1
86 POKE F+29,1
90 POKE F+21,1
100 END
```

Run your program.

4. Most important, you can determine the actual design of your sprite by typing in whatever you want on the **DATA** lines **1000** through **1200**. Before you start designing your own sprites, though, let's use the one you have a little while to see what you can do with it.

Making Your Sprite Move

Sprite animation is a matter of systematically changing the **X** and **Y** values. First, let's get your sprite back to the size and place it was before.

Edit line 70 to read 70 Y=210

Delete lines 85 and 86.

Now, edit line 60 to read 60 FOR X=0 TO 350

Type the following lines:

```
95 FOR DL=1 TO 100
96 NEXT DL
98 NEXT X
```

List lines 10 to 100.

```
10 PRINT"□"
20 F=53248
30 POKE 2040,192
40 GOSUB 600
50 POKE F+39,1
60 FOR X=0 TO 350
70 Y=210
80 GOSUB 400
90 POKE F+21,1
95 FOR DL=1 TO 100
96 NEXT DL
98 NEXT X
100 END
```

These new lines set up a loop that systematically sets the value of **X** from **0** (which is offstage to the left) to **350** (which is offstage to the right). Lines **95** and **96** delay the action a little, so your ballerina's dance is not a dash! (You have to put the **NEXT** instruction all the way down at line **98** because line **90** turns on the sprite. If the **NEXT** instruction were before that line, the computer would carry out the animation instructions just fine, but you would not see any animation because the sprite would be "off.")

Run your program.

If your dancer's legs appear to shimmer a little, as if she's really dancing, that wasn't part of our animation. Thin vertical lines flicker a little on the screen. (On some screens, thin vertical lines barely show up at all.) In any case, it looks very nice on our TV!

When the dancer is across the screen, stop the program with [RUN/STOP].

Controlling Your Sprite from the Keyboard

You can also control your sprite's movements from the keyboard.

Retype line 60 to read 60 X=180

Delete lines 95, 96, and 98.

You will not need the **NEXT**, since the **FOR . . . TO . . .** line is gone (it was line **60**). Nor will you want the delay loop when you control the figure from the keyboard.

Now add the following lines:

```
92 GET LR
93 IF LR=0 THEN GOTO 92
94 IF LR=8 THEN X=X-1
95 IF LR=9 THEN X=X+1
100 GOTO 80
```

List lines 10 to 100.

This little addition should be familiar to you from the discussion of keyboard-controlled movement in the last chapter. Do you remember? **GET** is like **INPUT** except that it takes what you type directly, without your having to press [RETURN]. But it can only take a single key, and it goes right on without your pressing anything if you do not stop it with a line such as line **93**. Depending on whether you press the [8] or [9] key, lines **94** and **95** either subtract **1** to make **X** smaller (moving the sprite to the left one space) or add **1** to make **X** larger (moving the sprite to the right). Line **100** sends the computer back in the program so that it can poke in the fancy-numbered equivalents for the new **X** values. (The **END** instruction is no longer needed, since there is no way the computer can get past this **GOTO** line.)

Run the program, controlling movement with the [8] and [9] keys.

Making Two Sprites— *Pas de Deux*

Making a second sprite is easier than making the first. You edit a couple of lines and add 10 new ones, plus 21 new **DATA** lines with your design for the new sprite.

Edit lines 90 and 610 as shown:

```
90 POKE F+21,3
610 FOR A=1 TO 42
```

Type the following lines:

```
31 POKE 2041,193
51 POKE F+40,0
61 XS=110
71 YS=210
401 JS=XS
421 IF XS>255 THEN G=G+2
441 IF XS>255 THEN JS=XS-255
```

```
451 POKE F+2,JS
461 POKE F+3,YS
705 IF A=21 THEN P=P+1
```

We will add the 21 **DATA** lines in a moment. First, however, let's briefly look at what you've added.

Type LIST 10–90

```
10 PRINT"□"
20 F=53248
30 POKE 2040,192
31 POKE 2041,193
40 GOSUB 600
50 POKE F+39,1
51 POKE F+40,0
60 X=180
61 XS=110
70 Y=210
71 YS=210
80 GOSUB 400
90 POKE F+21,3
```

The key new lines are lines **31**, **51**, **61**, and **71**. These lines each do exactly the same thing for sprite #2 as the one right above each of them did for sprite #1. Thus, line **31** is a technical line that tells the computer where to find the sprite #2 design information in its memory. Copy this exactly as it is. Line **51** sets the color for sprite #2. You can put in any color code you wish here. In this example, we put in **0**, for black. Lines **61** and **71** set the starting location for sprite #2.

We have picked **XS** and **YS** to stand for the column and row numbers for sprite #2. You can position sprite #2 anywhere by setting these to be whatever you like. In the example, sprite #2 has the same row number as the sprite #1 so that they will be side by side, and a lower column number so that #2 will start out to the left of #1.

The important change to understand is the one in line **90**. The code **3** means turn on *both* the first and second sprites.

List lines 400 to 470.

```
400 J=X
401 JS=XS
410 G=0
420 IF X>255 THEN G=G+1
421 IF XS>255 THEN G=G+2
430 POKE F+16,G
440 IF X>255 THEN J=X-255
441 IF XS>255 THEN JS=XS-255
450 POKE F,J
451 POKE F+2,JS
```

```
460 POKE F+1,Y
461 POKE F+3,YS
470 RETURN
```

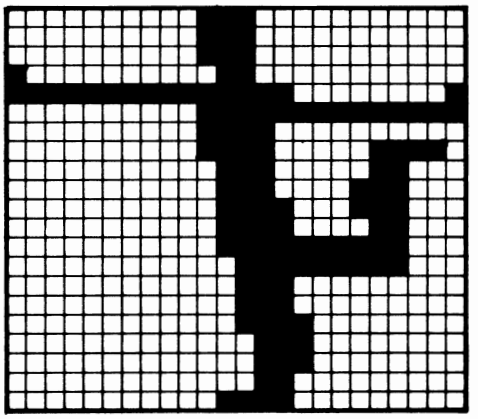
Your new lines **401**, **421**, **441**, **451**, **461**, and **705** and the edit of line **610** are just technicalities necessary to make the subroutine take into account that you are making two sprites.

Now, let's design your second sprite.

Type the following DATA lines. (Again, leave 24 spaces between the quote marks.)

```
2000 DATA "                        "
2010 DATA "                        "
2020 DATA "                        "
2030 DATA "                        "
2040 DATA "                        "
2050 DATA "                        "
2060 DATA "                        "
2070 DATA "                        "
2080 DATA "                        "
2090 DATA "                        "
2100 DATA "                        "
2110 DATA "                        "
2120 DATA "                        "
2130 DATA "                        "
2140 DATA "                        "
2150 DATA "                        "
2160 DATA "                        "
2170 DATA "                        "
2180 DATA "                        "
2190 DATA "                        "
2200 DATA "                        "
```

Design your second sprite in the blank spaces so that it looks like the following figure.



```
2000 DATA "           HHH                "
2010 DATA "           HHH                "
2020 DATA "           HHH                "
2030 DATA "H           HH                "
2040 DATA "HHHHHHHHHHHHHHH             H"
2050 DATA "           HHHHHHHHHHHHHHHH"
2060 DATA "           HHHH               "
2070 DATA "           HHHH        HHHH   "
2080 DATA "            HHH        HH     "
2090 DATA "            HHH       HHH     "
2100 DATA "            HHHH      HHH     "
2110 DATA "            HHHH       HH     "
2120 DATA "            HHHHHHHHHHHH      "
2130 DATA "             HHHHHHHHHH       "
2140 DATA "             HHH              "
2150 DATA "             HHH              "
2160 DATA "             HHHH             "
2170 DATA "              HHH             "
2180 DATA "              HHH             "
2190 DATA "              HH              "
2200 DATA "            HHHH              "
```

Be sure you have pressed [RETURN] on each line. Then list lines 2000 to 2200 and check your typing.

Now, run your program.

You should get two dancers, and you should be able to control one of them.

To control the new sprite, you will have to add a couple of lines:

Stop the program with [RUN/STOP] and [RESTORE].

Type the following lines:

```
96 IF LR=1 THEN XS=XS-1
97 IF LR=2 THEN XS=XS+1
```

List lines 10 to 100.

```
10 PRINT"J"
20 F=53248
30 POKE 2040,192
31 POKE 2041,193
40 GOSUB 600
50 POKE F+39,1
51 POKE F+40,0
60 X=130
61 XS=110
70 Y=210
71 YS=210
```

```
80 GOSUB 400
90 POKE F+21,3
92 GET LR
93 IF LR=0 THEN GOTO 92
94 IF LR=8 THEN X=X-1
95 IF LR=9 THEN X=X+1
96 IF LR=1 THEN XS=XS-1
97 IF LR=2 THEN XS=XS+1
98 GOTO 80
```

Now, run the program.

You should now be able to move both sprites using the [1] and [2] keys for the second sprite.

Incidentally, notice that sprite #2 always passes *behind* sprite #1. The higher-numbered sprites (according to the **POKE** addresses used to design them) always go behind the lower-numbered sprites. If you had drawn in a background design, all sprites would go in front of it.

Finally, if you want to make both sprites twice as tall, add this line: **POKE F+23,3**. To make them both twice as fat, the line would be **POKE F+29,3**.

To review, here's how to make two sprites:

1. Copy the two-sprite "Handy-Dandy Sprite Maker" program exactly.
2. Put in the desired color codes for the two sprites in lines **50** and **51**.
3. Put in the starting locations for the two sprites in lines **60**, **61**, **70**, and **71**.
4. Draw your own sprite designs into the two sets of **DATA** lines—**1000** to **1200** and **2000** to **2200**.

That's it. You've got two sprites!

Save the program you have just made on tape, and you will never again have to type anything but color, starting screen location, and your design to make sprites.

There is a listing of this two-sprite program in Appendix G. It is the same as what you have just done except that it includes program lines to permit right-left *and* up-down movement of both sprites.

Making Three or More Sprites—Your *Corps de Ballet*

You can make up to eight sprites at once. Each new sprite requires editing the same 2 lines you did before, and adding 10 new lines and 21 **DATA** lines.

The rules for the lines you change and edit are given in Appendix C. The key lines are the same as for the second sprite—the lines that set color and location and, most important, the **DATA** lines with your original design.

In this chapter, you learned how to make sprites and how to make them do what you want them to. It may take a while to master it, but it's not really very difficult—just copy a long list of program lines, set your sprite's color and initial location, and then make the design. If you saved on tape the program you used in this chapter, the hardest part—typing all those program lines—is done already.

In the next chapter, we will conclude our discussion of computer "show business" with a study of sound on the Commodore 64. In the next chapter, the Commodore goes to a concert!

Summary

This chapter explains how to make and manipulate the Commodore 64 sprites.

A *sprite* is a small design you can program into a special section of the computer's memory and then easily animate and move around. The actual design is usually about the size of three regular characters across and three lines down and is made up of a pattern of dots 24 across by 21 down.

Producing a sprite takes five steps:

1. Poking a specific address and code to mark where the sprite will be in memory.
2. Poking a screen color code number into a particular address to determine the sprite's color.
3. Poking a code number into a specific address to turn the sprite "on."
4. Poking code numbers into two addresses to determine where the sprite should go on the screen horizontally and vertically.
5. Poking 63 addresses with special code numbers that tell the computer your actual sprite design.

Many of the details of this procedure, particularly steps 4 and 5, are fairly complicated and require techniques beyond the scope of this book. However, so that you can use sprites right away, we have provided the "Handy-Dandy Sprite Maker" program. Copy it exactly from the book, then put in the color code in the appropriate place and the column and row locations for where to start the sprite. Then just draw the sprite on the screen (within blank places in the special **DATA** lines).

By adding a line with a particular **POKE** address and code number, you can make your sprite twice as tall. Another address and code make it twice as fat.

The procedures you learned in Chapter 13 for animation and keyboard-controlled movement of an ordinary design also apply to sprites. By putting the numbers for the sprite's row or column location in a loop, you can animate it. Or you can systematically add or subtract from the column or row number, according to what keys are pressed. This requires using **GET**, along with **IF . . . THEN . . .** instructions to evaluate what was "gotten."

You can make more than one sprite by adding 10 lines and editing 2, plus adding your design. The key new lines, besides the design, are those that determine the new sprite's color and starting location. Once you save the two-sprite version of the program on tape, you can make two new sprites by simply changing the sprite-design **DATA** lines and, if you like, the lines for their color and original location.

With two or more sprites, the ones with the lower-numbered lines always pass "in front" of the ones with the higher-numbered lines.

Making more than two sprites involves doing the same thing you did for the second sprite for each new sprite. That is, for each additional sprite add 10 new program lines, edit 2 old ones, and put in the 21 new **DATA** lines with the sprite's design.

Terms and Concepts Introduced in Chapter 14

Sprite

Using the "Handy-Dandy Sprite Maker" to set color, location, and design

Making sprites larger

Animation and keyboard-controlled movement of sprites

Practice Exercises

(Answers and comments are in Appendix A.)

1. Use the "Handy-Dandy Sprite Maker" to produce two original sprites of your own design. (If you really can't think of any designs, how about boats, fish, seagulls, dancing sailors—the Commodore likes them all.)

2. Add program lines that make your sprites twice as tall and twice as fat.

PART III CHAPTER 15

Sound and Music

The Commodore 64 has been silent long enough. It's time for it to speak up—or at least make some noise.

Actually, the Commodore 64 sound generator is an acoustic armada. With it, you can reproduce just about any sound from a boat whistle to a cannon firing, from Beach Boys music to Mozart, from the mandolin to the saxophone. There is almost no limit.

To make use of this luscious wonder of modern technology, you must understand a little about sound.

Basically, any sound has four aspects:

1. Its *volume* or loudness.
2. Its *pitch* or "wave frequency"—which we experience as its "high" or "low" quality.
3. Its *duration*—how long it lasts.
4. Its *quality*—a catchall term for other aspects of a sound wave that create the differences between the same note played on a trumpet and a xylophone.

In this chapter, you will learn how to use the Commodore 64 sound generator to control each of these aspects of sound. In fact, every time you produce a sound, you must give the computer a setting for all four of these aspects.

This chapter first presents a short program for a particular sound, then systematically considers how to vary each aspect of the sound to produce different effects.

A Simple Program for a Single Tone

Here is a program for a sound that approximates playing middle C on a piano.

Clear the computer's memory (with NEW) and type the following lines:

```
10 B=54272
20 POKEB+24,15
30 POKE B+5,9
40 POKE B+6,0
50 F=262
60 POKE B,F/.06097-(256*INT(F/15.6083))
70 POKE B+1,F/15.6083
80 POKE B+4,33
90 FOR D=1 TO 1000
100 NEXT D
110 POKE B+4,32
```

B = Beginning of **POKE** addresses for sound

Sets volume to maximum

Sets some components of quality

F = Frequency; **262** is one octave below middle C

These two lines convert frequency to pitch code numbers and poke them into appropriate addresses

Component of quality; turns sound on

Delay loop to set duration

Turns off note

List your program.

We will now go over the role of each line in this program. Since we will also be discussing the new ideas you encounter here throughout this chapter, be patient about understanding it all right away.

Line **10** sets **B** equal to the **POKE** number that prods the part of memory that produces sound on the Commodore 64. Thus, all the other **POKE** numbers you need are arrived at by making small additions to this first number.

Line **20** sets the volume (aspect #1) at **15.**

Lines **30** and **40** have to do with the quality of the sound (aspect #4).

Lines **50**, **60**, and **70** set the pitch (aspect #2), which in this case will be one octave below middle C.

Line **80** involves a **POKE** address (**B+4**) that further determines the quality of the sound (aspect #4 again). It is also responsible for starting and stopping the tone. In this case, line **80** is starting the tone. After the delay loop of lines **90** and **100** (which determines aspect #3, duration), line **110** pokes **B+4** again, with a code number that stops the tone.

Okay, here goes!

Run your program.

You should hear something similar to a piano playing one octave below middle C. (If not, be sure your TV volume isn't turned down all the way, and try again.)

Setting the Volume of a Sound

POKE address **54296** controls the volume. Since it is **24** above the beginning **POKE** address for sound, you wrote **POKE B+24**. Volume settings go from **0** (off) to **15** (loudest).

Usually, you set the volume at **15** and use the TV volume control to adjust it to your ear's comfort. You may, however, want to change the volume in the middle of a tone (to create special effects), or in the middle of a series of tones (as when you want part of a melody to be softer or louder). In these cases, just poke in a new volume at the appropriate point in the program.

Pitch—How High or Low You Want the Sound

The pitch of a tone is how high or low it is. For example, a flute is pitched higher than a tuba. For those who play music, pitch has to do with the note and the octave—for example, middle C is a lower pitch than high C.

Pitch is measured by the *frequency* of the sound wave—how many vibrations it makes per second. It is not important to understand exactly what this means. All you need to know is that the higher the frequency, the higher the pitch. The frequency of middle C is 523. The table that follows gives the frequency for two octaves of the musical scale.[1] You can also use frequencies that fall between the standard musical notes. Sound effects, for example, certainly do not have to be the exact frequencies of musical notes. Use any frequency from 0 to 3995, but remember—the human ear cannot hear a frequency below 20!

NOTE	FREQUENCY (octave ending before middle C)	FREQUENCY (octave beginning with middle C)
C	262	523
C#	277	554
D	294	587
D#	311	622
E	330	659
F	349	698
F#	370	740
G	392	784
G#	415	831
A	440	880
A#	466	924
B	494	988

[1]The *Commodore 64 User's Guide* gives tables of the actual code numbers to poke in to get each musical note. On the surface, using the code numbers directly would seem simpler than writing that seemingly incomprehensible conversion arithmetic we provided in lines **60** and **70**. But we recommend doing it our way for two reasons. First, once you have written the program, there is only one number to change—frequency—for the pitch. Second, frequency means something—high frequency is high pitch; low frequency is low pitch. The logic of the two code numbers is not intuitively meaningful. You must always look them up to work with them. If, however, you do use Commodore's table of code numbers, be sure to use the longer one in the *User's Guide* Appendix.

Lines **60** and **70** turn the frequency specified at line **50** into two code numbers because, to produce a particular frequency, the Commodore 64 sound generator requires two different addresses to be poked: **54272** (**B**) and **54273** (**B+1**).The formulas for converting the specified frequency to those two code numbers are complicated. We have written those formulas right into the program in lines **60** and **70**. As with sprites, it is not necessary to understand the mathematics of this conversion. Just copy in the lines as shown and let the computer do the hard stuff for you.

Run your program (the same one you ran before).

That was C again.

Edit line 50 to read 50 F=523

List your program.

```
10 B=54272
20 POKEB+24,15
30 POKE B+5,9
40 POKE B+6,0
50 F=523
60 POKE B,F/.06097-(256*INT(F/15.6083))
70 POKE B+1,F/15.6083
80 POKE B+4,33
90 FOR D=1 TO 1000
100 NEXT D
110 POKE B+4,32
```

Run your program.

That was one octave higher—middle C.

Let's modify the program so that you can try several different notes.

Type the following lines:

```
45 PRINT "TYPE FREQUENCY"
50 INPUT F
120 GOTO 45
```

List your program.

```
10 B=54272
20 POKEB+24,15
30 POKE B+5,9
40 POKE B+6,0
45 PRINT "TYPE FREQUENCY"
50 INPUT F
60 POKE B,F/.06097-(256*INT(F/15.6083))
70 POKE B+1,F/15.6083
80 POKE B+4,33
90 FOR D=1 TO 1000
```

```
100 NEXT D
110 POKE B+4,32
120 GOTO 45
```

Lines **45** and **50** are a prompt-and-**INPUT** sequence in which the frequency you type is set equal to **F**. Then, as usual, lines **60** and **70** convert **F** to the appropriate code numbers and poke them into the pitch-control addresses. Line**120** sends the computer back to **45** so that you can give it a new note to play.

Run your program. When the prompt appears the first time, try C (262). The next time, try middle C again (523). Then try several more from the table, and some frequencies not in the table (anything from 0 to 3995), until you have a feel for how this works and just what we mean by pitch. Be sure to try some very high and very low frequencies to see the tremendous range of tones available on the Commodore 64.

When you are done, stop the program with the [RUN/STOP] and [RESTORE] keys.

It's almost a musical keyboard, isn't it?—except that you are typing three digits for each note. Obviously, it could become a keyboard. First, however, let's explore more of the multitude of variations possible in a single note.

Duration—How Long a Sound Lasts

A buzzer can go on for hours; the sound of a pencil tap lasts a fraction of a second. In music, the length of a note is very important (for instance, whether it is a quarter note or a half note). The duration of a tone on the computer is determined in the same way most durations are determined on a computer—with a delay loop.

In the program we have been using, lines **90** and **100** make the computer go through this loop 1000 times before going on to the next instruction. Since line **80** turns "on" the tone and line **110** turns it "off," the loop between the two lines determines how long the tone will be heard.

You can adjust the length to be as long or as short as you like by adjusting how many times the computer goes through the delay loop. For example, you could set the loop to **2000** for a whole note, **1000** for a half note, **500** for a quarter note, and so on. This would be a fairly slow tempo. **1800** for a whole note is more of an average tempo. Rates as fast as **1000** for whole notes are not unusual.

The Quality of a Sound

A violin, flute, steam-engine whistle, or the human voice could all create the same note. They could all be exactly the same loudness, the same pitch, and last the same amount of time. Yet they would still sound very different. That difference is what we mean by the quality of a sound.

The quality of a sound has several components, which combine with each other (and with volume, pitch, and duration) to produce all the different sounds you hear. The variety of sound quality that can be produced by your Commodore 64 is stupendous.

To explain thoroughly how all the components of sound quality combine, we would need to discuss a lot of physics. Even without physics, however, we can still give you some sense of the possibilities. If you are interested in more details, see Box 15-1.

One component of sound quality you can vary is the sound wave's shape (the *wave pattern* or *wave form*). The **POKE** address that sets this is **54276** (**B+4**). As we mentioned earlier, this **POKE** address is also responsible for turning the tone "on" and "off." Thus, in the program we are using, with one code number line **80** told the computer what kind of wave pattern to set up and it told it to start the tone. The **POKE** in line **110** turned the tone "off."

There are three main wave patterns for you to work with.[2] The one you have been using so far is called a *sawtooth wave*. Its code numbers are **33** for "on" and **32** for "off." In combination with other settings (which we will discuss shortly), this produces a piano-like sound.

Another possibility is what is called a *triangle wave*. In combination with other settings, it produces a sound like a department store elevator bell. Its code is **17** for "on" and **16** for "off."

Edit the following lines as shown:

```
80 POKE B+4,17
110 POKE B+4,16
```

List and run your program. Put in middle C (523) first, then whatever you like.

The third wave-pattern possibility is called *white noise*. It is very useful for making various sound effects and for simulating percussion instruments. Its code is **129** for "on" and **128** for "off."

Edit the following lines as shown:

```
80 POKE B+4,129
110 POKE B+4,128
```

[2]Actually, there are four, but one of them, the pulse or flat wave setting, is not much use without two other **POKE** addresses, each with special codes you would have to learn to use. Thus, it's beyond what we can cover in an introductory book.

List your program.

```
10 B=54272
20 POKEB+24,15
30 POKE B+5,9
40 POKE B+6,0
45 PRINT "TYPE FREQUENCY"
50 INPUT F
60 POKE B,F/.06097-(256*INT(F/15.6083))
70 POKE B+1,F/15.6083
80 POKE B+4,129
90 FOR D=1 TO 1000
100 NEXT D
110 POKE B+4,128
120 GOTO 45
```

Run your program. Brace yourself, then try middle C (523) again.

Another whole component of the quality of a sound is how sharply it rises to and settles from its maximum volume. In other words, a sound starts out being silent; then it gets louder (a factor controlled by what is called *attack rate*), settles a little (controlled by *decay rate),* stays at that level (controlled by *sustain level),* and finally drops to silence (controlled by *release rate*).

This stuff is, frankly, rather complex. We have put further details in Appendix H, where you can, if you wish, study and become an expert at controlling sound quality.

Or you can skip the appendix and just play around with the program that follows to get the feeling by trial and error for the effects of poking in different code numbers. You can write down what different combinations sound like—a shoe falling, a cricket, your kindergarten teacher's whistle for milk time—and use that as a guide for your sound making.

Now, let's do the actual poking. The Commodore 64 has two **POKE** addresses that determine four elements of this particular component of the quality of sound (which, to review where we are at, is but one aspect of sound!). Attack and decay are controlled by **POKE** address **54277** (**B+5**). Sustain and release are controlled by **POKE** address **54278** (**B+6**).

First, let's try these different settings.

Edit the following lines as shown:

```
30 POKE B+5,14
80 POKE B+4,33
110 POKE B+4,32
```

List your program.

Run your program. When it calls for notes, try some fairly low and some fairly high ones.

This should produce various whistles and horns.

Now try other settings (between 0 and 255) for lines 30 and 40 and see what you get. Make note of what some of them sound like for future reference.

So far, you have varied one or two things at a time. Like a good recipe, combinations of all these parts of sound produce much more than the sum of those parts. Ideally, you would listen to every possible combination of sound qualities, durations, pitches, and volumes—then remember them and use them. But even if you used the same note, duration, and volume, and varied only the quality—if you listened for one second to every sound you can get from the different attack/decay, sustain/release, and wave pattern settings, it would take about 50 hours to hear all the possibilities![3]

In Appendix H is a program that produces those settings randomly. Even without absorbing the whole box, you can use that program.

Now, you have learned how to control the four aspects of sound—volume, duration, pitch, and quality—on the Commodore 64. Shall we celebrate with a little music?

Putting a Song into Your Program

To play a song, you need only tell the computer which notes to play and what length of time to play each note. Of course, you must select appropriate quality settings—whatever makes sense for your song. Obviously, you wouldn't pick kazoo sounds for a Beethoven sonata (or maybe you would). For now, we will stick with the piano, focusing on how to program notes and their duration.

Edit your existing program so that it comes out as follows. (This editing will delete a number of lines.)

```
10 B=54272
20 POKE B+24,15
30 POKE B+5,9
40 POKE B+6,0
50 READ F,T
60 POKE B,F/.06097-(256*INT(F/15.6083))
70 POKE B+1,F/15.6083
80 POKE B+4,33
90 FOR D=1 TO 1000/T
100 NEXT D
110 POKE B+4,32
120 GOTO 50
```

B = Beginning of sound addresses
Sets volume to maximum
Sets attack/decay
Sets sustain/release
Gets frequency (**F**) and time indicator (**T**) from **DATA** lines
Converts **F** to code numbers and pokes them in to set pitch
Sets sawtooth wave and starts tone
Delay loop
Stops tone
Goes back to read numbers for next note

[3]And again, there are some sound settings on the Commodore 64 we have not even considered. If you are fascinated and want more information, see the *Commodore 64 Programmer's Reference Guide.*

List your program.

To review: Line **10** sets **B** equal to the first of the **POKE** addresses used for generating sound. Line **20** sets the volume to the maximum (**15**).

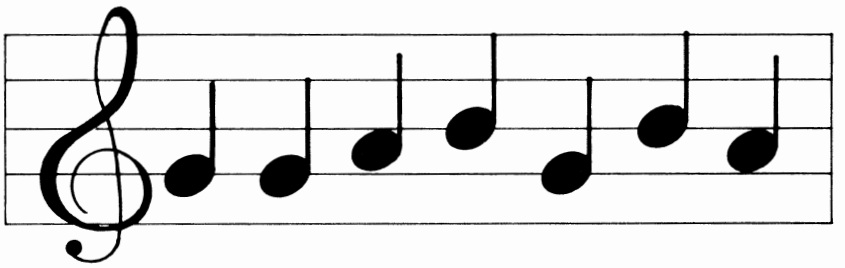
Lines **30** and **40** set the attack/decay and sustain/release settings (part of sound quality) to a piano-like sound.

Line **50** tells the computer to read from **DATA** lines (that you have not typed yet) the information to be used for setting **F** and **T**. Lines **60** and **70** are the technical lines that convert **F** to code numbers that can be poked in to produce the appropriate frequency sound.

Line **80** starts the note and sets a sawtooth wave pattern. Lines **90** and **100** are a delay loop. How long each note plays depends on how many times the computer must go through this loop. In this case, it must go through the loop **1000/T** times. (**T** is read for each note, along with the frequency, from the **DATA** lines.) For example, if **T** equals **2**, the computer goes through the loop 500 times (that is, **1000/2**). 1000 trips through the loop is about the time for one *measure* for a sprightly tune. Thus, for a whole note (which lasts a full measure), you would use **1**. **1000/1** is 1000. For a quarter note, you would use **4. 1000/4** is 250, or a quarter of a measure.

Line **110** turns off the note. Line **120** sends the computer back to line **50**, where it reads two more **DATA** values (for the pitch and time of the next note).

For the music, you have to put in two numbers for each note you want played: one number—the frequency—for the pitch, and the second number for the length of time each note will play. As we've described, we will use **1** for a whole note, **2** for a half note, **4** for a quarter note.



If you read music, you'll recognize this series of quarter notes, G–G–A–B–G–B–A. If you don't read music, don't worry. When you want to make your computer sing, just get someone who does read music to convert the notes into a series of letters (as we did above) and indicate the time for each (whether it is a quarter note, half note, whole note, and so on).

Box 15-1
Using Your Computer as a Piano

The following program makes pressing the number keys [1] through [8] produce the notes C–D–E–F–G–A–B–C. Then it allows you to press those keys to play tunes as long as you want. In other words, it turns your computer keyboard into a piano keyboard (or valves, or strings, or whatever your favorite instrument is—as you choose the sound quality, etc.).

If you have been working on the sound program in this chapter, save it and come back to it later. Let's edit the program and add some new lines. Edit and add lines as follows:

```
10 B=54272
20 POKEB+24,15
30 POKE B+5,9
40 POKE B+6,0
45 FOR K=1 TO 8
50 READ F(K)
54 NEXT K
55 GET N
56 IF N=0 THEN GOTO 55
57 POKE B+4,32
60 POKE B,F(N)/.06097-(256*INT(F(N)/15.6083))
70 POKE B+1,F(N)/15.6083
80 POKE B+4,33
90 GOTO 55
500 DATA 262,294,330,349
510 DATA 392,440,494,523
```

Line **10** sets **B** equal to the beginning of sound **POKE** addresses.

Line **20** sets volume to maximum.

Lines **30** and **40** set attack/decay and sustain/release to a piano-like sound. (You may change these if you wish.)

Type the following lines:

```
500 DATA 392,4,392,4,440,4
510 DATA 494,4,392,4,494,4,440,4
```

List your program.

```
10 B=54272
20 POKE B+24,15
30 POKE B+5,9
40 POKE B+6,0
50 READ F,T
60 POKE B,F/.06097-(256*INT(F/15.6083))
70 POKE B+1,F/15.6083
80 POKE B+4,33
90 FOR D=1 TO 1000/T
100 NEXT D
110 POKE B+4,32
120 GOTO 50
500 DATA 392,4,392,4,440,4
510 DATA 494,4,392,4,494,4,440,4
```

This is the full program for this little bit of a tune. Each time the computer comes to line **50**, it reads two numbers from the **DATA** lines. The first time, it reads **392,4**. **F** equals **392** and **T** equals **4**. Lines **60** and **70** poke in the codes to make a G note. Because **T** equals **4**, the number of repetitions of the delay loop beginning at line **90** is **1000/4**—that is, 250—which makes a quarter note.

The second time through the loop, the computer reads **392,4** again, since there are two G quarter notes in a row. The third time, it reads **440,4**—the numbers for an A quarter note.

After the last note has been played, the program sends the computer to line **50** again. It tries to read more numbers. Since there are no more numbers in the **DATA** lines, the computer stops the program and gives you the error message **?OUT OF DATA ERROR IN 50**. That's all right—you were finished, anyway.

Run the program.

Did you recognize those few notes of "Yankee Doodle"?

Save this program.

From now on, whenever you want the computer to play a song, just load this program from the tape and put in the **DATA** lines for the notes you want.

There are still more possibilities. You can make your computer keyboard into an eight-key piano if you read Box 15-1. Another possibility is to make several tones at once, as with chords. For that, see Box 15-2.

This chapter completes Part III of this book. We must admit, the last two chapters are a bit of a challenge to the old gray mat-

Lines **45**, **50**, and **54** read frequencies for **8** notes (from the **DATA** lines). Subscript **1** will be used for the frequency of C, subscript **2** for the frequency of D, etc.

Line **55** sets the key you press equal to **N**.

Line **57** turns the previous note "off."

Lines **60** and **70** convert **F(N)**—the frequency corresponding to the note read in line **50**—to two codes and **POKE** them in.

Line **80** turns on the note.

Line **90** goes back to get the next key press.

The **DATA** lines **500** and **510** hold the frequencies for one octave of the C scale.

List your program and check your typing.

This program makes your number keys into a one-octave piano.

Run your program, and start making music! If you don't know any songs, try the following (you may have to play with the rhythm a little to figure it out):

<4>-<6>-<6>-<7>-<6>-<4>-<2>-<3>-<4>-<4>-<3>

<2>-<3>-<2>-<3>-<4>-<6>-<6>-<7>-<6>-<4>-<2>

<3>-<4>-<4>-<3>-<3>-<2>

Perhaps you can figure out the rest on your own.

ter. But aren't you glad you joined us on this little exploration? You have now learned the fundamentals of using your Commodore 64 to make graphics and sound. You have also made a very good start in an area where some folks are earning millions—programming video games and fancy graphics for business and education.

The next chapter begins the final leg of your maiden voyage with the Commodore. You will learn how to write original programs and then, in the chapters that follow, how to uncover errors and how to use and modify programs from books and magazines (even if they are not written for the Commodore 64). Finally, we will discuss programming as an art.

Box 15-2
Making a Chord or a Chorus—Producing More than One Sound at a Time

So far, you have made only one sound at a time. The Commodore 64 has three separate voices, so you can make three different sounds at once. The volume has to be the same for all three, but the pitch, quality, and duration can be different. This expands the possibilities for sound effects even further. It also makes better music, since most interesting music involves playing notes simultaneously.

To use the second and third voices, follow the same procedures as for voice #1. Only the **POKE** addresses are different. The codes you poke into them are the same.

***POKE* ADDRESSES FOR SOUND**

VOICE 1

PITCH-FIRST	54272 (B)
PITCH-SECOND	54273 (B + 1)
WAVEFORM-ON/OFF	54276 (B + 4)
ATTACK/DECAY	54277 (B + 5)
SUSTAIN/RELEASE	54278 (B + 6)

Summary

In this chapter, you learn to use the sophisticated Commodore 64 sound generator.

A sound has four main characteristics: its volume, pitch, duration, and quality. Each of these can be controlled by the Commodore 64 sound generator, using various **POKE** addresses and codes.

A program that produces sound usually begins by setting a variable, such as **B,** equal to the beginning of the **POKE** addresses involved in creating sound, which is **54272**.

Volume (loudness) is controlled by **POKE** address **54296** (which would be **B + 24**). It ranges from **Ø** (silent) to **15** (loudest). Usually, you set it at **15** and then use your TV volume control for further adjustments.

Pitch (technically, *frequency*) is how high or low the tone is. Two addresses must be poked to set the pitch: **54272 (B)** and **54273 (B + 1)**. The formulas that convert frequency to codes that the Commodore 64 sound generator can use are included in the program lines in the various sample programs throughout this chapter. It is not necessary to understand the arithmetic or the physics behind them to use them.

Set the duration of a tone (how long it lasts) with a delay loop. A full musical *measure* requires about **18ØØ** times through the loop; a quarter note would require **45Ø**, and so forth.

The quality of a sound is what makes a tone played on a trumpet different from the same tone played on a xylophone. With three main **POKE** addresses, you can produce a great variety of sound qualities. **POKE** address **54276 (B + 4)** sets the wave pattern at either of two different musical sounds *(triangle* or *sawtooth),* or a special-effect sound *(white noise).* This address also turns the tone "on" and "off." For the first wave pattern (triangle wave), use **POKE 54276,17** (**POKE B + 4,17**). To turn it "off," use **POKE 54276,16** (**POKE B + 4,16**). For a sawtooth wave, use the same **POKE** address with codes **33** and **32**. For the white-noise effect, use codes **129** and **128**.

POKE address **54277** (**B+5**) sets *attack* and *decay* rates, two further components of sound quality. The attack rate is the time it takes for the tone to reach its maximum volume. The decay rate is the time it takes for the tone to settle down from peak volume to a more moderate volume. These components get combined into 255 different code number possibilities.

POKE address **54278** (**B+6**) sets *sustain level* and *release rate,* two more components of this one aspect of sound—its quality. The sustain level is the *part of the volume* where the tone is sustained *following the decay from peak volume.* The release rate is the length of time it takes for the tone to settle down to silence from the sustain level. These two components are also combined to produce 255 different possible code numbers and effects.

Writing a program that plays a song requires converting the musical notation to codes the computer can use. Once you know the notes and the duration of each note, you can use the table of frequencies (found earlier in this chapter) to set the pitch. The duration can be controlled by setting a variable such as **T** at **1** for a whole note, at **2** for a half note, at **4** for a quarter note, and so on. Then you write a delay loop that goes from, say, **1 TO 1600/T**. The program progressively reads the frequency and duration values from **DATA** lines and puts the converted frequency numbers into the appropriate **POKE** addresses and the number for duration into the delay loop.

VOICE 2

PITCH-FIRST	54279 (B+7)
PITCH-SECOND	54280 (B+8)
WAVEFORM-ON/OFF	54283 (B+11)
ATTACK/DECAY	54284 (B+12)
SUSTAIN/RELEASE	54285 (B+13)

VOICE 3

PITCH-FIRST	54286 (B+14)
PITCH-SECOND	54287 (B+15)
WAVEFORM-ON/OFF	54290 (B+18)
ATTACK/DECAY	54291 (B+19)
SUSTAIN/RELEASE	54292 (B+20)

Notice that the **POKE** addresses for Voice 2 are all exactly **7** higher than for Voice **1**, and those for Voice **3** are **7** higher still.

To write a program for several voices, type the instructions for one voice three times. It's a lot of program lines to type, but it's not hard—only a little tedious. Once you have typed it and saved it, you can use it from then on whenever you want to program "serious" music!

Terms and Concepts Introduced in Chapter 15

Four aspects of a sound and how to set each of them:

1. Volume
2. Pitch (or frequency)
3. Duration
4. Quality

Aspects of quality and setting each:

1. Wave pattern (sawtooth wave; triangle wave; white noise)
2. Attack rate
3. Decay rate
4. Sustain level
5. Release rate

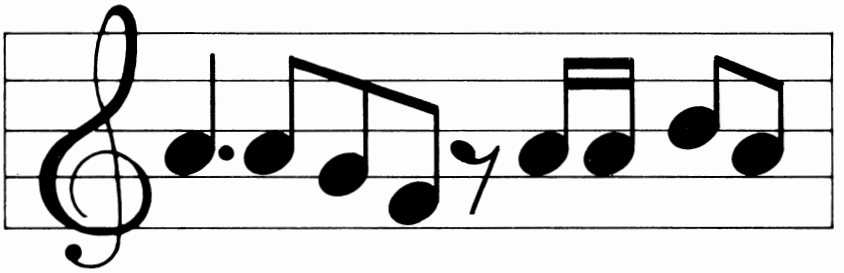
Making a tune

Practice Exercises

(Answers and comments are in Appendix A.)

1. Try to create sounds that approximate the following:
   a. A high-pitched whistle
   b. The G note on a piano
   c. A door slamming
2. Write a program that plays the following notes.

Use **1600** for the length of a measure. (Using the program in the chapter, set **T** equal to **4** for a quarter note, **8** for an eighth note, **16** for a sixteenth note, **2.7** for the "dotted quarter note; for the "rest," use a frequency that is too low to hear, such as **0** or **1.)**



A (dotted 1/4)	A (1/8)	G (1/8)	F (1/8)	rest (1/8)
A (1/16)	A (1/16)	B (1/8)	A (1/8)	

PART IV
Putting It All Together

PART IV CHAPTER 16

Writing Your Own Programs

This chapter examines the steps involved in taking an idea and making it into a program—charting your own course for the fleet.

You have now been introduced to the major tools of programming your Commodore 64. The next step is mastering those tools. As you do so, you will want to begin using those tools to write original programs.

The process of going from a vague idea to a workable program is one of going from the general to the specific. You move in steps, with each step getting a little more explicit, until you have it.

Contrary to the popular image of computers as rigid, demanding, anti-artistic, anti-creative, sour, and grumpy machines, *programming a computer is a creative activity*. There is no "cookbook" that tells you exactly what to do at each step. If there were, your program would not be original. What we can do, though, is give you some general directions about how to move from idea to program—a guide rather than a recipe.

Basically, there are *seven* steps. You do not *have* to follow them. We won't be watching and we won't be insulted. Maybe you can do without them. But they often save time in the long run because they help you to organize your approach to prevent big or totally incomprehensible problems. Now to the "mighty seven":

1. Come up with an idea of something you want to program.
2. Make the general idea explicit.
3. Make an outline of the major parts of the program.
4. Make a more detailed outline and diagram of how it all fits together.
5. Write the program lines and check for correct use of programming rules.
6. Make the program actually work on the computer.
7. Enjoy the benefits.

Let's consider these one at a time. But first. . . .

Don't Reinvent the Wheel—See If the Program You Want Has Already Been Written

Books, computer magazines, friends who own computers, fellow Commodore 64 owners belonging to a users' group, and, of course, computer stores all have programs that do just about anything you can think of. So when you want to program something, start with a little research through some magazines and books. Save your time with the Commodore 64 for really creative purposes. Because, of course, not *everything* has been done before.

Even if it has been done, if you have to spend $50 to buy a program you could write in an hour, or take a day searching through old computer magazines for something you could do yourself in a half day—then of course, do it on your own. Remember, even professional computer programmers spend only a little of their time working on brand-new programs. Mostly, they are busy modifying and adapting existing programs. In Chapter 18, we will consider in detail how to find and adapt programs. Now, we will focus on the steps involved in writing your own programs for those few (but enjoyable) times you will need them.

Step 1: Getting an Idea

Not only are books and magazines the best source of programs to copy or adapt, but they are also one of the best sources of inspiration for original program ideas. They give you a sense of the possibilities and the approaches taken by others.

Of course, even better sources of direction are your own needs and desires, necessity being, as ever, the mother of invention. How could your work, hobbies, or pure fun be improved by a machine that can remember lots of information, manipulate it quickly, and make various kinds of printing, pictures, and sounds come from your TV screen?

If you do not have any particular ideas for programs you want to write, and if this book or other books do not suggest any ideas to you—fine, maybe the existing programs do everything a computer can do for you right now. That is the purpose of the existing programs—to meet your needs and fulfill your desires. Don't feel obliged to come up with a new idea. But if and when you do have an inspiration that will be useful or fun to program, the next step is. . . .

Step 2: Making Your Idea Explicit

As an idea grows into a program, it becomes more and more explicit, until it is finally a set of step-by-step instructions for your dear old uncreative but obedient-to-the-letter Commodore 64.

When you start to write your program, the odds are that you are not very clear about what you want it to do. So first, take your idea and *write it out,* in at least three sentences that spell out as much detail as possible. *This works.* It forces you to make decisions you did not even realize were involved.

Step 3: Write an Outline of the Major Program Parts—a Grand Overview

The next step is to make your idea *more* explicit. Figure out what are the major parts of your program and write a short paragraph (or outline) of just what *each part* does. For most programs, there are four main sections (like the three earlier parts of a program, except that input is now divided into two parts).

The four sections are:

1. *Permanent information.* Examples: formulas for converting feet to meters; information on coins in a coin collection; the details of the design of a sprite; the notes of a song you want played.
2. *Input that will be different each time the program is run—that is, "user input."* Examples: the number of feet to be converted to meters; the dollar amounts of checks and deposits in a checkbook-balancing program; answers to questions (posed in prompts) about which program option you want carried out (such as, do you want a listing of all coins, or just of a particular value); the responses of students to test questions in an educational program; directions from the keyboard that move a screen design in a game; key presses to be transformed into musical notes.
3. *Transformations, sorting, and so forth—what the computer does with the information.* (This is the hardest part to make explicit.) Examples: carrying out a conversion in a feet-to-meters program; adding deposits and subtracting checks in a checkbook program; going through all the customers in your accounts to see which buy product "Zilch" in region "Kumquat"; checking whether your answer to a question in an educational program is right or wrong; converting your key presses to meaningful numbers that will move a sprite or play a tune.

4. *The output.* Examples: the number of meters; your final checkbook balance; a table of coins; an indication whether you have produced the correct answer; the flag waving on a ship; a sprite dancing; a song that sounds as if it's being played on differently tuned file cabinets.

Organize your idea into these four sections. Either make your original paragraph more explicit or use outline form. For example, the feet-to-meters program you did in Chapter 7 might look like this:

1. PERMANENT INPUT: Feet-to-meters conversion formula.
2. USER INPUT: Number of feet to be converted.
3. COMPUTER TRANSFORMATIONS: Carry out conversion.
4. OUTPUT:
   a. Number of feet put in (so users can check that the computer got what they thought they'd given it).
   b. Result of conversion (number of meters).

A diagram for the checkbook program you did in that same chapter would be a little more complex:

1. PERMANENT INPUT: There is none.
2. USER INPUT:
   a. Original balance.
   b. Dollar amounts of all checks.
   c. Dollar amounts of all deposits.
   d. Indication that all information is entered.
3. COMPUTER TRANSFORMATIONS:
   a. Subtract each check from original balance.
   b. Add each deposit to original balance.
4. OUTPUT: Amount of new balance.

Step 4: More Detailed Outline and Diagram of Program

Now, and only now, can you start laying out the program itself—but no numbered program lines yet! It's time to make a *very* explicit outline, adding computer language to the English paragraphs or outline you just made.

It is usually best to start with the output (because you are probably clearest about it—it is the purpose of the program). Then work on the input (because you know what is available). Finally, figure out how the computer will get from input to output.

In each case, the best approach is to consider what tools are available to accomplish what you want. Then select the ones that seem appropriate and see how they fit into the plan. To make this easy, we have summarized those tools for you on this and the following pages. (If you skipped either Part II or Part III of this book, be prepared to ignore some of what follows.)

I. TOOLS MAINLY FOR OUTPUT

A. **PRINT**

1. Ordinary message displayed across the screen (for example, **NEW BALANCE=$**)
2. Tables (for example, the table of prices in Chapter 8)
3. Simple charts, graphs, and designs

B. **POKE**

1. Complex and color designs, graphs, and charts
2. Animated or flashing sequences
3. Sprites
4. Sound effects
5. Music

C. Other Associated Tools

1. Delay loops to keep messages or designs on screen
2. Clear screen

II. TOOLS MAINLY FOR PERMANENT INPUT

A. Assigning information to a variable (to store it in memory)

1. Important numbers to be used again and again (for example, **P=2.37** in the price table, or **F=53248**, the first **POKE** address for sprite making)
2. Conversion formulas (for example, **M=F*.3048** in the feet-to-meters program, or **L$=CHR$(L)** in the intuition test)
3. Definitions of codes for sprites, music, and so forth (for example, **X=220** to set sprite location)
4. Putting information into a long string to take slices out of as needed (for example, **A$="————FIRST SECOND-THIRD FOURTH"** in the string-slicing example in Chapter 11)

B. Putting information into **IF . . . THEN . . .** lines. This is usually done as a kind of conversion formula (for example, **IF N>12 THEN P=2.64** in price table)

C. **READ** and **DATA**

1. For information to be kept on file in database management (such as coins or customers), or for questions and answers in a flashcard program

2. For a series of numbers to control some activity (such as the codes for a series of musical notes or the shape of a design)

III. TOOLS FOR USER INPUT WHILE THE PROGRAM IS IN PROGRESS

A. Prompt-and-**INPUT** sequence

1. For taking in numbers (for example, checks and deposits in the checkbook program or the number of feet in the feet-to-meters program)
2. For taking in character string information (for example, your guess as to the French equivalent of an English word in the flashcard program)
3. For taking in a number to determine what program activity to carry out next (as in the checkbook program, in which you type a **3** to indicate "no more items" or in the coin collection program where you type in a number to indicate the kind of listing you want)

B. **GET**

1. To determine some immediate action (such as the direction to move in keyboard-controlled movement of a design or sprite, or as in the computer-as-piano program)
2. As an alternative to **INPUT** for determining what program activity to carry out next in response to a prompt

IV. TOOLS MAINLY FOR "DOING SOMETHING WITH IT"

The tools for turning input into output include just about everything you have learned in this book. It is possible, however, to summarize these tools under two major categories:

A. Tools that mainly transform variables

1. Setting a variable equal to some value (such as **X=22Ø**)
2. **IF . . . THEN . . .** (for example, **IF X=1 THEN B=B+AM**)
3. Various transformation instructions that are used with other instructions:
   a. **CHR$**
   b. **ASC**
   c. **INT**
   d. **RND(1)**
   e. **MID$**
   f. The various mathematical symbols: + − * / ↑
   g. Combining character strings with +

B. Tools that mainly control the order of how the program is carried out

1. Line numbers
2. **GOTO**
   a. To change the usual order of going through a program
   b. At the end of a program, to send the computer back to the beginning, or some earlier section, so that the main activity of the program will be repeated
   c. To make an endless loop to keep the program going so that the **READY** message and the cursor will not appear on the screen
   d. In conjunction with **IF...THEN...**
3. **FOR . . . TO . . . , STEP**, and **NEXT** loops
   a. To control how many times the computer goes through a loop, as in determining how many times some program activity will be carried out (for example, in the practice program in Chapter 6, it determines how many times **I CAN PROGRAM** is repeated on the screen)
   b. To make a variable progressively change value (for example, in the price table in Chapter 8, the variable in the loop stands for the number of items)
   c. As a delay loop
4. **GOSUB** and **RETURN**
   a. To perform an activity that will be called upon several times during the program
   b. To set apart a section of a program that would be confusing (to someone looking at the program) if it were mixed in with the main program (as in the sprite-making program, in which the information on the sprite's shape is set off in a subroutine)
5. **IF . . . THEN . . .**
   a. To decide where to go next in the program (for example, in the checkbook program **IF X=3 THEN GOTO 12Ø**)
   b. To check information against a criterion (as in the coin collection program or in the flashcard program)
   c. To see if some limit has been reached (as in checking if all information in the **DATA** lines has been read in the program to play a song)
6. **END**
   a. As a good habit for completing programs
   b. At the end of the main part of a program, to keep it from continuing into the subroutines that follow (for example, in the subroutine exercise in Chapter 11)
   c. With **IF . . . THEN . . .** when checking against input from a prompt and the user indicates to finish, or when checking against a **DATA** list (for example, in the design-making programs, **IF X<Ø THEN END**)

With this little guide, make a brief program plan, stating the main tools you will use.

Some programmers actually make diagrams (called *flow charts*). This can be helpful with a long, complicated program, but the procedures are beyond what we can cover in an introductory book. For shorter programs, we suspect you can figure out how to diagram them very nicely without more verbiage, if diagrams help you. For most people and for most programs, it is enough to make a detailed outline. For example, if you had been designing the checkbook program from scratch, you might have outlined it like this:

I. PERMANENT INPUT

None.

II. USER INPUT

A. Original balance—use a prompt-and-**INPUT** sequence

B. and C. Check and deposit amounts—use prompt-and-**INPUT** sequences to find out if the item is a check or a deposit, and then to find out the amount

D. Find out with a prompt-and-**INPUT** sequence when the last item is put in

III. COMPUTER TRANSFORMATIONS

A. and B. Add deposits and subtract checks from the starting balance

—**IF . . . THEN . . .** lines to see if each item is a deposit or a check, and then add or subtract it from the balance, accordingly

—**GOTO** loop to keep the program going through the whole sequence of events until it has added or subtracted all the deposits and checks

—**IF . . . THEN . . .** line to check if all the items are done yet; then, if so, going somewhere to print the final balance

IV. OUTPUT

Amount of new balance

A. Clear the screen

B. Print a few words explaining what the number (the balance) is, such as **NEW BALANCE=$**

C. Print the new balance

Remember, in working out the computer-language outline, begin with the output, then the input, and do the transformation part last!

Step 5: Writing the Program Lines and Checking for Correct Use of Programming Rules

Now, you write the program.

First, draft it out on paper, putting in the main program lines with lots of room between them. To the right of them, jot down some **English-language** notes from the outline you just wrote, along with some other details, to keep everything very clear for yourself. At this point, do not even think about line numbers, and just write "prompt" for prompts. For **GOTO** lines and other spots that need line numbers, put an arrow for now.

A first draft for the checkbook program might have looked like this:

prompt **INPUT B**	To get the starting balance.
prompt **INPUT X**	Clear screen of last prompt. Have user indicate if the next item is a check or deposit, or if finished. Tell user to put in **1** for deposit, **2** for check, **3** for no more items.
IF X = 3 **THEN**	Go down to end and print the final balance.
prompt **INPUT AM**	Clear screen from last prompt. Have user put in amount of check or deposit.
	IF a deposit (IF X = 1) THEN add to balance B = B + AM) **IF a check (IF WA = 2) THEN subtract from balance (B = B − AM)**
GOTO— **PRINT**	to get back up to next-item prompt final balance

Once you have the rough draft, go through it to think out the general logic. Does it make sense? Follow it through as if you were the computer. Then actually write out the program.

There are two main considerations in writing out a program:

1. Correct and complete use of programing rules
2. Clarity of what you have written

Regarding the first, remember the following:

a. Start out with any **DIM** lines and any setting of variables equal to **0** or **1** (if that is needed).

b. Put a **NEXT** for every **FOR . . . TO . . .** line.

c. Put a **RETURN** at the end of every subroutine.

d. Put a **THEN** in every **IF** line.

e. Check that all your variable names match up—it's easy to start with a variable called **F** for "first" and later remember it as **S** for "start.

f. Check that whenever you use **GOTO** or **GOSUB**, the line number it directs the computer to actually exists (and is the right one!).

g. Carefully follow all the rules about quotes and punctuation in **PRINT** commands.

h. Check that all your prompts are there and that they make sense.

Okay, your program follows the rules. If it is of any use at all, you will probably want to revise or expand it in the future. There is quite an art to writing programs that are easy to recall (or grasp the first time) just by looking at them. (Professional programmers focus on this, since programs they write are for others and are reviewed or co-written with other programmers and must be modifiable by someone else in the future when the programming needs a change.)

Obviously, with very short programs, such as the examples in this book, this is not crucial. But as programs become longer, comprehensibility becomes *very* important. The rules for making your program easy to follow are also easy to follow:

1. *Use lots of* **REM** *lines.*

Using **REM,** put a heading before each major part of the program, each subroutine, and any unusual or not obvious program lines. Put a few blank **REM** lines (for example, **210 REM** and **211 REM**) between major sections of the program. In a very long program, you might even use a few **REM** lines to put in a table of contents.

2. *Use* **REM** *lines to put a program title at the very start of the program.*

(Make it 16 characters or fewer so that you can use this title when saving.)

Also, put your name, the date you wrote (or most recently revised) the program, and any details that are not obvious about the purpose of the program and how it is to be used.

3. *Put major parts of the program in separate subroutines.*

If you look at a program written by a professional, the first few program lines (after all the introductory **REM** lines) are probably **GOSUB** instructions, each of which directs the computer to a subroutine that carries out a different main section of the program. This is especially important in long programs. (You used

this kind of procedure to some extent in the sprite-making programs.) A good rule to follow is that the maximum length of any part of the program or any subroutine should be no longer than the 22 program lines you can list on the screen at once.

4. *Stick to a standard procedure of organizing your programs.*

This way, even in the dark, you can figure that if that's a nose, this down here *must* be a mouth.

In general, it should go as follows:

1. **REM** headings
2. Any **DIM** lines

Then, within each subroutine or for the program as a whole if it is a short one:

3. Any variables that you are setting to a basic value
4. Any variables that need to be set to zero or one (or whatever)
5. Your prompt-and-**INPUT** sequences
6. The body of the program
7. Subroutines
8. **DATA** lines

5. *Set a variable equal to any number you will use repeatedly, or to a large number that starts a series.*

For example, in the price table, you set **P=2.37**. In the sprite-making program, you set **S** equal to the first **POKE** address of the sprite-making section of the Commodore 64.

6. *Use variable letters or names that have some relation to what they stand for.*

For example, use **D** for delay, **SM** for sum, and **F** for first.

7. *Keep the number of* **GOTO** *lines to a minimum.*

It is hard to avoid them completely in BASIC, but the fewer the better because they make programs hard to follow.

Step 6: Making the Program Work on the Computer

The next step is to type in the program and make it actually work. This is the most time-consuming step in writing a usable program, because your dear computer is so very picky about every

little rule. Every letter has to be perfect. In the next chapter, we will discuss what is called *debugging*—getting the bugs (mistakes) out of the program so that it works. Of course, the main tool in debugging is prevention.

Most important, proofread your program listing carefully. The vast majority of errors are simple typos! Be especially careful to look for missing or misplaced quote marks, semicolons, and commas. Also, anywhere in the program, it is very easy to type the wrong letter for a variable name, or the wrong number.

Next, with the program on the screen, run through the logic of it once again, checking that you followed all the programming rules we just gave you.

Then run it!

Frankly, if it works the first time, feel free to faint. Don't ever expect to have that kind of luck again.

If it doesn't work the first time, rest assured that your program is absolutely normal. In the next chapter, we will discuss how to fix mistakes.

Once you *have* gotten the program to go through to its conclusion without any errors, you are still not finished. A program that works does not necessarily give the right answer, in the right form. Does the table produced contain the right information? Does the tune produced sound like what you had imagined in the shower?

Sometimes, it's obvious that the program isn't doing what it's supposed to—when you get strange images on the screen or wildly wrong answers (16 feet can't equal 4500 meters!).

Sometimes, things look right and actually aren't. It is *very important* to try out some simple input (for example a small number of checks and deposits in the checkbook program) for which you already have the result, using your own brain.

Even then, there is no guarantee that the program will work with all other input. An educational program may work fine when it tells you that you're correct, but not work when you're wrong. Or it may translate all five-letter words but not six-letter words. These things happen!

Ideally, you should try a lot of different sets of information to test your program. The test information should get at all the possibilities—especially the extremes (very high and low numbers in a conversion program, all deposits or all checks in a checkbook program, a design at the very edge of the screen in a keyboard-controlled animation program, and so on).

Step 7: Enjoy the Benefits

Once you have the program working and it tests out well, be sure to make a copy on tape. If you don't have a tape or disk machine, write it down carefully. Make more than one copy.

Then enjoy using it. Share it with your friends. If it is really original and could be useful to a lot of people, send it to a computer magazine, or even try to get a commercial software company to market it for you.

The main thing is to applaud yourself for a job well done. Your program is part of a long and laudable human tradition of doing less and accomplishing more. Let's hear it for evolution!

In this chapter, we have considered the steps involved in writing an original program. The biggest step in the process, however, is what we will cover in the next chapter: "Debugging—Getting Rid of Mistakes."

Summary

This chapter outlines the steps involved in writing an original program.

Before embarking on an original program, check whether the program you need is already available in a book or magazine, or for sale as commercial software. Usually, a program exists that you can either adapt or use directly—but not always. Even when programs are available, writing your own may save money or time.

Writing a program involves going from a general idea to a highly specific set of instructions. There are seven steps:

1. Get an idea. Your needs or interests are the best place to start. Books and magazines are also good sources of inspiration. (If you don't have any ideas for original programs, that's not surprising, since programs have already been written for almost every imaginable purpose!)
2. Write down your idea on paper. Make it about a paragraph long. Be specific.
3. Write an overview of the major parts of the program, with a few sentences describing each. Do not use computer language yet. Either expand your paragraph from Step 2 or make an outline. Again, most programs have three—or better, four—main parts: input of permanent information; input of information for each time the program is run; transformation of information; and output.
4. Make a detailed outline listing the actual terms for the programming tools you will use for the various parts of your program.

   Start with the output, for which you can use mainly **PRINT** (for words and simple designs) and **POKE** (for screen graphics and sound). Next, work on the input.

   The main tools for input of permanent information are variables that equal numbers or strings, and the **READ**-and-**DATA** procedure. The main tools for getting information while the program is running are the prompt-and-input sequence and **GET.**

Finally, in order to transform input to output, you have all the tools you have learned in this book. These fall into two main categories: tools that transform the value of variables and tools that control the flow of the program.

5. Write the actual program lines. These must be correct and complete. Remember **DIM** and other lines that go at the outset. Type carefully. Be sure to follow all the standard procedural rules such as ending a **FOR . . . TO . . .** loop with **NEXT.**

   Your program should be written so that it's easy to understand just by looking at it. Put **REM** lines at the beginning to identify the program, its author, and date. Use **REM** lines to label and separate sections of the program. If it is a long program, separate major program sections into different subroutines. Follow a standard order for writing different parts of the program. Set variables equal to large numbers used repeatedly during the program. Use variable names that are meaningful. Minimize the use of **GOTO.**

6. Make your program work. The next chapter is about discovering and correcting problems, but you can prevent problems in advance. Especially watch for typos and obvious logic errors before trying the program. Once the program runs, be sure it is producing the right result. Test it. Use input for which you know the result and that tests the extremes of what the program might face.

7. Save your program. Congratulate yourself. Enjoy the benefits of your work.

Terms and Concepts Introduced in Chapter 16

Seven steps of writing a program

Practice Exercise

(Answer and comments in Appendix A.)

Here is a short paragraph describing a program idea (Step 2 on our list):

"I want a program that will take numbers and convert them to Morse code. I want to be able to just type in a number from **0** to **9.** (It would be okay to press [RETURN] afterward.) Then the screen should give me the correct dot and dash pattern."

Take this idea and follow Steps 3 to 7 to make this into a program. It can be done with only the tools learned in Chapters 1 to 8, but if you have read any of Chapters 9 to 15, you can do a more elegant job!

Here are the Morse-code conversions you will need. There is a pattern to these codes, but you don't have to pay any attention to that for this program.

Ø = – – – – –
1 = · – – – –
2 = · · – – –
3 = · · · – –
4 = · · · · –
5 = · · · · ·
6 = – · · · ·
7 = – – · · ·
8 = – – – · ·
9 = – – – – ·

PART IV CHAPTER 17

"Debugging"—Getting Rid of Mistakes

As we hinted in the last chapter, writing a program (or adapting a program from a book) is the easy, boring part. The fun begins when you try to make it work on your computer. The majority of time involved in programming is spent in finding and eliminating the *bugs*.

It really is fun—detective work of the highest order. Once you discover the culprit, justice is swift. Change a letter here or there, press [RETURN]—and the bug is squashed.

When a Program Stops Midway—?SYNTAX ERROR

If you try to run your program and it stops midway, your Commodore 64 helpfully tells you something such as **?SYNTAX ERROR IN 85**. Error messages begin with a question mark followed by two or three words of explanation, then **ERROR IN**, followed by the line number with the problem. Let's look at some typical error messages, beginning with **?SYNTAX**.

?SYNTAX is a catchall message. It appears when something about a program line violates some programming rule (but not one for which there is a special error message). Usually (as with most errors), it's due to a typographical error:

Clear the computer's memory (with NEW) and type 10
```
PRING "SOMETHING"
```
(This isn't a typo—we really want you to type PRING)
List your program.

Notice that, although the line is wrong, the computer accepts it as a program line and lists it with no objection.

It catches the error only when you try to run it.

Run the program.

The screen should show the following:

```
10 PRING "SOMETHING"
LIST

10 PRING "SOMETHING"
READY.
RUN

?SYNTAX  ERROR IN 10
READY.
```

Correct the error (using the editing procedure of moving the cursor over the G, typing in a T, and then pressing [RETURN]).

List to see that you fixed it properly.

Run the program.

It should work fine this time.

If you are in command mode—that is, giving commands without using line numbers—the computer gives the same kinds of error messages. Of course, it can't refer to a line number.

Type PRINT 'HELLO' and press [RETURN]. (The error is using single quotation marks.) The screen should show this line:

```
?SYNTAX  ERROR
```

Fix the error and press [RETURN].

This time, it should work.

Another common syntax error is not having the same number of right and left parentheses:

Clear the computer's memory and type
10 PRINT CHR$(65+INT(26*RND(1))

List the program.

We discussed lines like this one in Chapters 10 and 11. This line first picks a random number between **0** and **.9999999999**, multiples it by **26**, and drops off the decimals to give a random whole number from **0** to **25**. Adding **65** puts it from **65** to **90**, the character codes for numbers. **CHR$** converts the code to a letter.

But . . . count the parentheses. There are three left parentheses and two right parentheses—a computer no-no!

Run the program.

You should get **?SYNTAX ERROR IN 10**

Edit the line by adding the final right parenthesis (and pressing [RETURN] while still on the line).

Now, run the program a few times.

One other kind of typo to watch for—and it requires some sharp watching—is mixing up the numeral "**1**" and the letter "**l**," and the numeral "**Ø**" and the letter "**O**."

More Error Messages—Some Obvious and Some Not So Obvious

Some error messages are pretty blunt:

?DIVISION BY ZERO ERROR IN...
?NEXT WITHOUT FOR ERROR IN...
?RETURN WITHOUT GOSUB ERROR IN...
?CAN'T CONTINUE ERROR IN...

Other error messages are not quite so obvious. (Many involve things you would know about only if you read Chapters 9 to 15.)

?BAD SUBSCRIPT ERROR IN....

(Gives you shudders, huh?) In Chapter 9, we said that whenever a subscripted variable has more than 10 subscripts, you must have a **DIM** instruction for it. **?BAD SUBSCRIPT** simply means you used a subscripted variable with a subscript that is higher than the number of subscripts you dimensioned for (or the subscript is higher than **1Ø**, and you forgot to dimension at all).

?ILLEGAL DIRECT ERROR.

(Talk about rubbing it in—couldn't it leave off either "illegal" or "error"?) In Chapter 4, you learned how to give a direct command to the computer without program line numbers. We called it operating in command mode. It is also known as *direct mode.* Only certain kinds of instructions are allowed in command (direct) mode. **PRINT** is okay. So are **RUN, NEW, LIST, CONT, SAVE, LOAD, VERIFY, GOTO,** and **POKE**. But many other BASIC instructions are not allowed.

Type INPUT X and press [RETURN].

You should get **?ILLEGAL DIRECT ERROR**

This message does not give a line number. It can't. It only applies when you are trying to carry out a command that doesn't have a line number.

ILLEGAL QUANTITY ERROR IN...

Many instructions—such as **POKE** commands (covered in Chapters 12 to 15), or instructions involving the character code

(Chapter 11)—use numbers of a specified range. For example, the character code ranges from **0** to **255**. The two codes you poke into the different sound-quality addresses have the same range. (**0** to **255** is a very common range on computers.) The highest **POKE** address is **65535**. If you use a number outside of one of these ranges, you get this error message.

Clear the computer's memory and type 10 PRINT CHR$(300)

List and then run the program.

Now, use a number within the allowable range—try 200. It should work just fine.

OUT OF DATA ERROR IN...

You learned about **READ** and **DATA** in Chapter 9. This error occurs when you've run out of data for a **READ** instruction to read. Either there are no **DATA** lines at all, or the computer has already used up the information on the existing **DATA** lines.

Sometimes, you don't mind this error. If the program has completed everything it was supposed to do, it doesn't matter if it ends with an error message. But the program does end when it gets to this point. Once the computer gives an error message, it stops running the program. If the program has more to do, it's a problem.

Actually, you may have seen this particular message quite a bit when editing program lines you had listed. At the bottom of your listing, the computer says **READY**. After you edit a few program lines, it's easy to keep pressing [RETURN] to get to a clear line to relist or to run your program. Thus, you might press [RETURN] while on the line that says **READY**. If you do, the computer thinks you are commanding it to read the variable **Y**! Of course, there is no **DATA** line for this, so it tells you **?OUT OF DATA ERROR**. This doesn't hurt anything. Ignore it and continue what you are doing.

?REDIM'D ARRAY ERROR IN...

This is short for "redimensioned array." An array is a group of subscripted variables (Chapters 9 and 10). For any particular variable, you can only use a **DIM** instruction once in a program. If you accidentally dimension the same variable twice, you get this message. This usually happens when you have used a loop to go back to an earlier part of the program and included the **DIM** line by accident.

?STRING TOO LONG ERROR IN...

This occurs very rarely. A character string can be a maximum of 265 characters long. If you try to make one longer than that, you get this message. It's hard to make one longer than this,

since any program line can be only 80 spaces long, but it's possible. For example, you could add together (concatenate—a procedure discussed in Chapter 10) 10 string variables, each of which is, say, 70 characters long.

?TYPE MISMATCH ERROR IN...

This means you have defined a number as a string, or vice versa.

Clear the computer's memory and type the following lines:

```
10 A="WORD"
20 PRINT A
```

List and run the program.

Change A to A$ (Variables that stand for a string of characters that aren't numbers, such as words or letters, are designated by $ and are called string variables. See Chapter 10 for more explanation.)

Run the program again. This time, it should work.

Clear the computer's memory and type the following lines:

```
10 X$=70-10
20 PRINT X$
```

List and run the program. Then correct the error and run it again.

?UNDEF'D STATEMENT ERROR IN . . .

STATEMENT means "numbered program line." **UNDEF'D** is short for "undefined." This error occurs when you tell the computer **GOTO** or **GOSUB** (Chapter 11) to a nonexistent numbered line. Usually, it's due to a typo.

Clear the computer's memory and type the following lines:

```
10 PRINT "REPEAT"
20 GOTO 100
```

List and run the program.

There is no line **100**. Hence, the error in line **20**.

Edit line 20 to read 20 GOTO 10 and run the program again. (When you have seen that it works, stop it with [RUN/STOP].)

There are also several error messages that have to do with using disk and tape *files* and with *user-defined functions* (things you haven't learned to do in this book).

A few other messages are quite rare. Unless you do a lot of mathematics, you won't see **FORMULA TOO COMPLEX** or **OVERFLOW** (which occurs when the result of a calculation is too large for the computer to handle). And on the Commodore 64, you are rather unlikely to be **OUT OF MEMORY**.

?LOAD tells you there is a problem loading a program from a tape, and **?VERIFY** tells you that the program on tape does not match the one in the computer.

Messages Associated with INPUT

Certain messages are produced in conjunction with the **INPUT** procedure. If you are supposed to input a number and give a character string instead, the computer tells you **?REDO FROM START** and puts a question mark and the cursor on the next line. (Even if you are inputting several variables for the same **INPUT** instruction, if any of them are a "mismatch," you have to redo all of them from start.)

If the **INPUT** line calls for several pieces of data (for example, **INPUT X,Y,Z**) and you press [RETURN] before you have entered everything called for, the computer puts two question marks on the screen to tell you it is waiting for more information.

Finally, if you type in too many pieces of information for an **INPUT** instruction, the computer accepts only those up to the amount of numbers asked for and ignores the rest. It then tells you **?EXTRA IGNORED**.

Clear the computer's memory and type the following lines:

```
10 INPUT X,Y,Z$,B
20 PRINT X,Y,Z$,B
```

List the program.

(Again, the variable with the dollar sign after it—**Z$**—is a string variable, something covered in Chapter 10. A string variable is simply a variable that stands for a string of characters instead of a number.)

Run the program.

When the question mark appears, type 7 and press [RETURN].

The computer sets **X** equal to **7**. Since it is still waiting for more information, it displays two question marks on the screen.

Now, type H,H,5 and press [RETURN].

The computer has a **7** and is looking for a number for the second variable (which is **Y**). As soon as you typed in the first **H**, a letter, it saw a problem. It now asks you **?REDO FROM START.**

So, try again. This time, type 5,6,H,7,8 and press [RETURN].

This time, the computer is happy. It has everything it asked for in just the right format. The only problem is that you gave it one more item than it asked for. But it goes on with the program. It simply informs you that the extra number (the **8**) was not used—**?EXTRA IGNORED**.

So much for error messages. Now, what if . . .

The Program Runs without Error Messages, But Doesn't Do What It Is Supposed to Do

The first step is to see if you can figure out the problem from what the program *does* do. Suppose the program stops prematurely (which you know because not all of the output was printed), or it gives all zeros instead of the various numbers it was supposed to print, or the design you have worked on is upside down. Cute little problems like these are your first and most important clues.

With this evidence, go back and look at your program listing. What could explain this result? Think it through again, line by line.

Some things to look for that come up fairly often—besides the usual typos—are misnamed variables (remember that any variable the computer sees for the first time, it sets at **Ø**), misplaced punctuation in **PRINT** lines, confusion over where loops end, and misuse of greater than (>) and less than (<) signs. (The most common mistake concerning the last item is to think that "greater than" means "greater than or equal to," and that "less than" means "less than or equal to.")

If you still cannot solve your problem:

1. Try different input information and see what the effect is. In particular, try numbers that fall close to any decision points or limits (values in **IF** comparisons) in your program.
2. Insert a **PRINT** line that prints the value of the variables after any suspicious parts of the program. You often get some surprises (and clues) from this.
3. If your program goes through one or more loops and you suspect that it is not going through them as it should, set of *counters*—a line like **N=N+1**—that gets one higher each time the program comes to it. Have the value of the counter printed each time through.
4. Set up a variable table on paper (as we did in Chapters 6

and 10), and follow the logic of the program that way. This quite often solves your problem, but it can be a lot of work, so it is a last resort.

If you still cannot solve the problem, take a break and come back to it later. No violence, please! This is an utterly common, normal, human predicament. A rested brain often discovers the answer it was ignoring or misinterpreting when it was tired and jaded. (Psychologists, and birds, call these breaks "incubation" because a lot happens when you seem to be just sitting around.) Leaving the computer for a while is by far the most successful of all solutions once you sense you are making no progress.

One other point: Although computers are wonderful and can do all kinds of marvelous things for you, they can't do everything. It's possible that the reason you can't get your program to do what you want is that it can't be done on your computer!

In this chapter, you have learned about error messages and steps to debug your original programs. Now, we're coming into port—from here on, it's all tug boats and harbor pilots. In the next chapter, you will learn about finding, copying, and adapting programs from books and magazines.

Summary

This chapter describes the various error messages the computer produces and how to discover and fix errors that do not produce error messages. Finding and repairing program errors can be exciting detective work.

If a program stops midway, the computer tells you the type of problem and the line number where it occurred. The error message—**?SYNTAX ERROR IN 85**, for example—consists of a question mark, a short phrase describing the problem, and the number of the line the computer was trying to carry out when it ran into the problem.

?SYNTAX is a catchall error message. It covers any problem that does not have a more specific error message. Usually, it results from a typing mistake (as do most computer errors), misplaced punctuation in a **PRINT** instruction, or a missing parenthesis.

Many error messages are self-explanatory—such as **?NEXT WITHOUT FOR** or **?RETURN WITHOUT GOSUB**. Some are not so obvious:

?BAD SUBSCRIPT means you used a higher numbered subscript than you dimensioned for. (Or you used a subscript higher than **10** if you did not dimension at all.)

?ILLEGAL DIRECT means you used an instruction in command (or *direct*) mode that can only be used in numbered program lines.

?ILLEGAL QUANTITY means you used a number in a **POKE** or other instruction that is outside the allowable range for that instruction. For example, since character codes go up to only **255, CHR$(300)** would produce this error message.

?OUT OF DATA means the computer came to a **READ** instruction and there were no **DATA** lines in the program, or the **DATA** lines had already been used by previous **READ** instructions. (This error message often appears when you accidentally press [RETURN] when the cursor is on a screen line that says **READY.** It interprets it as a command to **READ** a value for the variable **Y.** Of course, there may not be any data from which it can read a value for **Y.** When this happens, ignore it. It doesn't hurt anything.)

?REDIM'D ARRAY means the computer came to a **DIM** line for the second time.

?STRING TOO LONG means you told the computer to remember a character string of more than 255 characters. This is very rare.

?TYPE MISMATCH means you set a numeric variable equal to a character string, or vice versa.

?UNDEF'D STATEMENT means a **GOTO** or **GOSUB** instruction sent the computer to a nonexistent line number.

The computer can produce several other error messages. Some involve misuse of advanced procedures (such as *data files* and *user-defined functions* that you have not learned). Some occur very rarely or only in connection with the kind of extensive mathematics you are unlikely to use.

Some error messages are not really error messages. For example, **?LOAD** tells you the computer is having trouble loading a program, and **?VERIFY** tells you the program on tape and in memory do not match. **?VERIFY** always follows a **VERIFY** instruction.

Finally, some special messages are associated with **INPUT.** If you do not type as many numbers as the **INPUT** instruction calls for, after you press [RETURN], the screen shows **??**. If you put in letters when the computer is expecting numbers, it tells you **?REDO FROM START**. If you put in more pieces of information than are called for, it uses as much as it can, then tells you **?EXTRA IGNORED** and continues with the program.

If your program runs without any error messages but gives the wrong result, try to figure out the problem from the result it does give you. How could it have come up with that?

If you still can't figure out the problem, you can get some clues by doing the following:

1. Try out different input information.
2. Insert lines to print variable values at various points throughout the program.

3. Insert program lines that count the number of times they are passed (such as **N=N+1**) at various points in the program, printing the counts along the way.
4. Construct a variable table for the program.

If you *still* can't solve the problem, take a break and come back to it later with a fresh mind.

Terms and Concepts Introduced in Chapter 17

Debugging

Error messages (see Appendix F for list)

Testing program with trial data

Practice Exercise

(Answers and comments in Appendix A.)

Here is a little program to quiz a youngster on the multiplication tables.[1] IT IS FULL OF MISTAKES. See if you can fix the mistakes and get the program to run correctly!

```
10 RE MULTIPLICATION TABLE QUIZ
20 FOR X=1 TO 9
30 FOR Y=1 TO 9
40 PRINT X;" X";Y;" = ?"
50 PRINT "□TYPE ANSWER" &PRESS<RETURN>"
51 INPUT A
60 IF A=X*X PRINT "CORRECT!!"
70 IF A<>X*Y THEN PRINT"SORRY, IT'S"X*Y
80 FOR DL=1 TO 300
90 NEXT D
100 NEXT X
110 NEXT Y
```

[1]You could make a more interesting program with **RND(1)**, but we want to use an example that only requires having read Chapters 1 to 8.

PART IV CHAPTER 18

Using Programs from Books and Magazines

You have now learned the steps of writing and debugging an original program. As we pointed out in Chapter 16, most of the things you might want to program have already been programmed—or at least programs exist that are close enough to what you want to be adaptable. Thus, programming mainly involves locating and adapting existing programs. That's what this chapter is about. (Now, don't holler, "Why did I learn all this stuff then?" You need to know just as much, just as well, to understand and adapt other people's programs as you need to write your own.)

Where to Find Programs—and Whether to Use What You Find

The main sources for programs to copy are computer magazines and books. There are now more than a dozen popular magazines for users of personal computers—including at least four exclusively for Commodore owners. Each month, computer magazines publish articles on such topics as how to buy the best printer for your needs or how to evaluate prepackaged programs. Most important, they include many articles with programs for you to copy and use on your own computer. There are programs for games and for business, for photography and for education, for keeping track of sports statistics, and for keeping track of your investments—an almost unlimited wealth of programs. Your library probably keeps back issues of many of these magazines, so you aren't even limited to the current month's crop.

Book shops and computer stores are also crammed with books containing program listings. Some books specialize in programs for various business or professional uses, others offer a smorgasbord of programs for home or educational use. Some are just for fun and games!

Moreover, if you join a users' club or if you have a friend with a computer, you'll quickly find yourself trading and sharing programs.

Thus, the problem is not finding programs—it's sorting through them to find the ones you want! Here are some important considerations:

1. Is the program written for the Commodore 64?
2. Does it require accessories you don't have? A disk drive? A printer? A modem?
3. Can you understand the program? If the program is written in machine language—or even if it is in BASIC but looks too obscure—leave it alone. Just *copying* a long program accurately is hard if you don't understand it. And once you have copied it, you will probably want to modify it.
4. Does the program do what you want done?

Actually Copying the Program

The first step in copying a program is to be sure you thoroughly understand the program.

Often, books and magazines use some of the shortcuts described in Chapter 11. Review those before attempting to figure out such a program.

Once you understand it, copy it. Check your typing carefully. If the program doesn't work, it's probably due to a typographical error on your part.

Books and magazines, however, also have typographical errors. (We hope this book will be an exception.) Undoubtedly, the author probably had the program working properly when it was submitted, so it should still be close to functional after typesetting. But you may have to debug the program slightly, just as if it were a new program of your own.

Modifying Programs Written for the Commodore 64

Often, you can take a program that does something similar to what you want and modify it to do exactly what you want. For example, if you want a program that does a monthly statement of income and expenses, you might start with the checkbook-balancing program in this book. Or, if you want a program that keeps track of your stamp collection, the coin collection program in Chapter 10 would be an obvious place to start.

Modifying programs should be a familiar activity—that's what you've mainly been doing in the practice exercises and throughout this book. Just be careful not to mix up loops and variable names. Also, be sure to make the appropriate changes in any **DIM** instructions and in the line numbers in **GOTO** and **GOSUB** instructions.

Converting a Program Written for the VIC-20

Because the VIC-20 has been around longer than the Commodore 64, more programs have been written for it. If you are going to use a program written for another machine, those written for the VIC-20 are the easiest to convert.

VIC-20 BASIC is identical to Commodore 64 BASIC. However, there are some differences:

1. The **POKE** addresses are all different—even the screen location and screen color addresses—but the principles for screen displays are similar and the screen display code for characters is identical.
2. The VIC-20 sound generator works differently. If sound is involved, you'll have to figure out what the program is trying to accomplish and totally rewrite the sound portion for the Commodore 64.
3. Since the VIC-20 screen is only 22 spaces across, any designs laid out for that screen size may look puny on the Commodore 64.
4. The Commodore 64 gives you far more memory and a bigger screen size. In addition, sound and animation possibilities (because of sprites) are much, much richer on the Commodore 64. You should be able to modify a VIC-20 program to run rings around the original!

In sum, the differences that are likely to affect you are all in the areas of graphics, sound, and anything else that is poked. Otherwise, try using VIC-20 programs directly—they should work.

Adapting Programs Written for Other Computers

BASIC is fairly—but not completely—standard among personal computers. However, **POKE** addresses, screen size, means of producing screen graphics and color, and sound generators are almost completely different from machine to machine. If the program involves a lot of sound or graphics, or many **POKEs** of any kind (or if it is written in machine language), it is nearly impossible to adapt it for the Commodore 64. If you insist, you will have to figure out what the program is supposed to do and rewrite most of it from scratch for the Commodore 64.

Otherwise (if the program does not involve a lot of graphics, sound, or other **POKE** instructions, and it is in BASIC), adapting it for the Commodore 64 is not very difficult. Often, like programs for the VIC-20, you can just copy it directly.

Every computer has a few idiosyncracies in its BASIC. If these come up, you'll have to find solutions. (Most of these have to do with the commands taught after Chapter 8.) We'll try to point out some of the most common variations from the Commodore 64's version of BASIC:

1. Almost all other computers allow you to use a variable to stand for the destination of a **GOTO** or **GOSUB** instruction. With the Commodore 64, you can't have **GOTO X** or **GOSUB Y**—you must use a line number. If a program contains this, you have to get around it. Usually, an **IF . . . THEN . . .** line works. For example, a part of a program might say:

```
300 PRINT "TYPE 1 TO GO ON, 2 TO STOP"
310 INPUT CS
320 GOTO CS*1000
1000 ..........
2000 END
```

   On the Commodore 64, you could replace line **320** with these two lines:

```
320 IF CS=1 THEN GOTO 1000
330 IF CS=2 THEN GOTO 2000
```

2. String-slicing procedures are not very standard. The method of the Commodore 64 is fairly common, but some computers use **SEG$.** Some use **MID$** but define it differently. Some use procedures that would make **A$(7 TO 9)** what the Commodore 64 would call **MID$(A$,7,3)**. You'll just have to figure out what the program is doing (from the purpose of the string slicing) and convert it to the Commodore 64 procedure.
3. Many computers require you to give the maximum length of each subscripted string variable in the **DIM** instruction, as well as the number of them. For example, **DIM A$(50,8)** would mean to allow room in memory for 50 subscripted variables of up to eight characters each. On the Commodore 64, **DIM A$(50)** would be sufficient. (And **DIM A$(50,8)** would give an error message at the first line you used **A$**!)
4. The Commodore 64 can condense several lines into one (as we discussed in Chapter 11)—for example, **50 FOR D=1 TO 1000:NEXT D**. Some computers can't condense lines. Others permit it, but use a backslash ("\") instead of a colon as a separator. This is easy to fix when you see it.
5. Many computers allow—or require—you to use brackets [like these] instead of parentheses (like these). The Commodore 64 will not accept brackets in program instructions. (But they are okay as part of a character string.) Thus **SPC[7]** and **PRINT 6*[3+2]** would have to be converted to **SPC(7)** and **PRINT 6*(3+2)**.

6. Punctuation with **PRINT** instructions is fairly standard. The effect of a comma, however, depends on the size of the print zones. In the Commodore 64, there are 10 spaces per print zone and four print zones across the screen. On some computers, print zones are 8 or 11 spaces long, or some other length. Often, there are only two print zones across the screen. This may require replanning some tables.
7. The clear-screen procedure of putting the reverse-heart symbol in quotes is unique to the Commodore 64. Other computers have a special BASIC command for this purpose—**CLS** and **CALL CLEAR** are the most common. (Be careful—**CLEAR** or **CLR**, which look similar, *usually* mean something like "clear variable values from memory," not "clear the screen.")

In this chapter, you learned how to find and select programs and how to copy, modify, and adapt them for the Commodore 64.

This completes your introduction to the Commodore 64. From now on, it's just words—English words, but useful ones. In the next chapter, we will take stock of how much you have learned, how much you still can learn, and what else to buy (or not buy) for your computer. Finally, we'll talk about programming as an art.

Summary

Magazines and books abound with programs you can copy and use on your computer. Some magazines are exclusively for Commodore owners. Libraries are a good place to look for back issues. Book shops and computer stores are good sources for books containing program listings—either for specialized business or professional use, or for more general home, educational, or entertainment use. Friends with computers and users' groups are other good sources of program material.

When selecting a program to copy, consider whether it is written for the Commodore 64, whether it requires any accessories you do not have, whether you can understand it, and whether it really does what you want done.

Once you have selected a program, study it until you thoroughly understand it. Then copy it carefully. If it doesn't work, and you copied it accurately, there could be a mistake in the printed program. In that case, you'll have to debug it just as you would a program you had written yourself.

This book has already given you experience in modifying programs written for one purpose to be used for a different but related purpose. You'll use that experience frequently.

As for programs written for other computers, those for the VIC-20 can be transferred to the Commodore 64 with little change, unless they involve graphics or sound. The **POKE** addresses for screen display and the screen size are totally different. Also, the VIC-20 sound generator works on different principles.

Programs written for other computers besides the VIC-20 can be adapted for use on the Commodore 64 if you understand them and they do not involve a lot of graphics, sound, or poking. There are a few differences among computers in the actual BASIC programming. The Commodore 64 does not permit the use of variables as destinations of **GOTO** or **GOSUB** instructions. String-slicing procedures vary. Many computers require you to dimension for the lengths of strings when you use subscripted string variables. On the Commodore, you must use parentheses instead of brackets. The condensing of program lines, the number and length of print zones, and the procedures to clear the screen all vary.

Terms and Concepts Introduced in Chapter 18

Adapting programs written for the Commodore 64, the VIC-20, and other computers

Practice Exercise

(Answer and comments in Appendix A.)

The following is a program written for another computer. It converts U.S. dollars to their equivalent in three other currencies. (The exchange rates in the program are probably not correct as of the day you do this program, so look them up in the financial section of a newspaper if you plan to use this program.) Adapt this program to the Commodore 64.

```
10 PRINT "$ AMOUNT TO EXCHANGE?"
14 INPUT D
20 PRINT TAB[4];"CANADIAN=1"
30 PRINT TAB[4];"ENGLISH=2"
40 PRINT TAB[4];"FRENCH=3"
50 PRINT "TYPE NUMBER"
60 INPUT C
65 CLS
70 GOTO C*200
200 PRINT D/.84;"CANADIAN DOLLARS"
210 END
400 PRINT D/1.58;"BRITISH POUNDS"
410 END
600 PRINT D/.14;"FRENCH FRANCS"
610 END
```

PART IV CHAPTER 19

Where to Go from Here—Programming as an Art

This is it. This chapter concludes your introduction to the Commodore 64. Now, we will discuss how to learn more and—if you wish—buy more to suit your needs. In addition, we will discuss the "essence" of programming.

Learning More

You've learned almost all of the BASIC language, you've learned how to use the sophisticated abilities of the Commodore 64 to make animated figures and sound, and you've learned some of the tricks of the trade for putting all this together into fun and useful programs. You've learned a lot.

But there's still more you could learn. The main thing is to improve on what you've learned already. You now have the tools and some experience in using them. What you now need most is more experience. That means practice, practice, and more practice.

A good way to master these fundamentals is to read and/or copy programs from books and magazines, being sure to think through exactly why they are written as they are. With every program, you may learn some new trick or some new angle on programming. In addition, you solidify the basics.

There are, of course, many other things to learn—a little more about BASIC, a lot more about the graphic and sound capabilities of the Commodore 64, something about machine language, and the languages and procedures made available by various software—such as spreadsheet business forecasting, more advanced programming languages such as Pascal, and so on. There are books and courses about all these. You now have enough background to learn almost anything in these areas.

But the main thing—and we can't emphasize this enough—is to practice what you have already learned. Master these tools. Once you have mastered them, there will be few programs you can't handle.

Buying More

If you look at computer magazines or walk into any computer store, you'll be overwhelmed by all the accessories you can buy for your Commodore 64. In addition to the software described in Chapter 2, there are all kinds of additional machinery you can purchase—printers, disk drives, modems, and so on. These are called *peripherals*. Which do you really need?

If you'll be doing a lot of programming, you almost have to buy something on which to store programs. For most home users, the Commodore Datassette™ is quite sufficient.

If your programs are long, however, or involve keeping track of a lot of information, or if you want to use sophisticated software, then you need a disk drive. A disk drive is a machine that uses a disk—an item that comes permanently sealed in a 5-1/4 inch-square case. The disk is made of a magnetic material that allows it to record programs and other information. These disks are often called *floppy disks,* although they are not really very flexible (and it's best not to bend them much). They are also known as *diskettes*. Both of these terms came about as a comparison to the *hard disks* that are used on large-scale industrial computers.

A disk drive works pretty much the same as the Datassette. But you can save and load long programs on the disk drive much more quickly. More important, the computer can find anything you ask for on a disk (such as a small program buried among larger ones) without your having personally to search for it, as is necessary on a tape.

Another useful item, if you will be doing a lot of programming, is a printer. A printer allows you to make a copy of your programs and the output from programs (such as tables and lists) that you can keep, send to people, or do whatever you want with.

Commodore makes a printer that is designed for the Commodore 64. It hooks up easily and produces all the Commodore 64 special graphic characters. It is what is called a *dot matrix* printer, which means it produces letters made up of little dots. The program listings in this book were done on the Commodore printer. This printing quality is adequate for most purposes, but would probably not do for business letters or important school reports.

Many other printers work on the Commodore 64 with a special converter your Commodore dealer sells. There are four main things to consider when buying a printer:

1. Will it work with the Commodore 64 without a lot of special wiring or attachments? If you plan to use it for word processing, is it compatible with the word-processing program you are using?
2. Do you like the quality and style of the letters it prints? (See Chapter 2 for a discussion of this issue.)

3. How fast does it print? Any printer is much faster than a typewriter. But, if you are printing a lot, even three times the speed of a typist can feel pretty slow. On the other hand, if you are not printing a great deal, you can get a much better quality printer for the money if you buy a "slow" one.
4. As with any purchase, you must consider price, availability of service, and quality. Printers are mechanical devices and they do break down. Also keep in mind that some printers require expensive or frequent replacements of ribbons.

Yet another item you can buy is called a *modem*—a device that connects your computer to a telephone line. It works with most ordinary phones (although not with a Princess™ phone). A modem allows you to communicate with other people who have computers hooked up to modems, and with various information services that can give you information from stock prices to encyclopedia entries. Being part of these networks can be very exciting.

The modem for the Commodore 64 is not very expensive, but the phone bills and the bills for using the various information services add up quickly. Be careful if money is tight![1]

Finally, a popular addition to your system is a color monitor—a special TV set that produces a much sharper picture than an ordinary set. The monitor won't work for ordinary TV shows, but does a beautiful job with any graphics you produce, or with such software as games or educational programs. The costs are continually coming down, so if graphics are important to you, take a look at monitors.

Programming as an Art

You have learned about the tools of programming. With practice, you will master those tools. Programming, though, is more than the tools.

A painter must first know the brushes and the paints and all the techniques for creating perspective and proportion and shading. But all of that doesn't make a painting. Something more must be added—the painter's creativity.

With the tools of programming, you *create* a program. There are some directions, some guidelines. Ultimately, the program you produce is a product of your own mind. It's a creation. Sometimes, it's a work of art.

The art of programming requires a lot from you. It requires the ability to always keep in mind the large view of what you are trying to accomplish while maintaining constant vigilance for the

[1] One popular use for modems is not very expensive. Many college students taking computer courses find that the terminals that let them use the large university computers are constantly busy. With a modem, they can use their Commodore 64 as a terminal to the college computer!

most minute details—a combination of broad comprehension and fine focus and creativity on top of that.

On the other hand, you don't need any training in math or science. You don't even have to know very much grammar. Your level of education is almost irrelevant. All you need are the few simple tools you have learned in this book, plus your mind.

Bring to your programming a rested and alert mind, and you'll enjoy expressing your creativity.[2]

Thank you for letting us help you learn.

Summary

This chapter concludes your introduction to the Commodore 64 with some advice on learning more, buying more, and programming as an art.

You have learned most of BASIC, and you've learned how to use most of the sophisticated graphic and sound capabilities of your Commodore 64. There is more you could learn—a little more BASIC, more techniques for using Commodore 64 graphics and sound, and whole new topics such as machine language and other programming languages.

Most important, however, is mastering the tools you have already learned in this book. Mainly, that comes from practice. You can also learn a lot from thoroughly studying programs in magazines and books.

You can buy all kinds of accessories for your Commodore 64. We discussed software in Chapter 2. The accessory machines you can buy are called *peripherals*.

It is very useful to be able to save programs you write and use prepackaged programs that are not available on cartridges. For most Commodore 64 owners, the Commodore Datassette suffices for these purposes. If you will be storing many programs, though, or using sophisticated software, you'll need a disk drive. Instead of tape, this machine records information on disks—little records that come permanently sealed in 5-1/4 inch-square envelopes. Disks allow you to find any saved program without having to go through all the ones in front of it.

A printer makes copies of your programs and the output from programs. It gives you a printed record of what you've done. The Commodore printer hooks up directly to your computer. Other printers can also be made to hook up to the Commodore 64 with

[2] As psychologists, we are very aware of the importance of the quality of awareness that approaches any task—be it computer programming or flying a plane. But it is difficult to recommend anything well established, from sound scientific research, that really improves the functioning of our nervous system. The only significant exception, in our opinion, is suggested by the recent physiological and psychological studies on the Transcendental Meditation program. Findings indicate that it has been highly effective in developing precisely the mental abilities needed for creative, successful programming. From the research, and our own experience, we recommend it to you highly.

some additional connectors. When buying a printer, consider how easy it is to hook up to the Commodore 64 and how suitable the quality, style of print, printing speed, and overall value and reliability are for your purposes.

A modem hooks up your computer to a telephone line so that you can communicate with other computer users and take advantage of various information services.

A monitor is a special TV set designed specifically for computers. It makes your screen displays much sharper.

In this book, you have been introduced to the main tools of computer programming on your Commodore 64. With practice, you will master these tools. But programming requires something more—alertness and creativity. In the end, programming is an art, a creative art. Have fun with it.

APPENDIX

APPENDIX A

Answers to Practice Exercises

Chapter 4

1. To solve each of these problems, type **PRINT,** then the calculation, then press the [RETURN] key. Remember not to use quotes when using the **PRINT** command for arithmetic calculations.

a. Type **PRINT 198+207**
Press [RETURN].

The result should appear on the next line: **405**

b. Type **PRINT 123.7−100**
The result should be **23.7**

c. Type **PRINT 1798+34**
NOTE: Do not use commas when you put large numbers into the computer!
The result should be **1832**

d. Type **PRINT 100*54**
NOTE: Use the * for "times"—the computer confuses the "×" with the letter **X**.
The result should be **5400**

e. Type **PRINT 547*67*32.19**
The result should be **1179731.3**

f. Type **PRINT 132.80/4**
NOTE: Don't put in the dollar sign—this would confuse the computer, since it is only looking for numbers. Also, remember that the division sign is the slash on the computer.
The result should be **33.2**
NOTE: In displaying results, the computer only prints as many decimal places as is necessary to give an accurate answer. **33.20** is the same as **33.2**, so it displays the shorter form.

2. a.
```
PRINT "1 TO GET READY"
```
b.
```
PRINT "THE WORLD IS TO MUCH WITH US;
                    LATE AND SOON,"
```

NOTE: Remember not to press [RETURN] just because you are at the end of a line.

3. Obviously, there are many ways to make these pictures. Here's how we did it:

```
PRINT "▛▂▂▂▜"

PRINT "▚▔▔▔▔▔▔▞"
```

5. If you had trouble with this exercise, reread pages 40 to 41 on editing within quotes.

Chapter 5

1. You can use any number, but it should be a numbered line something like this:

```
10 PRINT "ONE LINE PROGRAM"
```

2. Use the normal [SHIFT] and [CLR/HOME] procedure.

6. Remember to be careful when editing within quotes and to press [RETURN] when you are finished.

9. The new line must be a **GOTO** line with a higher number than the first. For example:

```
10 PRINT "2 PROGRAM LINES"
20 GOTO 10
```

12. Press the [RUN/STOP] key.

13. Type **CONT** and press [RETURN].

Chapter 6

Remember, there is no one right way to write a program. If what you do works, it's okay. The following programs are written the way they are to be close to the examples in the chapter.

1.
```
10 FOR SN=1 TO 2
20 PRINT "SHIP NUMBER"
30 PRINT SN
40 NEXT SN
```

2.
```
10 FOR SN=100 TO 102
```

(The rest is the same as the first answer.)

```
10 FOR SN=100 TO 400 STEP 100
```

3. (The rest is the same as the first answer.)
4. Same as above, but add the following line:

```
15 IF SN=300 THEN GOTO 40
```

5.
```
5 INPUT N
10 FOR SN=0 TO 2
20 PRINT "SHIP NUMBER"
30 PRINT SN+N
40 NEXT SN
```

NOTE: Be sure not to use the same variable name for **INPUT** as for the **FOR . . . TO . . .** and **NEXT** loop.

6. Add these lines to the above program:

```
4 PRINT "TYPE STARTING SHIP NUMBER"
6 PRINT "□"
```

7. Hold [RUN/STOP] and press [RESTORE].

Chapter 7

Again, there are many ways to do each of these programs. If your way works, it doesn't have to be the same as the answers given here.

2.
```
10 PRINT "HOW MANY YARDS"
20 INPUT Y
30 PRINT "□" Y "YARDS =" Y*36 "INCHES"
```

3.
```
10 PRINT "HOW MANY FEET"
20 INPUT F
30 PRINT "□" F "FEET =" F*.333 "YARDS"
```

4. Same as number 3 above, but add the following line:

```
40 PRINT "□" F "FEET =" F*.3048 "METERS"
```

Chapter 8

1. Here's one way to do this:

```
10 PRINT "        HERE";SPC(10);"AND"
20 PRINT
30 PRINT
40 PRINT SPC(12);"THERE"
50 PRINT
60 PRINT
70 PRINT SPC(13);"AND"
80 PRINT
90 PRINT
100 PRINT ,"EVERYWHERE"
```

2. Here's one way to make this table:

```
10 PRINT SPC(12);"METRIC CONVERSIONS"
20 PRINT
30 PRINT"METER","INCH","FOOT","YARD"
40 FOR M=1 TO 10
50 I=M*39.37
60 F=M*3.282
70 Y=M*1.094
80 PRINT M,I,F,Y
90 NEXT M
```

Chapter 9

1. There are several ways to make this program do what you want. One way is to add the following lines:

```
110 IF L=3 THEN GOTO 300
120 IF L=4 THEN GOTO 400
150 PRINT "FACE VL","YEAR","MINT CODE"
160 FOR J=1 TO 5
170 PRINT FV(J),Y(J),MC(J)
180 NEXT J
190 PRINT
195 GOTO 60
300 PRINT "LIST COINS OF WHICH YEAR"
310 INPUT WY
320 PRINT "FACE VL","YEAR","MINT CODE"
330 FOR J=1 TO 5
340 IF Y(J)=WY THEN PRINT FV(J),Y(J),MC(J)
350 NEXT J
360 PRINT
370 GOTO 60
400 PRINT "LIST COINS OF WHICH MINT CODE"
410 INPUT WM
420 PRINT "FACE VL","YEAR","MINT CODE"
430 FOR J=1 TO 5
440 IF MC(J)=WM THEN PRINT FV(J),Y(J),MC(J)
450 NEXT J
460 PRINT
470 GOTO 60
```

2. You must change all the lines that refer to the number of coins—**10, 20, 230,** and the corresponding lines you added for your answer to question 1.

```
10 DIM FV(12),Y(12),MC(12)
20 FOR K=1 TO 12
230 FOR J=1 TO 12
```

To the program lines we wrote in the answer to question 1, we would add the following lines:

```
160 FOR J=1 TO 12
330 FOR J=1 TO 12
430 FOR J=1 TO 12
```

Chapter 10

1. Here's one way to write this program:

```
10 OD$="-JAFEMAAPMAJNJLAUSEOCNODE"
20 PRINT "TYPE # OF MONTH"
30 INPUT N
40 PRINT MID$(OD$,2*N,2)
```

2. Use a **FOR . . . TO . . .** and **NEXT** loop to get 10 numbers.

For example, for a.:

```
10 FOR X=1 TO 10
20 PRINT 1+INT(RND(1)*5)
30 NEXT X
```

Programs for b. through e. are the same except for line **20**:

b. **20 PRINT 1 + INT(RND(1)*8)**

c. **20 PRINT INT(RND(1)*3)**

d. **20 PRINT INT(RND(1)*26)**

e. **20 PRINT 1 + INT(RND(1)*4)**

3. Here's one way to do it:

```
10 DIM NT$(8),FQ(8)
20 FOR M=1 TO 8
30 READ NT$(M),FQ(M)
40 NEXT M
50 R=1+INT(RND(1)*8)
60 PRINT "□";NT$(R)
70 PRINT "TYPE ITS FREQUENCY"
80 INPUT T
90 IF T =FQ(R) THEN GOTO 120
100 PRINT "SORRY, ANSWER IS ";FQ(R)
110 GOTO 130
120 PRINT "*****CORRECT*****"
130 FOR D=1 TO 1000
140 NEXT D
150 GOTO 50
160 DATA C,262,D,294,E,330,F,349
170 DATA G,392,A,440,B,494, C,523
```

Chapter 11

1. Here's a possible program to do this:

```
10 N=1+INT(RND(1)*2)
20 PRINT "GUESS NUMBER 1 OR 2"
30 INPUT GN
40 IF GN=N THEN GOSUB 1000
50 IF GN<>N THEN GOSUB 2000
60 N=1+INT(RND(1)*4)
70 PRINT "GUESS A NUMBER 1 TO 4"
80 INPUT GN
90 IF GN=N THEN GOSUB 1000
100 IF GN<>N THEN GOSUB 2000
110 N=1+INT(RND(1)*10)
120 PRINT "GUESS A NUMBER 1 TO 10"
130 INPUT GN
140 IF GN=N THEN GOSUB 1000
150 IF GN<>N THEN GOSUB 2000
160 GOTO 10
1000 FOR X=1 TO 50
1010 PRINT "■"
1020 PRINT "■"
1030 PRINT "CORRECT"
1040 NEXT X
1050 RETURN
2000 PRINT "SORRY IT WAS";N
2010 FOR D=1 TO 1000
2020 NEXT D
2030 RETURN
```

2. Here's one way to get a character code printed:

```
10 PRINT "TYPE IN A LETTER"
20 INPUT L$
30 PRINT ASC(L$)
40 GOTO 10
```

3. `10?"HOW MANY FEET":INPUTF:?"■";F;"FEET = ";F*.3048;"METERS"`

Chapter 12

Our approach to making the face exactly follows the program used in the chapter to make the boat:

```
10 PRINT "■"
20 READ X,Y,SDC
30 IF X<0 THEN GOTO 30
40 PL=1024+X+Y*40
50 POKE PL,SDC
```

```
60 POKE PL+54272,7
70 GOTO 20
1000 DATA 2,1,93,3,1,93,4,1,93,5,1,93
1010 DATA 6,1,93,2,2,40,3,2,15,5,2,15
1020 DATA 6,2,41,2,3,40,4,3,65,6,3,41
1030 DATA 6,4,41,2,4,40,3,4,74,4,4,67
1040 DATA 5,4,75,3,5,77,4,5,100,5,5,78
1050 DATA -1,-1,-1
```

NOTE: SAVE THIS PROGRAM—YOU WILL USE IT IN ANSWERING THE PRACTICE EXERCISE IN THE NEXT CHAPTER!

Chapter 13

1. Because the poking process is rather slow, alternating the "open eye" (the letter "O") with the dash gives the appearance of a wink. Here's our version of a program to do this:

```
10 PRINT "□"
20 READ X,Y,SDC
30 IF X<0 THEN GOTO 80
40 PL=1024+X+Y*40
50 POKE PL,SDC
60 POKE PL+54272,7
70 GOTO 20
80 POKE 1107,45
90 POKE 1107,15
100 GOTO 80
1000 DATA 2,1,93,3,1,93,4,1,93,5,1,93
1010 DATA 6,1,93,2,2,40,3,2,15,5,2,15
1020 DATA 6,2,41,2,3,40,4,3,65,6,3,41
1030 DATA 6,4,41,2,4,40,3,4,74,4,4,67
1040 DATA 5,4,75,3,5,77,4,5,100,5,5,78
1050 DATA -1,-1,-1
```

2. This program follows exactly what you did with animating the boat:

```
8 FOR A=0 TO 30
10 PRINT "□"
20 READ X,Y,SDC
30 IF X<0 THEN GOTO 80
40 PL=1024+X+Y*40
45 PL=PL+A
50 POKE PL,SDC
60 POKE PL+54272,7
70 GOTO 20
80 RESTORE
90 NEXT A
1000 DATA 2,1,93,3,1,93,4,1,93,5,1,93
1010 DATA 6,1,93,2,2,40,3,2,15,5,2,15
```

```
1020 DATA 6,2,41,2,3,40,4,3,65,6,3,41
1030 DATA 6,4,41,2,4,40,3,4,74,4,4,67
1040 DATA 5,4,75,3,5,77,4,5,100,5,5,78
1050 DATA -1,-1,-1
```

3. Again, we just took the program in the chapter and used it with this design:

```
8 A=0
10 PRINT "□"
20 READ X,Y,SDC
30 IF X<0 THEN GOTO 80
40 PL=1024+X+Y*40
45 PL=PL+A
50 POKE PL,SDC
60 POKE PL+54272,7
70 GOTO 20
80 RESTORE
90 GET K
100 IF K=0 THEN GOTO 90
110 IF K=8 THEN A=A-1
120 IF K=9 THEN A=A+1
130 GOTO 10
1000 DATA 2,1,93,3,1,93,4,1,93,5,1,93
1010 DATA 6,1,93,2,2,40,3,2,15,5,2,15
1020 DATA 6,2,41,2,3,40,4,3,65,6,3,41
1030 DATA 6,4,41,2,4,40,3,4,74,4,4,67
1040 DATA 5,4,75,3,5,77,4,5,100,5,5,78
1050 DATA -1,-1,-1
```

4. This only required two additional lines for the up and down direction. We picked the [6] and [7] keys, but any number keys would do. We picked these because they were close to the others we were already using. Here's how the program looks with the two new lines:

```
8 A=0
10 PRINT "□"
20 READ X,Y,SDC
30 IF X<0 THEN GOTO 80
40 PL=1024+X+Y*40
45 PL=PL+A
50 POKE PL,SDC
60 POKE PL+54272,7
70 GOTO 20
80 RESTORE
90 GET K
100 IF K=0 THEN GOTO 90
110 IF K=8 THEN A=A-1
120 IF K=9 THEN A=A+1
123 IF K=6 THEN A=A-40
124 IF K=7 THEN A=A+40
130 GOTO 10
```

```
1000 DATA 2,1,93,3,1,93,4,1,93,5,1,93
1010 DATA 6,1,93,2,2,40,3,2,15,5,2,15
1020 DATA 6,2,41,2,3,40,4,3,65,6,3,41
1030 DATA 6,4,41,2,4,40,3,4,74,4,4,67
1040 DATA 5,4,75,3,5,77,4,5,100,5,5,70
1050 DATA -1,-1,-1
```

Chapter 14

1. This only requires following the instructions on page 000. We hope you enjoyed making your sprite. If you had trouble, reread the chapter.

2. The two new lines you need require poking code number **3** into addresses **F+23** and **F+29**.

Chapter 15

1. There are many ways to get these sounds. Your best bet is to play around with the sound generator until you either find one that fits the bill, or until you get a sense of how to systematically create one.

a. This requires a high frequency and a longish decay time, as in the example in the chapter in which address **B+5** was set to **14**.

b. Use the table in the chapter to get the frequency for G (**392**) and use the quality settings we used throughout the chapter (**POKE B+5,9** and **POKE B+6,0**).

c. A white-noise wave pattern is required. Use the **129** and **128** wave-pattern codes (poked into **B+4**), along with the other quality settings used in b. above.

2. Our version is shown below. These notes are the low singers' part of the first few notes of the Hallelujah Chorus of Handel's *Messiah*. Here is our program for it:

```
10 B=54272
20 POKE B+24,15
30 POKE B+5,9
40 POKE B+6,0
50 READ F,T
60 POKE B,F/.06097-(256*INT(F/15.6083))
70 POKE B+1,F/15.6083
80 POKE B+4,33
90 FOR D=1 TO 1000/T
100 NEXT D
110 POKE B+4,32
120 GOTO 50
500 DATA 440,4,440,8,392,8
510 DATA 349,8,000,8,440,16,440,16
520 DATA 494,8,440,8
```

Chapter 16

There are many elegant ways to write this program using the programming tools covered in Chapters 9 to 15. However, because not everyone will have read those chapters, we will show how we would program this using only the procedures covered through Chapter 8. If you have written the program some other way—as long as it works, it's fine.

Step 3: Outlining the major program parts

I. Permanent Input—
Conversion rules for numbers to Morse Code

II. User Input—
Number to be converted

III. Computer Transformations—
Carry out the conversion

IV. Output—

A. Number input (so users can check to see that the computer got what they gave it)

B. Dots and dashes that correspond to number input

Step 4: More detailed outline

I. Permanent Input
IF . . . THEN . . . lines to make conversions—that is, if the number input is such and such, then print a particular dot and dash pattern

II. User Input
The number to be converted—prompt-and-**INPUT** sequence; use a **GOTO** loop to let users ask for another conversion, after the first is complete

III. Computer Transformations
Accomplished by the **IF . . . THEN . . .** lines in I, above

IV. Output
Print number, equals sign, and dot and dash pattern at end of each **IF . . . THEN . . .** line

Step 5: Writing program lines

First draft:

```
Get the number to be converted.          prompt
                                         INPUT N
                                         Series of IF . . . THEN . . . lines,
                                         each line with number and then
                                         PRINT conversion with format,
                                         "number=dot and dash pattern"
                                         goto starting prompt line
```

Second draft:

```
100 PRINT "TYPE NUMBER TO CONVERT"
110 INPUT N
```

```
120 IF N=0 THEN PRINT "0 = -----"
130 IF N=1 THEN PRINT "1 = .----"
140 IF N=2 THEN PRINT "2 = ..---"
150 IF N=3 THEN PRINT "3 = ...--"
160 IF N=4 THEN PRINT "4 = ....-"
170 IF N=5 THEN PRINT "5 = ....."
180 IF N=6 THEN PRINT "6 = -...."
190 IF N=7 THEN PRINT "7 = --..."
200 IF N=8 THEN PRINT "8 = ---.."
210 IF N=9 THEN PRINT "9 = ----."
220 GOTO 100
```

Because this is such a short program, it is not necessary to divide it into subroutines or to have extensive **REM** lines, but it should still be identified with a title with which to save it.

Third draft:

```
10 REM "NUMBERS-TO-MORSE"
20 REM 8/10/83 VERSION
30 E. ARON
```

The rest would be the same as before.

Step 6: Making your program work

Try it out with sample input. In this case you would want to try all 10 numbers to be sure the program gives the correct result.

Step 7: Enjoy your handiwork. And save it.

Chapter 17

Looking at it line by line, the corrected version would look something like this:

```
10 REM MULTIPLICATION TABLE QUIZ
```

—The error was leaving out the **M** in **REM**

```
20 FOR X=1 TO 9
30 FOR Y=1 TO 9
```

—These lines were okay.

```
40 PRINT "D";X;" X";Y;" =?"
```

—This line was okay as it was, but you need a clear screen to get rid of the old problem on the second and subsequent times through the program. It doesn't work in line **50**, so this is the logical place to put it. It's a fine point, actually. It will, of course, work just fine without it, but learners would see their previous answers as they went along if you did not clear the screen here.

```
50 PRINT "TYPE ANSWER & PRESS <RETURN>"
```

—The clear screen (reverse-heart symbol) had to be omitted because otherwise the screen would be cleared of the problem just printed in line **40**. Also, there was an extra quote mark after **ANSWER** and no space before the [RETURN]. Only the extra quote mark would create an error message, but when you tried to run the program, you would discover the other points.

```
51 INPUT A
```

—This line was okay.

```
60 IF A=X*Y THEN PRINT "CORRECT!!"
```

—First of all, you should have gotten an error message for this line because it left out the **THEN.** Even more serious, it was checking **A** against the wrong answer—something you would probably not notice until you tried the program with test data. The right answer is **X*Y**, not **X*X**!

```
70 IF A<>X*Y THEN PRINT"SORRY, IT'S"X*Y
80 FOR DL=1 TO 300
```

—These lines are okay, although you could put a semicolon into line **70** for good style.

```
90 NEXT DL
```

—The variable in line **90** has to match the variable name in line **80**. You could have changed either, as long as the two match.

```
100 NEXT Y
110 NEXT X
```

—The order of these two lines was reversed, so the loops incorrectly overlapped.

Chapter 18

To make this program work on the Commodore 64, you have to change three things:

1. Lines **20** through **40** are the most glaring problem. The Commodore will not accept brackets for parentheses. This is easy to fix.
2. Line **65** is the next big problem. **CLS** (which means "clear screen" to many computers) means nothing to the Commodore. You have to substitute as shown in our rewrite.
3. As we explained in the chapter, you cannot use **GOTO** with a computation or a variable—it must be a line number. Hence, we substituted the three **IF . . . THEN . . .** lines. They accomplish the same thing.

Here's how we would rewrite the program for the Commodore 64:

```
10 PRINT "$ AMOUNT TO EXCHANGE?"
14 INPUT D
20 PRINT TAB(4);"CANADIAN=1"
30 PRINT TAB(4);"ENGLISH=2"
40 PRINT TAB(4);"FRENCH=3"
50 PRINT "TYPE NUMBER"
60 INPUT C
65 PRINT"□"
70 IF C=1 THEN GOTO 200
80 IF C=2 THEN GOTO 400
90 IF C=3 THEN GOTO 600
200 PRINT D/.84;"CANADIAN DOLLARS"
210 END
400 PRINT D/1.58;"BRITISH POUNDS"
410 END
600 PRINT D/.14:"FRENCH FRANCS"
610 END
```

APPENDIX B

Tables

A. Character Code Used with CHR$ and ASC

33=!	59=;	85=U	111=┌	171=┤
34="	60=<	86=V	112=┐	172=▝
35=#	61==	87=W	113=●	173=└
36=$	62=>	88=X	114=▁	174=┐
37=%	63=?	89=Y	115=▚	175=▂
38=&	64=@	90=Z	116=▏	176=┌
39='	65=A	91=[	117=╱	177=┴
40=(	66=B	92=£	118=╳	178=┬
41=)	67=C	93=]	119=○	179=┤
42=*	68=D	94=↑	120=▗	180=▏
43=+	69=E	95=←	121=▕	181=▌
44=,	70=F	96=─	122=●	182=▐
45=-	71=G	97=♠	123=┼	183=▔
46=.	72=H	98=▏	124=▒	184=▀
47=/	73=I	99=─	125=▕	185=▃
48=0	74=J	100=▔	126=π	186=▗
49=1	75=K	101=▔	127=▚	187=▖
50=2	76=L	102=▁	162=▃	188=▝
51=3	77=M	103=▏	163=▔	189=┘
52=4	78=N	104=▕	164=▁	190=▀
53=5	79=O	105=╲	165=▏	191=▖
54=6	80=P	106=╲	166=▒	
55=7	81=Q	107=╱	167=▕	
56=8	82=R	108=▗	168=▒	
57=9	83=S	109=╲	169=▛	
58=:	84=T	110=╱	170=▕	

ALSO: 13=<RETURN> 14=<CLR/HOME> 32=<SPACE>

B. Abbreviations of BASIC Terms

```
CHR$=C|        GOTO=G¯        NEW-NONE       RND=R/
CONT=C¯        IF-NONE        NEXT=N¯        RUN=R/
DATA=D♠        INPUT-NONE     POKE=P¯        SAVE=S♠
DIM=D↘         INT-NONE       PRINT=?        SPC(=S¯
END=E/         LET=L¯         READ=R¯        STEP=ST¯
FOR=F¯         LIST=L↘        REM-NONE       TAB(=T♠
GET=G¯         LOAD=L¯        RESTORE=RE♠    THEN=T|
GOSUB=GO♠      MID$=M↘        RETURN=RE|     VERIFY=V¯
```

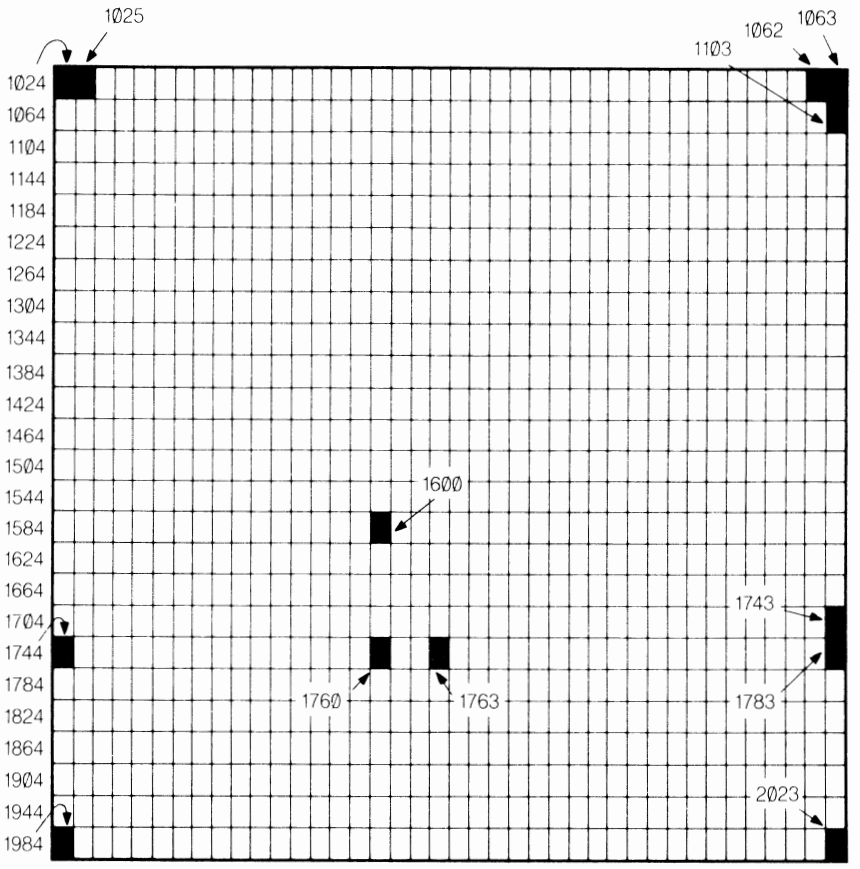
C. Screen Display Code

```
 0=@   20=T   40=(   60=<   80=¯   100=_   120=▔
 1=A   21=U   41=)   61==   81=♠   101=▌   121=▂
 2=B   22=V   42=*   62=>   82=_   102=▒   122=▁
 3=C   23=W   43=+   63=?   83=♠   103=▕   123=▖
 4=D   24=X   44=,   64=─   84=▌   104=▚   124=▝
 5=E   25=Y   45=-   65=♠   85=/   105=◤   125=▗
 6=F   26=Z   46=.   66=▌   86=X   106=▕   126=▘
 7=G   27=[   47=/   67=¯   87=O   107=├   127=▙
 8=H   28=£   48=0   68=¯   88=♠   108=▝
 9=I   29=]   49=1   69=¯   89=▌   109=└
10=J   30=↑   50=2   70=_   90=♠   110=┐
11=K   31=←   51=3   71=▌   91=+   111=▁
12=L   32=    52=4   72=▕   92=▒   112=┌
13=M   33=!   53=5   73=↘   93=▕   113=┴
14=N   34="   54=6   74=\   94=π   114=┬
15=O   35=#   55=7   75=/   95=▜   115=┤
16=P   36=$   56=8   76=L   96=    116=▌
17=Q   37=%   57=9   77=\   97=▊   117=▌
18=R   38=&   58=:   78=/   98=▃   118=▐
19=S   39='   59=;   79=¯   99=¯   119=¯
FOR REVERSE OF ANY CHARACTER, ADD 128
--MOST USEFUL IS REVERSE SPACE:160=▉
```

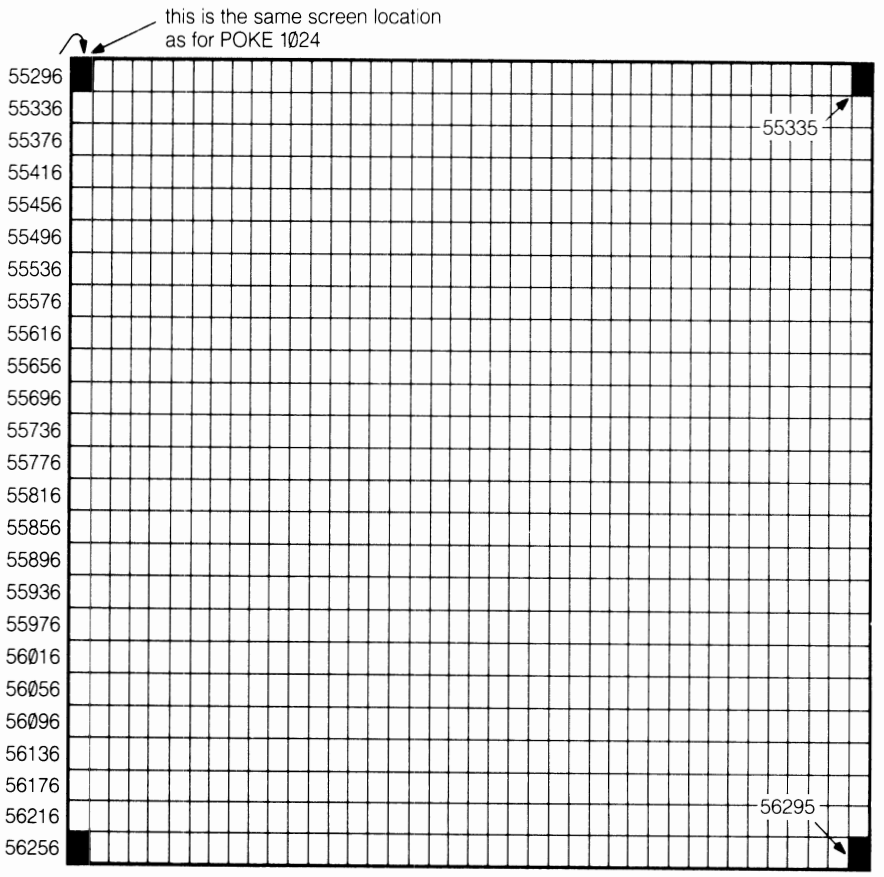
D. Color Code

CODE	COLOR	CODE	COLOR
Ø	BLACK	8	ORANGE
1	WHITE	9	BROWN
2	RED	1Ø	DARK ORANGE
3	"CYAN" (slightly greenish light-blue)	11	DARK GRAY
		12	MID-GRAY
4	PURPLE	13	MINT GREEN
5	GREEN	14	LIGHT BLUE (the normal character color)
6	BLUE (the normal screen color)		
7	YELLOW	15	LIGHT GRAY

E. Screen Location Addresses for Poking in Characters



F. Screen Location Addresses for Poking in Color



G. Notes and Frequencies

NOTE	FREQUENCY (octave before middle C)	FREQUENCY (octave beginning with middle C)
C	262	523
C#	277	554
D	294	587
D#	311	622
E	330	659
F	349	698
F#	370	740
G	392	784
G#	415	831
A	440	880
A#	466	924
B	494	988

H. POKE Addresses for Three Voices

VOICE 1

PITCH-FIRST	54272 (B)
PITCH-SECOND	54273 (B + 1)
WAVEFORM-ON/OFF	54276 (B + 4)
ATTACK/DECAY	54277 (B + 5)
SUSTAIN/RELEASE	54278 (B + 6)

VOICE 2

PITCH-FIRST	54279 (B + 7)
PITCH-SECOND	54280 (B + 8)
WAVEFORM-ON/OFF	54283 (B + 11)
ATTACK/DECAY	54284 (B + 12)
SUSTAIN/RELEASE	54285 (B + 13)

VOICE 3

PITCH-FIRST	54286 (B + 14)
PITCH-SECOND	54287 (B + 15)
WAVEFORM-ON/OFF	54290 (B + 18)
ATTACK/DECAY	54291 (B + 19)
SUSTAIN/RELEASE	54292 (B + 20)

APPENDIX C

Making Three or More Sprites

This appendix assumes you have thoroughly read and digested Chapter 14.

To write the program for the first two sprites, copy the "Handy-Dandy Sprite Maker" program on page 181 of Chapter 14, or the version in Appendix G.

Remember the following:

a. Be sure to put in your color choice for each sprite by putting in the color code after the comma in lines **50** and **51.**

b. Be sure to put in where you want each sprite to start on the screen by putting in **X** and **Y** coordinates (using the figure on page 180 of Chapter 14 as a guide) into the appropriate places after the commas in lines **60**, **61**, **70**, and **71.**

c. Design your sprites by putting **Hs** within the quotes in the spaces on the 21 data lines for each sprite.

Check to make sure the program works for two sprites.

To Make a Third Sprite

Edit the two-sprite program as shown:

```
90 POKE F+21,7
610 FOR A=1 TO 63
705 IF A/21=INT(A/21) THEN P=P+1
```

Add the following new lines:

```
32 POKE 2042,194
52 POKE F+41,0
62 XT=110
72 YT=210
402 JT=XT
422 IF XT>255 THEN G=G+4
442 IF XT>255 THEN JT=XT-255
452 POKE F+4,JT
462 POKE F+5,YT
```

Put in your desired color code after the comma in line 52.

Put in your desired starting-location X and Y numbers in lines 62 and 72.

Type 21 DATA lines—numbered 3000 to 3200—with your sprite design.

Test the program with three sprites before adding more.

To Make a Fourth Sprite

Begin with your working program for three sprites.

Edit as shown:

```
90 POKE F+21,15
610 FOR A=1 TO 84
```

Add the following new lines:

```
33 POKE 2043,195
53 POKE F+42,0
63 XF=110
73 YF=210
403 JF=XF
423 IF XF>255 THEN G=G+8
443 IF XF>255 THEN JF=XF-255
453 POKE F+6,JF
463 POKE F+7,YF
```

Put in your desired color code after the comma in line 53.

Put in your desired starting-location X and Y numbers in lines 63 and 73.

Type 21 DATA lines—numbered 4000 to 4200—with your sprite design.

Test the program with four sprites before adding more.

To Make a Fifth Sprite

Begin with your working program for four sprites.

Edit as shown:

```
90 POKE F+21,31
610 FOR A=1 TO 105
```

Add the following new lines:

```
54 POKE F+43,0
64 XV=110
74 YV=210
```

```
404 JV=XV
424 IF XV>255 THEN G=G+16
444 IF XV>255 THEN JV=XV-255
454 POKE F+8,JV
464 POKE F+9,YF
```

Put in your desired color code after the comma in line 54.

Put in your desired starting-location X and Y numbers in lines 64 and 74.

Type 21 DATA lines—numbered 5000 to 5200—with your sprite design.

Test the program with five sprites before adding more.

To Make a Sixth Sprite

Begin with your working program for five sprites.

Edit as shown:

```
90 POKE F+21,63
610 FOR A=1 TO 126
```

Add the following new lines:

```
35 POKE 2045,197
55 POKE F+44,0
65 XX=110
75 YX=210
405 JX=XX
425 IF XX>255 THEN G=G+32
445 IF XX>255 THEN JX=XX-255
455 POKE F+10,JX
465 POKE F+11,YX
```

Put in your desired color code after the comma in line 55.

Put in your desired starting-location X and Y numbers in lines 65 and 75.

Type 21 DATA lines—numbered 6000 to 6200—with your sprite design.

Test the program with six sprites before adding more.

To Make a Seventh Sprite

Begin with your working program for six sprites.

Edit as shown:

```
90 POKE F+21,127
610 FOR A=1 TO 147
```

Add the following new lines:

```
36 POKE 2046,198
56 POKE F+45,0
66 XN=110
76 YN=210
406 JN=XN
426 IF XN>255 THEN G=G+64
446 IF XN>255 THEN JN=XN-255
456 POKE F+12,JN
466 POKE F+13,YN
```

Put in your desired color code after the comma in line 56.

Put in your desired starting-location X and Y numbers in lines 66 and 76.

Type 21 DATA lines—numbered 7000 to 7200—with your sprite design.

Test the program with seven sprites before adding another.

To Make an Eighth Sprite

Begin with your working program for seven sprites.

Edit as shown:

```
90 POKE F+21,255
610 FOR A=1 TO 168
```

Add the following new lines:

```
37 POKE 2047,199
57 POKE F+46,0
67 XE=110
77 YE=210
407 JE=XE
427 IF XE>255 THEN G=G+128
447 IF XE>255 THEN JE=XE-255
457 POKE F+14,JE
467 POKE F+15,YE
```

Put in your desired color code after the comma in line 57.

Put in your desired starting-location X and Y numbers in lines 67 and 77.

Type 21 DATA lines—numbered 8000 to 8200—with your sprite design.

Test out the program.

You can only have eight sprites.

Keyboard Control and Animation of More Than Two Sprites

You can put as many sprites as you like under keyboard control. You cannot, however, use the number keys for this purpose, because there are only 10 (and the **Ø** would be eliminated too, since it would be confused with what happens when you don't **GET** a key at all). To have full keyboard control of movement, you need 4 keys for each sprite—a total of 32 if you have eight sprites!

The solution is to use letter keys. Your **GET** lines will all have to work with a string variable, even for your first sprites, which you may leave controlled by numbers by doing something like the following:

```
92 GET LR$
93 IF LR$="" THEN GOTO 92
94 IF LR$="8" THEN X=X-1
```

For the later sprites, you would pick letters. For example, the up movement for sprite #3 could be signaled by the [U] key:

```
104 IF LR$="U" THEN XT=XT-1
```

Of course, as you add more lines at the end, you have to move the final **GOTO 8Ø** line further down by giving it a higher number, for example, with **15Ø GOTO 8Ø**.

To make it easy to list and view your program, you might want to put all the **GET** and associated **IF...THEN...** lines in a subroutine. Thus, line **92** would become something like **92 GOSUB 2ØØ**, and all the **GET** lines would start with **2ØØ** and end with a **RETURN** line.

While it is possible to have keyboard control over as many sprites as you want, it isn't very practical in most cases. Who can remember which keys for which sprite for which direction when you get beyond two or three sprites?

A better solution for an interesting game is to have two or three keyboard-controlled sprites (or two or three maximum for each player, if there is more than one player), and animate the rest in some way. You can use the **RND(1)** procedure to create interesting locations and animation effects for your sprites in a situation like this.

APPENDIX D

Glossary

There is a separate glossary of BASIC instructions in Appendix E. The glossary here covers other words that you might not be familiar with.

The numbers in parentheses are the numbers of the chapters where the term comes up and where you can usually learn more about it.

Address—The number of a location in the computer's memory that controls some operation—screen color, for example. These numbers are used with the **POKE** command. (12)

Array—A name for all the variables with the same subscript. (9)

Arrow keys—Same as cursor-arrow keys, or directional keys, or cursor-movement keys. These two keys allow you to move the cursor up or down or right or left without erasing or otherwise interfering with what is on the screen. You can also use these keys to put special markers into a character string (within quotes) that will be carried out when the character string is printed. (3,4)

Attack rate—The time it takes for a tone to go from silence to its maximum volume. You can set the attack rate of a sound produced by the Commodore 64 sound generator between zero and eight seconds. (15)

Bug—An error or problem that keeps a computer program from working properly. (17)

Carriage return—The key on an electric typewriter (or handle on a manual) that you press to make the typing start a new line. On a computer, the [RETURN] key takes the cursor back to the beginning of the next line. But it also tells the computer to carry out a command or enter a program line into memory, so it must be used with care. The computer automatically takes you to the next line if you type to the edge of the screen. (3)

Cartridge (or *Program cartridge*)—A plastic box the size of a large bar of soap that plugs into the back of your computer, putting a prepackaged program into it. (2)

Character string—A group of letters or other characters (symbols, numbers, graphic characters, and so forth) you put together (usually within quotes) to be remembered or printed. (4,10)

Character code—Numbers that stand for the individual letters and other characters the computer can produce. You find the character code of any character by typing **PRINT ASC** followed by the character (within quotes and parentheses). (11)

Code number—A nontechnical term we've adopted to refer to the numbers you poke into computer addresses to make various

things happen. For example, to make the screen black, you poke the code number **0** into the address for screen color. (12)

Color code—For the colors of the screen and border, the characters poked in, and sprites, there is a code of the 16 available colors. A table showing the code is in Appendix B. (12)

Command—An instruction to the computer. Also used to distinguish a command given directly (such as **PRINT 40+5**) versus an instruction given as part of a program. (It is also used technically to refer to a certain subgroup of BASIC words that are often used outside of programs—**RUN**, **LIST** etc.) (4,5)

Command mode—Using the computer without numbered program lines to give direct commands, such as **PRINT 40+5**. (4,5)

Commodore key—The lower left key on the Commodore 64 with the Commodore corporate symbol [ ] on the front. It is used mainly in connection with the graphic characters and in connection with loading programs. (2,3,6)

Concatenating—Putting together two character strings. (10)

Counter—A program line, such as **N=N+1**, that gets one larger each time it is passed in a loop. Thus, it keeps count. (17)

Cursor-Arrow key—See *Arrow key.*

Cursor-Movement marker—See *Arrow key.*

Cyan—A light greenish blue. One of the 16 colors available on the Commodore 64. (3,12)

Debugging—Getting the problems and errors out of a program. (17)

Decay rate—The time it takes a tone to go from its maximum volume to a more moderate volume. The Commodore 64 sound generator permits you to set the decay rate anywhere from 0 to 24 seconds. (15)

Database management—Refers to computer programs that file and sort information. (2,9)

Datassette™—A cassette tape player designed especially to save and load programs on Commodore computers. (2,6,19)

Delay loop—A **FOR . . . TO . . .** and **NEXT** loop that serves no other purpose than to create a pause in the running of a program. (10)

Directional key—See *Arrow key.*

Direct mode—See *Command mode.*

Disk—A magnetic material packaged in a 5-1/4 inch envelope used with a disk drive that saves and loads programs more efficiently than cassette tapes. (2,6,19)

Disk drive—A small machine that "plays" and "records" programs with your disks. (2,6,19)

Diskette—See *Disk.*

Documentation—The written instructions and explanations that come with prepackaged programs. (2)

Dot matrix printer—A computer printer that forms its letters with many small dots. (2,8,19)

Duration—The length of time a tone lasts—usually determined by a delay loop. (15)

Empty string—A character string with no content. It is to characters what **0** is to numbers. It is represented by two quotes with

nothing in between—" ". When you use **GET** to get a string variable, the computer gets an empty string while it is waiting for your key press input. (13)

Endless **GOTO** loop—A loop in which a **GOTO** instruction sends the computer back to some place in the program from which it eventually comes again to the **GOTO** line in a never ending circle. (5)

Error message—A short phrase the computer displays on the screen when you have done something it cannot understand. It tries to tell you what the problem is. (17, Appendix F)

Exponent—A number that indicates the number of times a number is to be multiplied by itself. "Ten squared," means "10 with an exponent of 2."

Files—Loosely speaking, this means information organized in some way. The term also refers to a specific way of saving information on tape or disk. (9,17)

Frequency—Technically, this means the number of vibrations to create a sound per second. It corresponds to the pitch—how high or low a sound is. For example, the A below middle C has a frequency of 440 cycles per second. (15)

Graphic character—One of the little figures in the boxes on the fronts of the keys. They are made to appear on the screen by holding either the Commodore key (for the left little picture) or the [SHIFT] key (for the right little picture). (3)

"Handy-Dandy Sprite Maker"—A program we have written to make it easy for you to make sprites. (14)

Hard disk—A way of saving programs and memory that is used on very large computers. (2)

Hardware—The machinery—computer and accessories—as opposed to "software"—programs and prepackaged programs. (2)

Home—The top left corner of the screen. The cursor returns there when you clear the screen or press the [CLR/HOME] key without [SHIFT]. (3)

Infinite loop (or *Infinite* **GOTO** *loop*)—See *Endless loop.*

Instruction—A term we use to refer to various BASIC words (such as **PRINT** and **GOTO**) that tell the computer to do something. We also use the term to refer to the entire command or program line that contains the word. (3,4, and throughout the book)

Integer—A whole number. (10)

Joystick—An accessory for your computer that allows you to give input to your computer by moving a stick forward, backward, sideways, etc. It is used a lot in games.

Letter-quality printer—A computer printer that produces fully formed characters like a typewriter's (as opposed to those made of dots). (2,19)

Loop—A part of a program that circles through itself two or more times. (5,6)

Machine language—A way of giving instructions to the computer that works directly with its electronics. There are many esoteric things you can do with machine language that are not possible in BASIC. It is also much harder to use unless you know it very well. (2,19)

Memory—The aspect of the computer's functioning that keeps track of information and numbers. (1)

Modem—A device that hooks your computer up to a telephone so that you can communicate with other computers, including information services. (19)

Monitor—A special TV designed to give sharper images for computer-generated letters. (2,19)

Peripherals—Accessory machines—such as a Datassette, monitor, disk drive, modem, etc.—that you use with your computer. (2,19)

Pitch—The "high" or "low" aspect of a sound that makes a soprano different from a bass. See also *Frequency*. (15)

Print zone—A 10-space-long segment of the Commodore 64 screen line. There are four across the screen. A comma in a **PRINT** command makes the next item typed appear in the next print zone. (8)

Program mode—Using the computer with numbered program lines, which are instructions it remembers and carries out in the order of their numbers when you run the program. (5)

Prompt—A **PRINT** instruction that puts a message on the screen asking the user to type something, while the computer waits at an **INPUT** line. (6)

*Prompt-and-**INPUT** sequences*—The usual sequence of using prompts and the **INPUT** instruction. First comes the **PRINT** line with the prompt; then, immediately after, comes the **INPUT** line. Often, a clear-screen line follows to remove the prompt from the screen when it is no longer needed. (6)

Quality—A term we use to refer to the aspect of sound that differentiates even the same note played for the same duration at the same volume, for example, the difference between a flute and a xylophone. (15)

Release rate—The time it takes a tone to go from its sustained volume to silence, once it has been turned off. The settings can vary from 0 to 24 seconds. (15)

Reverse characters—Letters or other characters written on the screen in dark letters against a light background—the reverse of the usual. (3)

Reverse-heart figure—The marker the computer puts in a **PRINT** line (within quotes) to let you know that, when it carries out the **PRINT** instruction, it will clear the screen.

Scientific notation—A system of writing large numbers in which a number has only one place to the left of the decimal. The size of the number is indicated by how many times you would have to multiply it by 10 to get the original number. The computer uses this when it cannot fit the usual form of the number on the screen. It displays an **E**, followed by a dash and the number of times by which to multiply it by 10. (7)

Screen display code—A system of numbers that stand for letters and other characters. It is used when poking in designs on the screen. Although the principle is similar, the numbers are mostly different from the character codes used with **CHR$** and **ASC**. (12)

Screen line—The space for a horizontal row of letters across the screen. The Commodore 64 has 25 screen lines.

Spreadsheet program—A program that makes your screen into a columnar pad on which you can lay out, compute, and display financial activities. (2)

Software—Prepackaged sets of instructions for your computer. You simply plug in the cartridge, tape, or disk. It is called "software" as opposed to "hardware" (the machinery)—the other component of a computer system. (2)

Sprite—A small design you can program and then animate or make keyboard-operated. (14)

Statement—A numbered program line, or certain BASIC words, such as **INPUT** or **IF . . . THEN . . .**. (17)

String—See *Character string.*

String slicing—Taking out a segment of a long string. Usually done with **MID$**. (10)

Subscript—The number in parentheses after a variable. See *Subscripted variable.*

Subscripted variable—A variable with a number (or another variable) after it in parentheses, such as **X(3)**. It is an indispensable tool in programs that keep track of a lot of information. (9)

Subroutine—A miniprogram—a few program lines—used one or more times by the computer in carrying out the main program. It is written with **GOSUB** and **RETURN**. (11)

Sustain level—The proportion of the peak volume at which a tone is maintained after it declines from its initial peak. You can set this from full volume to no volume. (15)

White noise—A quality of sound produced on the Commodore 64 that is very useful for sound effects and that simulates the sound of percussion instruments. (15)

Word processing—Programs that make your computer work like a typewriter with special advantages—corrections are much easier, and instead of typing something out in a particular format, you only specify the format and the computer will carry out your instructions when printing. (2)

Variable—A letter or name that stands for a number. A string variable could also stand for a string of letters or other characters. (6,7)

Variable name—One or more letters that identify a variable. On the Commodore 64, you can use any letters you choose, although it only uses the first two letters if the name is longer than that. Numbers (after the first letter) are okay. The variable name cannot be the same as or have the same first two letters as a BASIC keyword. (For example, **PRIME** and **PRINT** would be confused by the computer.) (6)

Variable table—A layout, line by line, of what happens to the different variables as the computer proceeds through the program. (7,10)

APPENDIX E

BASIC Glossary

Appendix D has a complete glossary of other terms used in the book.

The Commodore 64 BASIC vocabulary is made up of 71 words. You have learned about 30 of them in this book. This appendix covers only those included in the book (and not those mentioned only in a footnote or box). Since many of them came up in later chapters that you may have skipped, you may not recognize all of them. Remember, you don't need to know a lot of BASIC words—just the right ones. (In parentheses you'll find the chapter number where the term was introduced.)

ASC—Gives you the code number for any character. (11)

CHR$—Gives you the corresponding character for a code number. (11)

CONT—Starts a program running again from where it left off when you pressed the [RUN/STOP] key. (5)

DATA—Lines that provide information to be taken into the program by a **READ** line. (9)

DIM—(short for *Dimension)*—Sets the amount of space to put aside in memory for subscripted variables. (9)

END—Ends a program. (11)

FOR . . . TO . . . , STEP, *and* **NEXT**—A **FOR . . . TO . . . , STEP** line sets a variable equal to a starting number (specified after the **FOR**). When the computer arrives at the corresponding **NEXT** line, it is sent back to the **FOR . . . TO . . .** line, where it sets the variable one step higher. (If no step size is specified, it is assumed to be 1.) This process continues until the variable has reached the upper limit specified after the **TO**. Then the computer goes through the subsequent lines one last time but does not return to the **FOR** line when it comes to the **NEXT** line. It simply proceeds to the lines that follow the **NEXT** line. (6)

GET—Sets the value of a variable equal to whatever key you press. If you don't press a key, the variable is set equal to **Ø** if it is a numeric variable, or to an empty string (" ") if it is a string variable. To avoid this, you usually have a line following the **GET** line to test for it: For example, **5ØØ GET K** would be followed by **51Ø IF K=Ø THEN GOTO 5ØØ**. (13)

GOSUB—Sends the computer to a subroutine specified in the **GOSUB** line. You must use an actual line number—a variable or computation will not be accepted. (11)

GOTO—Sends the computer to a line number specified in the **GOTO** line. You must use an actual line number—a variable or a computation will not be accepted. (5)

INPUT—Stops the flow of the program and waits for the person at the keyboard to type something which is then assigned to the variable specified in the **INPUT** line, and the computer continues with the rest of the program. (6)

INT—Rounds down a decimal number into a whole number by dropping the decimals. (10)

LET—Gives a value to a variable—as in **LET X=X+1**. However, the use of the actual word **LET** is optional—**X=X+1** by itself would be just as good. (7)

LIST—Commands the computer to give a list of all program lines in memory. You can also specify that it produce only lines within a particular range, for example **LIST 1000–12000**. (5,14)

LOAD—Commands the computer to put your program from a cassette or disk into the computer's memory. (2,6)

MID$—Lets you take part of a string. To use it, put three things in parentheses immediately following it—the variable whose character string you want to slice a piece out of, the number of the space in that string to begin slicing, and the total number of characters to take. (10)

NEW—Commands the computer to erase all program lines from its memory. (7)

NEXT—See **FOR...TO...**, **STEP**, *and* **NEXT**.

POKE—Puts some code number into a location in the computer's internal functioning. First, you give the address of the location, then, after a comma, the code number. For example, **POKE 53281,0** makes the screen black. (12)

PRINT—Displays on the screen whatever you put after it—either the result of a numeric calculation, or the exact character string you specify within quotes. (4,5,8)

READ—Takes information from **DATA** lines (elsewhere in the program) and assigns them to one or more variables specified in the **READ** line. (9)

REM—A remark. A **REM** line is maintained in the listing for notes to anyone looking at the program, but does not in any way affect the carrying out of a program. (6)

RESTORE—Tells a **READ** line to start taking information again from the very first **DATA** line in the program. (13)

RETURN—Used at the end of a subroutine to send the computer back to the line immediately following the **GOSUB** that sent it to that subroutine. (11)

RND(1)—Produces a random number between **0** and **1**. (10)

RUN—Commands the computer to carry out the instructions in the program in memory. It first sets any variables previously defined to **0**. (5)

SAVE—commands the computer to put a program in memory onto tape or disk. (6)

SPC—Used in a **PRINT** line to specify a certain number of spaces to insert when the **PRINT** instruction is carried out. (8)

STEP—See **FOR...TO...**, **STEP**, *and* **NEXT**.

TAB—Used in a **PRINT** line to specify a particular column number across the screen where the next item to be printed should begin. (8)

APPENDIX F

Commodore 64 Error Messages

Many of the error messages apply to advanced procedures you would not be likely to use. Most errors are just typographical errors that the computer interprets as some sophisticated, subtle error. If you make a mistake, check your typing. Then check whether you have left out a line or included any from an old program you forgot to eliminate.

The various error messages (and the steps for debugging a program) are discussed at length in Chapter 18. This appendix provides a brief description of those error messages you are likely to run into.

?SYNTAX—A catchall. It covers any problem that doesn't have a more specific error message. Usually, it results from a typing mistake (as do most computer errors), misplaced punctuation in a **PRINT** instruction, or a missing parenthesis.

?BAD SUBSCRIPT—Means you used a higher-numbered subscript than you dimensioned for (or a subscript higher than **10** if you did not dimension at all).

?CAN'T CONTINUE—Means you tried to use the **CONT** command when it was not allowed—probably because the program stopped due to an error, or you edited a program line.

?DEVICE NOT PRESENT—Arises when, intentionally or unintentionally, you give a command involving a printer, Datassette, or some other device that is not properly hooked up to the computer (or is not turned on).

?DIVISION BY ZERO—Not allowed in arithmetic—and certainly not on the Commodore 64. This often happens unexpectedly—for example, when you are dividing somewhere in your program by a variable that happened to have come out equal to **0** one time you used the program.

EXTRA IGNORED—Arises when, in response to an **INPUT** line, you put in too many numbers (or too many characters in a string). The computer takes as many as it needs to fill the variables in the **INPUT** line and ignores the rest—but it kindly lets you know.

?ILLEGAL DIRECT—Means you used an instruction in command (or direct) mode that can be used only in numbered program lines.

?ILLEGAL QUANTITY—Means you used a number in a **POKE** or other instruction that is outside the allowable range for that instruction. For example, character codes only go up to **255**—thus, **CHR$(300)** would produce this error message.

?NEXT WITHOUT FOR—Whenever you have a **NEXT** line, it must be preceded in the program by a **FOR . . . TO . . .** line. This error most often occurs when, in editing programs, you take out a **FOR . . . TO . . .** line and forget to take out the **NEXT** line that goes with it.

?OUT OF DATA—Means the computer came to a **READ** instruction but there were no **DATA** lines in the program, or the **DATA** lines had already been used up by previous **READ** instructions. (This error message often appears when you accidentally press [RETURN] while the cursor is on a screen line that says **READY**. It interprets it as a command to read a value for the variable **Y**. Of course, there are no **DATA** lines from which it could read a value for **Y**. When this happens, ignore it. It doesn't hurt anything.)

?OUT OF MEMORY—Doesn't happen often on the Commodore 64. If it does, you probably accidentally set a very large number of variables equal to something, or used very large **DIM** lines.

OVERFLOW—Occurs in that rare case when you have given the computer a calculation that results in a number too large for it to handle. It would have to be a number with more than 38 zeros after it!

?REDIM'D ARRAY—Means the computer came a second time to a **DIM** line for the same subscripted variable.

REDO FROM START—Means that, in typing information in response to an **INPUT** line, you did something funny—gave letters instead of numbers or accidentally hit something bizarre. This error message gives you a chance to reenter your data.

?RETURN WITHOUT GOSUB—Means the computer came to a **RETURN** line without having first come to a **GOSUB** line. There may be a **GOSUB** line in your program—but that doesn't help. This error often occurs when you fail to put in an **END** line and the computer continues on into the subroutines.

?STRING TOO LONG—Means you told the computer to remember a character string of more than 255 characters. This is very rare.

?SYNTAX—The most common error message, so we treated it out of alphabetical order. It is listed at the beginning of this glossary.

?TYPE MISMATCH—Means you tried to set a numeric variable equal to a character string, or vice versa.

?UNDEF'D STATEMENT—Means a **GOTO** or **GOSUB** instruction sent the computer to a nonexistent line number.

APPENDIX G

"The Handy-Dandy Sprite Maker"

The following program is the one that you worked on in Chapter 14, with one exception. This program allows you to control your sprites' movements both right and left *and* up and down.

To make two sprites—

1. Copy lines **10** to **720** of the following program exactly as shown.
2. Put in your own sprite designs by using **Hs** in **DATA** lines **1000** to **1200** (for sprite #1) and **2000** to **2200** (for sprite #2). (See the **DATA** lines in the following program for examples.)
3. Set colors for the sprites by putting the code numbers for the desired colors at the end of lines **50** (for sprite #1) and **51** (for sprite #2).
4. Run the program. Use the number keys [1] through [4] to control sprite #1, and [6] through [9] to control sprite #2.

```
10 PRINT"□"
20 F=53248
30 POKE 2040,192
31 POKE 2041,193
40 GOSUB 600
50 POKE F+39,1
51 POKE F+40,0
60 X=180
61 XS=110
70 Y=210
71 YS=210
80 GOSUB 400
90 POKE F+21,3
92 GET LR
93 IF LR=0 THEN GOTO 92
94 IF LR=8 THEN X=X-1
95 IF LR=9 THEN X=X+1
96 IF LR=6 THEN Y=Y-1
97 IF LR=7 THEN Y=Y+1
98 IF LR=1 THEN XS=XS-1
99 IF LR=2 THEN XS=XS+1
100 IF LR=3 THEN YS=YS-1
```

```
 101 IF LR=4 THEN YS=YS+1
 102 GOTO 80
 400 J=X
 401 JS=XS
 410 G=0
 420 IF X>255 THEN G=G+1
 421 IF XS>255 THEN G=G+2
 430 POKE F+16,G
 440 IF X>255 THEN J=X-255
 441 IF XS>255 THEN JS=XS-255
 450 POKE F,J
 451 POKE F+2,JS
 460 POKE F+1,Y
 461 POKE F+3,YS
 470 RETURN
 600 P=12287
 610 FOR A=1 TO 42
 620 READ D$
 630 FOR B=0 TO 2
 640 K=0
 650 FOR C=1 TO 8
 660 IF MID$(D$,B*8+C,1)="H" THEN K=K+2↑(8-C)
 670 NEXT C
 680 P=P+1
 690 POKE P,K
 700 NEXT B
 705 IF A=21 THEN P=P+1
 710 NEXT A
 720 RETURN
1000 DATA "           HHH                "
1010 DATA "    HH     HHHHH              "
1020 DATA "    H      HHHHH              "
1030 DATA "    H        H                "
1040 DATA "    HHHHHHHHHHHHHHHHHHHHHH"
1050 DATA "           HHHHHH             "
1060 DATA "            HHHH              "
1070 DATA "              HH              "
1080 DATA "              HH              "
1090 DATA "       HHHHHHHHHHHHHH         "
1100 DATA "        HHHHHHHHHHHH          "
1110 DATA "         HHHHHHHHHH           "
1120 DATA "             HH               "
1130 DATA "             HH               "
1140 DATA "             HH               "
1150 DATA "             HH               "
1160 DATA "             HH               "
1170 DATA "             HH               "
1180 DATA "             HH               "
1190 DATA "             HH               "
1200 DATA "             HH               "
2000 DATA "           HHH                "
```

```
2010 DATA "           HHH                "
2020 DATA "           HHH                "
2030 DATA "H           HH                "
2040 DATA "HHHHHHHHHHHHHHH             H"
2050 DATA "           HHHHHHHHHHHHHHHH"
2060 DATA "           HHHH               "
2070 DATA "           HHHH        HHHH   "
2080 DATA "            HHH        HH     "
2090 DATA "            HHH       HHH     "
2100 DATA "            HHHH      HHH     "
2110 DATA "            HHHH       HH     "
2120 DATA "            HHHHHHHHHHHH      "
2130 DATA "             HHHHHHHHHH       "
2140 DATA "             HHH              "
2150 DATA "             HHH              "
2160 DATA "             HHHH             "
2170 DATA "              HHH             "
2180 DATA "              HHH             "
2190 DATA "              HH              "
2200 DATA "            HHHH              "
```

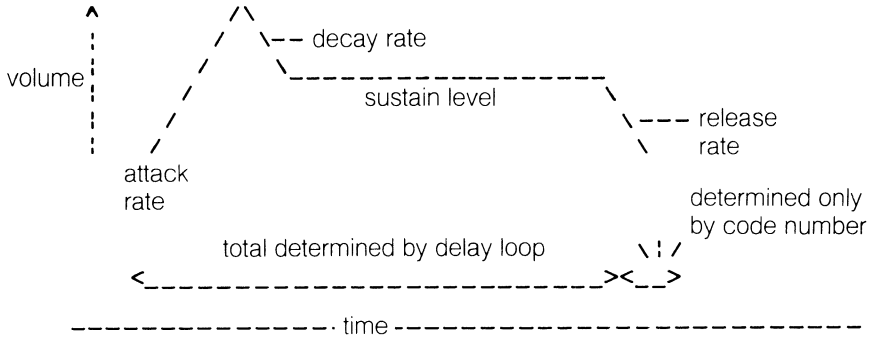
APPENDIX H

More about Sound Quality

The way attack/decay and sustain/release work is slightly complicated—but read carefully and you'll get it.

This component of the quality of a sound clearly divides the time the note is audible into four time periods. Each period is distinguished by a different volume and takes up a different period of time.

So, we have two things to consider for each period—length of time and volume. We can make a picture (or graph) of time and volume. The volume would be how high the line is. How far it moves to the right would indicate how much time has passed. Then we can divide that line into these four sections or time periods. Look at the following figure as we describe these four time periods.



Attack: During this period, the volume rises to its full or peak volume (remember, volume was determined by address **B+39** and the code that went from **Ø** to **15**).

Decay: During this period, volume settles from the peak to some more moderate level.

Sustain: During this period, volume stays at this moderated level.

Release: During this period, volume drops from the moderated level to silence.

The length of the first three periods *as a unit* is determined by the **FOR . . . TO** and **NEXT** delay loop. Of these three, the length of the first two is set by poking a code (which we'll get to in a moment) that determines attack rate and decay rate—and the sustain period gets the leftovers. If you set attack and decay to take up all the time or set a short duration to the entire note, you could eliminate the sustain period. If the attack takes up a lot of time or the note is very short, the decay period could even be cut short or eliminated. All these possibilities obviously affect the sound quality.

The piano-like sound in the program we use in this chapter has almost no attack time and about three-quarters of a second of decay. The sustain level is **Ø**, and the release rate (which was actually irrelevant, as there was nothing to release from a **Ø** sustain level) is also **Ø**. Thus, the pattern in this case is something like the following figure:

Now, more about the sustain period. Since the duration of the sustain period is determined by how much time is left for it after the first two periods are finished, the only thing to control about the sustain period is its volume. Thus, all we tell the computer about this period is the volume—the sustain *level.* Sustain level is expressed as a proportion of the volume at its peak, but that's not too important to remember. Just remember that the higher the sustain level, the louder the sound during this part of the note.

All that's left is the release rate. Like attack and decay, volume for the release period is already determined. (Remember? For attack, volume starts at **Ø** and goes to the volume poked at address **B+39**; for decay, volume starts where attack leaves off and drops to the sustain level. Again, the only "unknown" volume is the sustain level, which you poke in.) Release volume starts at the sustain level and drops to silence. So the only question is how long does the drop take?

What then does the computer need to know about these four time periods? To review:

Attack rate: How long the sound takes to rise to its peak volume at the beginning.

Decay rate: How long it takes to settle down to a more moderate level from that peak volume.

Sustain level: The level of volume at which the sound is sustained (expressed in terms of the proportion of the peak volume).

Release rate: How long it takes to drop to silence from the level it was sustained at.

Two **POKE** addresses determine the four items we have just listed. Attack and decay are controlled by **POKE** address **54277** (**B+5**). Sustain and release are controlled by **POKE** address **54278** (**B+6**).

1. The decay settings go from **Ø** (almost no decay time at all) to **15** (24 seconds).

2. Attack uses the *same* **POKE** address as decay—**54277** (**B+5**). But its settings go from **Ø** (almost instantaneous rise to peak volume) to **24Ø** (8 seconds to rise to peak volume), *in increments of* **16**. That means the settings available for attack are **16**, **32**, **48**, **64**, **8Ø**, and so forth, up to **24Ø**.

You combine attack rate and decay rate settings into a single number for their single, shared **POKE**. Just add the values for attack and decay. Since the attack values go up by multiples of **16**, if you add any number between **Ø** and **15** to them, it is always clear which is which. For example, if you want a very slow decay of **15** and a very slow attack of **24Ø**, you would poke **B+5,255**. If you want a short attack and a short decay, you would poke the sum of **1** and **16**—**B+5,17**.The following table illustrates this:

CODE NUMBER	DECAY RATE	ATTACK RATE	CODE NUMBER	DECAY RATE	ATTACK RATE
Ø	zero decay	instantaneous attack	32	zero decay	a little more attack time
1	"	"			
"	"	"	"	"	"
"	"	"	"	"	"
14	"	"	"	"	"
15	longest decay	instantaneous attack	"	"	"
			"	"	"
16	zero decay	a little attack time	"	"	"
			"	"	"
17	"	"	239	longest decay	very long attack time
"	"	"			
"	"	"	24Ø	zero decay	longest attack time
31	longest decay	a little attack time	241	"	"
			"	"	"
			"	"	"
			255	longest decay	longest attack time

3. The code for sustain level and release rate is created the same way. Sustain-level settings go from **Ø** to **15**, and release-rate settings go from **Ø** to **24Ø** (24 seconds) by increments of **16**. (Note that the release time begins when the tone is turned "off," so unless you put in a delay loop to leave time *between* notes, or it is the last or only tone played, it is irrelevant. The computer pokes in the next note right away and the release of the previous one is cut off.

This time the sum of the two code numbers is poked into address **5278** (**B+6**).

Now, you can get a feel for what different settings do by trying different values for one, while keeping all the others constant. To try different attack rates, add to the program you have been using in the chapter a prompt-and-input sequence (such as **24 PRINT "TYPE ATTACK RATE"** and **25 INPUT A**) that sets the attack code (**3Ø POKE, B+5,A**). Then try values from **1** to **15**. Then you could try decay values (**16** to **24Ø** in increments of **16**).

Next, set the attack/decay value to **Ø** and put in the same kind of program lines to input various sustain and release values. (In this case, be sure to add a second delay loop (such as **115 FOR K=1 TO 1ØØØ** and **116 NEXT K**) to allow time before the next note is played to hear the effect of different release rates.)

Ideally, you would hear all the combinations of these four parts of *this* sound quality, plus all the *other* components of sound quality, plus the other three aspects of sound, each with each other because, combined, they produce a whole which is much more than the sum of their parts. That, however, would take a long, long time. To get just a taste of the possibilities, try the following program, which randomly selects values for the various settings for sound quality only (both wave pattern and attack/decay and sustain/release). You can either type in this program from scratch or edit the one you have been using.

Run the program. Let it go for a while and make note of the settings for any interesting sounds.

After you have run the program as it was, change the note to a very low note and run it. Then try a very high note, and run it again.

APPENDIX I

Quick-Reference Guides

The information in this Appendix is provided to give you a fast guide to using BASIC terms, and a quick-reference guide to sound and graphics.

A Quick-Reference Guide to Using BASIC Terms

To get information *in,* use—

1. **READ** and **DATA**. Put information on **DATA** lines. When computer comes to **READY Y** (for example), it reads the first unread **DATA** line item and assigns it to **Y**.
2. Prompt-and-**INPUT**. Have computer ask for data in prompt. Respond by typing in data that will be assigned to the variable named in the **INPUT** line (for example, **10 INPUT Y**).
3. **GET**. Like **INPUT**, but you don't press [RETURN]—and it takes only one key press.
4. Write information directly into program, such as making a variable equal to your information (for example, **40 F=440**).

To transform information, use—

1. **LET**. Sets a variable equal to something, such as a conversion formula. (The word **LET** isn't needed: **30 M=F*.3048** is okay by itself.)
2. **IF . . . THEN . . .** . Evaluates a comparison. If it's true, it does the **THEN** part (for example, **IF X=1 THEN B=B+AM**).
3. **ASC** and **CHR$. ASC** transforms characters to character code numbers, **CHR$** does the opposite.
4. **RND(1)**. Selects a random number between **0** and**1**.
5. **MID$**. Slices out a section of a character string.

To control order of operations, use—

1. **GOTO**. Changes normal order, especially to repeat all or part of program.
2. **IF . . . THEN . . .** . Decides where to go next depending on value of some variable.

3. **FOR . . . TO . . . , STEP**, and **NEXT**. Determines number of times through a loop (progressively changes a variable). Also used to delay a program.
4. **GOSUB** and **RETURN**. Sends computer to a subroutine, and then returns it to the program line right after the **GOSUB** instruction.
5. **END**. Signals computer to stop running program. Especially useful to avoid continuing into subroutines.

To get information *out*, use—

1. **PRINT**. Displays results of calculations or character strings on the screen. Use quotes around nonnumbers; use commas or semicolons as separators.
2. **POKE**. Changes internal functioning of computer. Use to make complex graphics, sprites, sound, or animated output.

A Quick-Reference Guide to Sound and Graphics

For screen and border colors—

1. To change the screen color, **POKE** code **53281**. Follow it with a comma, then insert the code for the desired color (for example, **2Ø POKE 53281, Ø** for black).
2. To change the border color, use **POKE** code **5328Ø**. Follow it with a comma, then insert the code for the desired color (for example, **3Ø POKE 5328Ø, 1** for white).

Codes for Screen and Border Colors

CODE	COLOR	CODE	COLOR
Ø	BLACK	8	ORANGE
1	WHITE	9	BROWN
2	RED	1Ø	DARK ORANGE
3	"CYAN" (slightly greenish light-blue)	11	DARK GRAY
		12	MID-GRAY
4	PURPLE	13	MINT GREEN
5	GREEN	14	LIGHT BLUE (the normal character color)
6	BLUE (the normal screen color)		
7	YELLOW	15	LIGHT GRAY

For making designs on the screen—

1. Copy lines **10** to **70** of the following program exactly as shown.
2. Think of the screen as having spaces across from Ø to 39, and rows down from Ø to 24. Draw your design on graph paper. Then put into **DATA** lines data for your own design by putting in, for each space, its column number (from **Ø** to **39**), row number (from **Ø** to **24**), and the screen display code for the desired character (see the table that follows the program for character screen display codes). End your **DATA** lines with **−1**, **−1**, **−1**. (See the examples of **DATA** lines in program lines **1000** to **1050**.)

```
10 PRINT "■"
20 READ X,Y,SDC
30 IF X<0 THEN GOTO 30
40 PL=1024+X+Y*40
50 POKE PL,SDC
70 GOTO 20
1000 DATA 17,13,95,18,14,95,18,13,123
1010 DATA 19,14,76,20,8,103,20,9,103
1020 DATA 20,10,103,20,11,103,20,12,103
1030 DATA 20,13,103,20,14,122,21,14,122
1040 DATA 22,13,122,22,14,105,23,13,105
1050 DATA -1,-1,-1
```

Screen Display Code

```
 0=@   20=T   40=(   60=<   80=┐  100=_  120=▀
 1=A   21=U   41=)   61==   81=●  101=▌  121=▄
 2=B   22=V   42=*   62=>   82=▗  102=▒  122=▟
 3=C   23=W   43=+   63=?   83=▖  103=▝  123=▙
 4=D   24=X   44=,   64=─   84=▌  104=▚  124=▘
 5=E   25=Y   45=-   65=♠   85=▝  105=◤  125=▜
 6=F   26=Z   46=.   66=▏  86=╳  106=▕  126=▝
 7=G   27=[   47=/   67=▔  87=○  107=├  127=▖
 8=H   28=£   48=0   68=▔  88=♣  108=▗
 9=I   29=]   49=1   69=▔  89=▕  109=└
10=J   30=↑   50=2   70=▁  90=♥  110=┐
11=K   31=←   51=3   71=▏  91=┼  111=▁
12=L   32=    52=4   72=▕  92=▚  112=┌
13=M   33=!   53=5   73=╲  93=▕  113=┴
14=N   34="   54=6   74=╲  94=π  114=┬
15=O   35=#   55=7   75=╱  95=┘  115=┤
16=P   36=$   56=8   76=└  96=    116=▏
17=Q   37=%   57=9   77=╲  97=▌  117=▌
18=R   38=&   58=:   78=╱  98=▄  118=▐
19=S   39='   59=;   79=┌  99=▔  119=▔
FOR REVERSE OF ANY CHARACTER, ADD 128
--MOST USEFUL IS REVERSE SPACE:160=■
```

To play a tune—

1. Copy lines **10** to **120** of the following program exactly as shown.
2. For each note that you want to play, put into **DATA** lines its frequency number, followed by its time (for quarter notes put **4**, for eighth notes put **8**, etc.). See the examples of **DATA** lines in program lines **500** and **510**.

```
10 B=54272
20 POKE B+24,15
30 POKE B+5,9
40 POKE B+6,0
50 READ F,T
60 POKE B,F/.06097-(256*INT(F/15.6083))
70 POKE B+1,F/15.6083
80 POKE B+4,33
90 FOR D=1 TO 1000/T
100 NEXT D
110 POKE B+4,32
120 GOTO 50
500 DATA 392,4,392,4,440,4
510 DATA 494,4,392,4,494,4,440,4
```

Note and Frequency Table

NOTE	FREQUENCY (octave ending before middle C)	FREQUENCY (octave beginning with middle C)
C	262	523
C#	277	554
D	294	587
D#	311	622
E	330	659
F	349	698
F#	370	740
G	392	784
G#	415	831
A	440	880
A#	466	924
B	494	988

6450-0

DR. ARON'S GUIDE TO THE CARE, FEEDING, AND TRAINING OF YOUR COMMODORE 64®

ARTHUR ARON AND ELAINE ARON

Tired of computer instruction manuals that do everything but speak clearly? Try this one.

Dr. Aron's Guide to the Care, Feeding, and Training of Your Commodore 64® reads like a teacher talking to you. It anticipates any problems you might have and makes everything clear right from the start. It repeats new ideas until they are familiar...and answers questions as they arise. It even asks *you* questions to make sure you're absorbing everything, then gives you practice exercises so you can immediately apply your knowledge.

Everything you need to know to use your Commodore 64 as quickly and thoroughly as possible is here, including:

* The fundamentals of setup, keyboarding, and basic programming
* More on programming
* Graphics and sound programming—including helpful hints for using POKES and sprites
* Writing and debugging your own programs

HAYDEN BOOK COMPANY
a division of Hayden Publishing Company, Inc.
Hasbrouck Heights, New Jersey

ISBN 0-8104-6450-0