COMPUSOFT
LEARNING SERIES

LEARNING BASIC FOR THE MACINTOSH™

MICROSOFT®

By David A. Lien

NEW MICROSOFT 2.0 VERSION

Learning MICROSOFT® BASIC for the MACINTOSH

by
David A. Lien

COMPUSOFT®
PUBLISHING
A DIVISION OF COMPUSOFT, INC., SAN DIEGO

International Standard Book Number: 0-932760-34-1

Library of Congress Catalog Card Number: 85-71339

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America.

A Personal Note From The Author

In my note for the first edition, I said the Mac could well be the beginning of a new generation of computers -- a machine that could truly be called "brand-new."

Apparently Microsoft agreed. They developed a **Version 2.0 BASIC** for the Mac and created a language with features that BASIC never had before -- features that go hand in hand with the Mac's innovative capabilities and user-friendly personality.

Of course, BASIC is still BASIC, an easy to learn and easy to use programming language with lots of powerful functions and statements. In this revised edition, I make sure that learning to do all the "new stuff" doesn't interfere with understanding the "basics of BASIC."

The Mac is easy and non-threatening to use and so is this book. Have fun with both of them as you learn. Let your imagination wander. I'll supply all the facts and techniques we'll need. Remember, the real enjoyment begins when *your* imagination takes over and the computer becomes a tool under *your* control. *You* become the master -- not the other way around!

Dr. David A. Lien
San Diego -- 1985

Acknowledgements

The following played key roles in the creation of this book:

Technical Director: Dave Waterman
Project Coordinator: Inez Goldberg
Technical Researchers:
Dan Gookin
Morgan Davis
Jody Bailey
Editorial Director: Gary Williams
Production Coordinator: Janice Scanlan
Cartoonist: Bob Stevens

Introduction

Learning Microsoft BASIC for the Macintosh is organized into four major sections:

A. Fifty-one chapters which teach how to use the many capabilities of your Macintosh ... in small enough bites so you won't choke. Many chapters include check points and examples.

   In most chapters there are Exercises. If you're studying alone, use them to test yourself and exercise your creativity. If you're studying with a class, your instructor may use them to supplement his own.

B. A section with the Answers to the Exercises.

C. A section with Appendices which provide useful reference tables and charts.

D. An Index, for easy reference after you've learned it all but forgotten where you learned it.

The Computer helps you to learn ... a sort of "Computer Aided Instruction."

Table Of Contents

A Personal Note From The Author iii
Acknowledgements iv
Introduction v

Section A: Microsoft BASIC Tutorial 1

Part 1. Getting Started 3

0 Setting It Up 5
1 The Desktop 7
2 Computer Etiquette 13
3 Using The Editor 21
4 Expanded Program 26

Part 2. BASIC Fundamentals 35

5 Math Operators 36
6 Scientific Notation 46
7 Using ( ) And The Order Of Operations 49
8 Relational Operators 54
9 It Also Talks And Listens 60
10 Calculator Or Immediate Mode 65
11 SAVEing And LOADing Using Disk 72
12 FOR-NEXT Looping 76
13 Son Of FOR-NEXT 86
14 Formatting With TAB 97
15 Grandson Of FOR-NEXT 103
16 The INTeger Function 109
17 More Branching Statements 120
18 Random Numbers 130
19 READing Data 141

Part 3. Strings **149**

20 Smorgasbord 151
21 The ASCII Set 161
22 Strings In General 168
23 Measuring Strings 173
24 VAL And STR$ 179
25 Having A Ball With String 183
26 TIME$ And DATE$ 195

Part 4. Variable Precision And Math **199**

27 What Price Precision? 200
28 Intrinsic Math Functions 209
29 The Trigonometric Functions 218
30 DEFined FuNctions 223

Part 5. Display Formatting **227**

31 Video Display Graphics 228
32 Intermediate Graphics 236
33 Formatting With LOCATE 246
34 Graphing Trig Functions 251
35 INKEY$ And INPUT$ 255
36 PRINT USING 263
37 PRINT USING -- Round 2 273
38 Using A Printer 281

Part 6. Arrays **287**

39 Arrays 288
40 Search And Sort 302
41 Multi-DIMension Arrays 310

Part 7. Miscellaneous **321**

42 PEEK And POKE 322
43 Logical Operators 329
44 A Study Of Obscurities 340
45 Advanced Graphics 350
46 Introduction To Data Processing 357
47 Advanced SAVEing, MERGEing, And CHAINing 367

Part 8. Program Control **377**

48 Flowcharting 378
49 Debugging Programs 383
50 Chasing Bugs 393
51 Chasing The Errors 397

Section B: Answers To Exercises **405**

Section C: Appendices **429**

Appendix A -- ASCII Chart 430
Appendix B -- Reserved Words 434
Appendix C -- Error Messages 436
Appendix D -- The Apple Menu 442

Section D: Index **451**

SECTION A

MICROSOFT BASIC TUTORIAL

PART 1

GETTING STARTED

Setting It Up

Before we begin learning to program in BASIC, it's necessary to learn about a number of the machine's special features. If you haven't done so already, find and install the small plastic programmer's switch, labeled "INTERRUPT RESET," included in the packaging. Install the switch according to the directions given in Apple's manual.

A Guided Tour Of Macintosh

For those who are new to Macintosh, a tour of the Computer is in order. Locate the audio cassette tape and diskette, each named "A Guided Tour of Macintosh" which came with the Mac. Together, the tape and disk will provide an excellent introduction to the Computer's unique features. The tour guides will explain how to insert the Guided Tour diskette, and how to turn the Computer ON. They will also talk us through a few examples while the Mac provides the demonstration. Start the audio tape, and begin the tour at this time.

The first example the tape refers to is "Mouse Exercises." When it does, stop the recorder. The Macintosh should be turned ON. Place the "mouse" on a clean level surface, and move it around until the little arrow on the screen is inside the box titled "MOUSING AROUND." Press the button on the mouse.

When done MOUSING AROUND, take the other trips on the tour.

** ** **

Well, that was pretty exciting, and challenging. Don't try to absorb all the details at once. Instead, after every few Chapters of this book go back and replay the "Guided Tour," and learn a little more. Only a few of the many mouse and window features are absolutely necessary for learning Microsoft BASIC on the Macintosh.

When through with the Guided Tour, select the box with "I'm Ready to Stop," and the diskette will be ejected. Remove the Guided Tour disk.

Let's move on now to Chapter 1 to learn more about our electronic desktop.

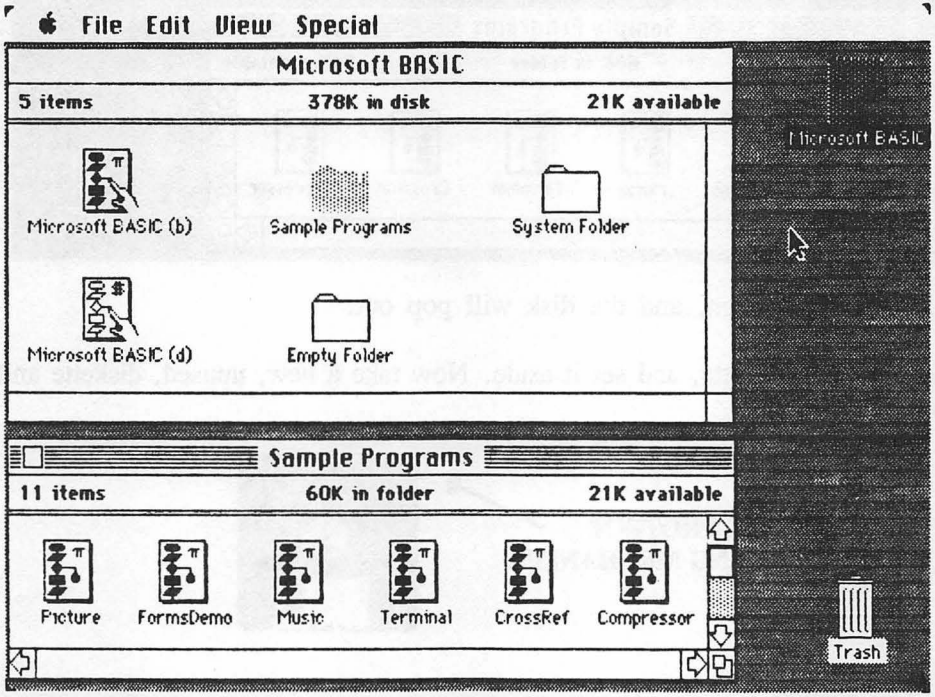
Chapter 1

The Desktop

Insert the diskette named MICROSOFT BASIC Interpreter into the slot. (We refer to this disk as the Master diskette.) In a few seconds, the screen will display the Microsoft BASIC window, the Sample Programs window, the Trash Can icon, the Microsoft BASIC icon in the upper right corner, and of course, the mouse pointer. This is our "desktop" which displays the contents of the disk currently in the drive.

If you've been using your Mac and Microsoft BASIC Master disk already, what appears on your desktop may be different. You may, for example, have other windows opened, or everything, including the Microsoft BASIC and Sample Programs windows, may be closed. No problem...

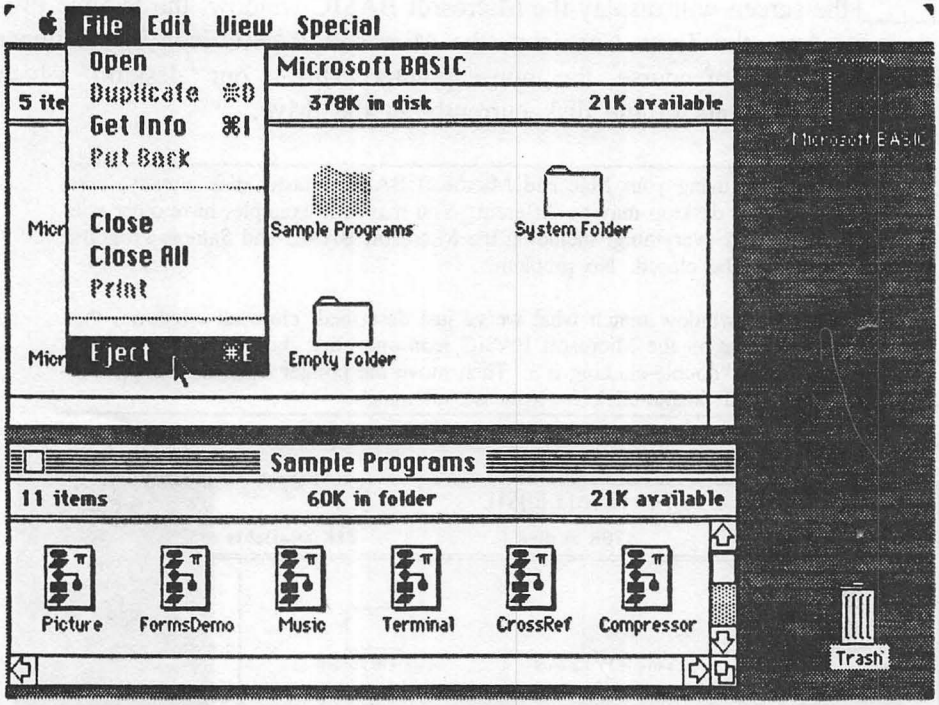
To make your window match what we've just described, close all windows, then place the pointer on the Microsoft BASIC icon and click the mouse button twice (also known as "double-clicking it"). Then move the pointer onto the Sample Programs icon, and double-click it. Now we all match.



Initializing A Diskette

We need to make a copy of this disk, one that we can store our programs and assignments on, but first, we need to prepare a completely blank diskette for our copy of the Master diskette. This process is called *Initialization,* and it involves putting special magnetic "race tracks" on the disk.

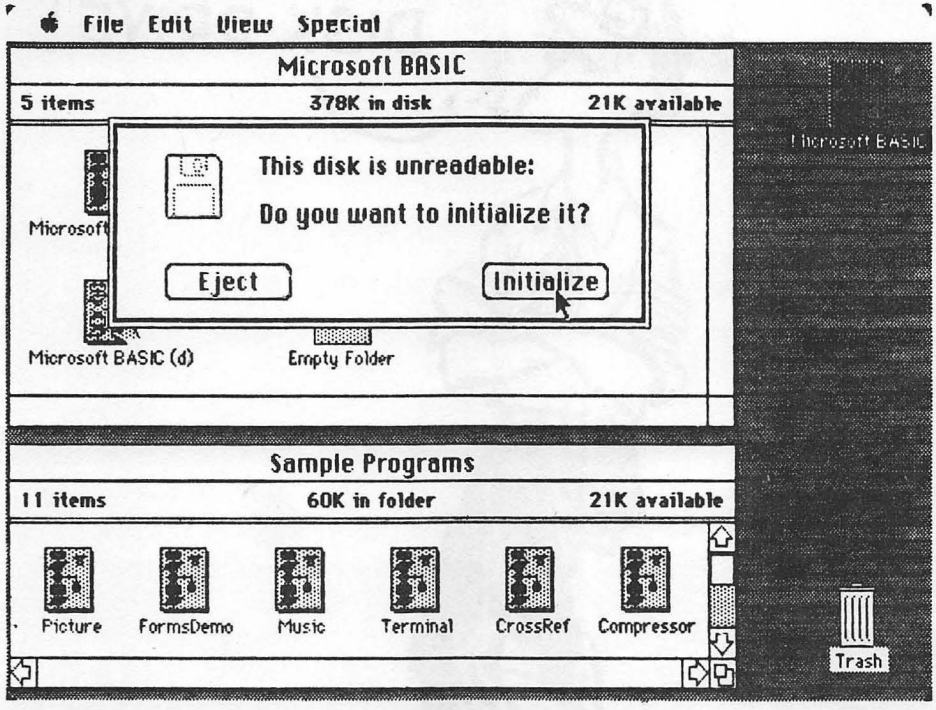
To Initialize a new disk, we'll need to eject this one. (This one has done its job for now -- it prepared the Mac for the initialization process we're about to carry out.) Move the mouse to the File menu at the top of the screen, hold down the button, and drag the pointer down to Eject. When Eject is shaded in black like this:



...release the button, and the disk will pop out.

Remove this diskette, and set it aside. Now take a new, unused, diskette and look it over.

The "write protect" locking mechanism is in the upper left-hand corner. When the disk is protected (locked), the tab is up and a hole is visible. To allow the Computer to write to the diskette, the write protect tab should be covering the hole. Slide the write protect tab downward, then insert the new disk into the Computer. This dialog box will appear on the screen:



Move the mouse pointer to the **Initialize** box, press the button once, and listen to the Computer hum.

When Macintosh is done, we are asked to name the disk. Any name of up to 27 characters (except the colon) can be used. With that kind of range, there should be no trouble choosing names that distinguish one disk from another. Let's use the name LEARNING. No need to use the Shift key -- letters are always *capital* once we press down the Caps Lock key. Type:

```
LEARNING
```

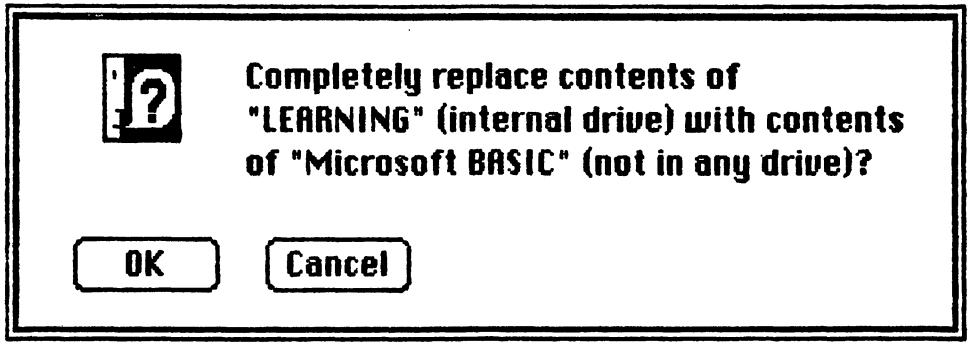
...and click the button inside the **OK** box (or press Return on the keyboard). The dialog box disappears, and the new disk's icon is placed on the screen below the Microsoft BASIC icon. Such magic!


DISK DRIVE!

Backing Up The Microsoft BASIC Disk

Our next task is to make a backup (safety copy) of the original Microsoft BASIC diskette. We'll use the copy as our "working master," and hide the original for safe keeping. This precaution may save a long drive down to the Apple computer store. You are allowed to make as many backup copies of the Master diskette as needed for your own personal use, subject to any provisions stated in the factory notice.

Move the pointer over the Microsoft BASIC icon, hold down the mouse button, and drag the icon's shadow down to the newly created LEARNING disk. Once the lower icon is shaded in black, release the button. A dialog box will ask:



which freely translated means: "Do you want to make a backup of the Microsoft BASIC diskette (which is not in the Computer right now) and put it on LEARNING (which *is* inside the Computer)?"

Respond by clicking the **OK** box.

Two dialog boxes appear. The top one shows how many files remain to be copied. The bottom one tells when to switch disks. This swapping process is slow, but with only one drive, it is necessary. The Macintosh will load some files from the original disk into temporary memory then eject the disk. When we insert the new disk, it will "dump" what it has stored onto the new disk, then ask for more. When the backup is complete, the boxes will disappear, the drive will stop and we'll have an exact duplicate of the Microsoft BASIC disk with enough empty space remaining to hold all our programs and exercises.

Eject the Master disk. Then press the RESET button on the programmer's switch. Macintosh beeps and clears the screen. Insert the new LEARNING disk. Observe that an icon of our new disk is displayed. Double-click the LEARNING disk icon to confirm that all the programs and files from the original are present on this backup.

Turning The System Off

It is best to have all files and windows closed and the disk ejected before RESETing or shutting the power off. Use either the File menu, or press the ⌘ (Command) and E keys at the same time to eject the disk. Then reach around to the left rear of the computer and turn it off.

It is not always necessary to turn Macintosh off. (It only uses 60 watts of electricity.) However, if you decide to keep it on for a long time, remember to turn down the screen brightness. If you don't, and the machine stays on for a long time, it could damage the screen by burning an image onto it. The brightness adjustment is located below the Apple logo under the lower left front of the computer where you see the ☼ symbol.

Learned In Chapter 1

Miscellaneous	Menu
Turning the Computer ON and OFF	File
Dialog box	Eject (⌘ E)
Command key - E (⌘ E)	
Initializing a disk	
Backing up a disk	
Programmer's switch (RESET)	

There is a review summary like this at the end of each chapter to be sure you didn't miss anything.

Chapter 2

Computer Etiquette

From the moment we turn it on, our Macintosh follows a well-defined set of rules for coping with us, the "master." This makes it an exceptionally easy computer to use. To a large extent, all we have to do is say the right thing (via the keyboard or the mouse) at the right time. Of course, there are lots of "right things" to say; putting them together for a purpose is called *programming*.

In this Chapter we'll start a conversation with our Macintosh and teach it some simple social graces. At the same time, you'll learn the fundamentals of computer etiquette. You'll even write your first computer program!

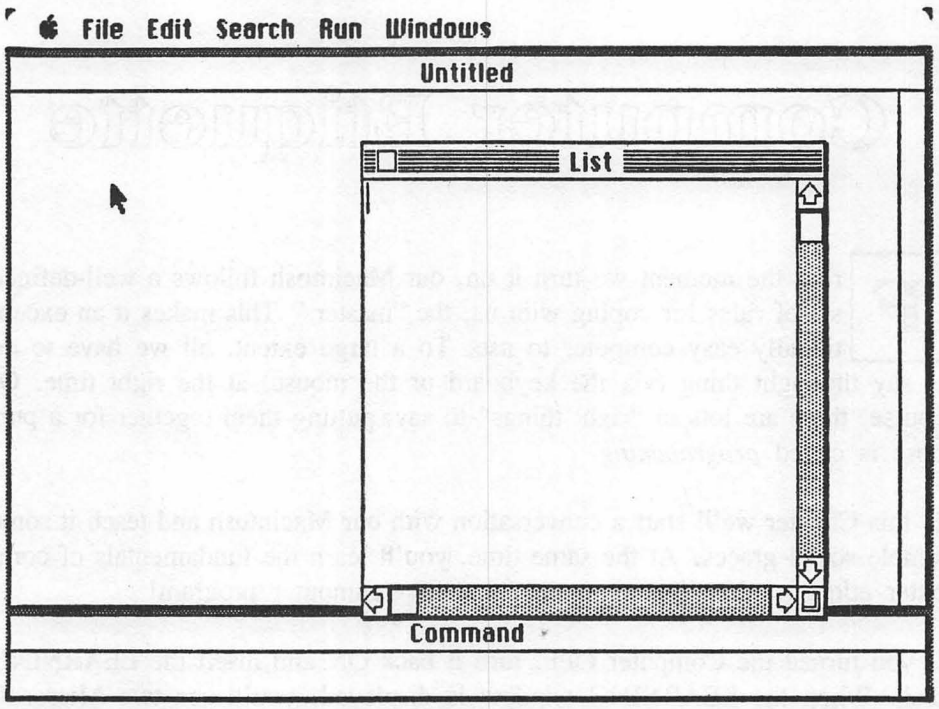
If you turned the Computer OFF, turn it back ON and insert the LEARNING disk. When the LEARNING window is displayed, you'll see two Microsoft BASIC icons in the upper window, Microsoft BASIC (b) and Microsoft BASIC (d), either of which will teach the Computer to speak BASIC.

For our purpose of learning BASIC, either version could be used. BASIC (b) is a Binary version while BASIC (d) is a Decimal version. All this means is that version (b) manipulates numbers in single precision while version (d) uses double precision. We'll see what this is all about in Chapter 27.

Move the mouse over the Microsoft BASIC (b) icon, and double-click the mouse to "load in" BASIC.

We chose Binary BASIC because it occupies a little less memory space and manipulates arithmetic operations a little faster.

When this screen:



appears, we're set to "go." The blinking vertical bar in the List window is called the *cursor* (also referred to as the *insertion point*). The Computer is saying:

"I'm ready -- it's your turn!"

To make sure we start off with a clean slate -- erasing all traces of prior programs -- drag the pointer down the File menu, and release the mouse button when New is outlined in black. The NEW command can also be entered from the keyboard. Place the pointer anywhere in the Command window at the bottom of the screen, and click the mouse button. Type NEW, and press Return.

Reactivate the Command window. Then type PRINT FRE(0) and Return. Note that while this Line appears in the Command window, the result appears in the Output window once Return is pressed. This is a very simple test to see that the Computer "powered up" properly. The display should read:

```
21000
```

If the number is not 21000, select Quit from the File menu and when you are back in the Finder, eject the disk. Turn the Computer OFF, and wait about 10 seconds before turning it ON again. Repeat the test, and verify that the number is in the ball park.

What Is A Computer Program?

A program is a sequence of instructions the Computer stores until we command it to follow (or "execute") them. Some programs for the Macintosh are written in a language called Microsoft BASIC, and its very name tells how easy it is to learn!

Let's write a simple one-Line program to let the Macintosh introduce us.

We must first place the cursor back in the List window since program Lines can only be written in this window. Position the pointer inside the List window, and click the button. Note that the title bar on the List window is highlighted and that the flashing cursor has moved to the upper-left corner of this window.

Type the following Line, *exactly* as shown:

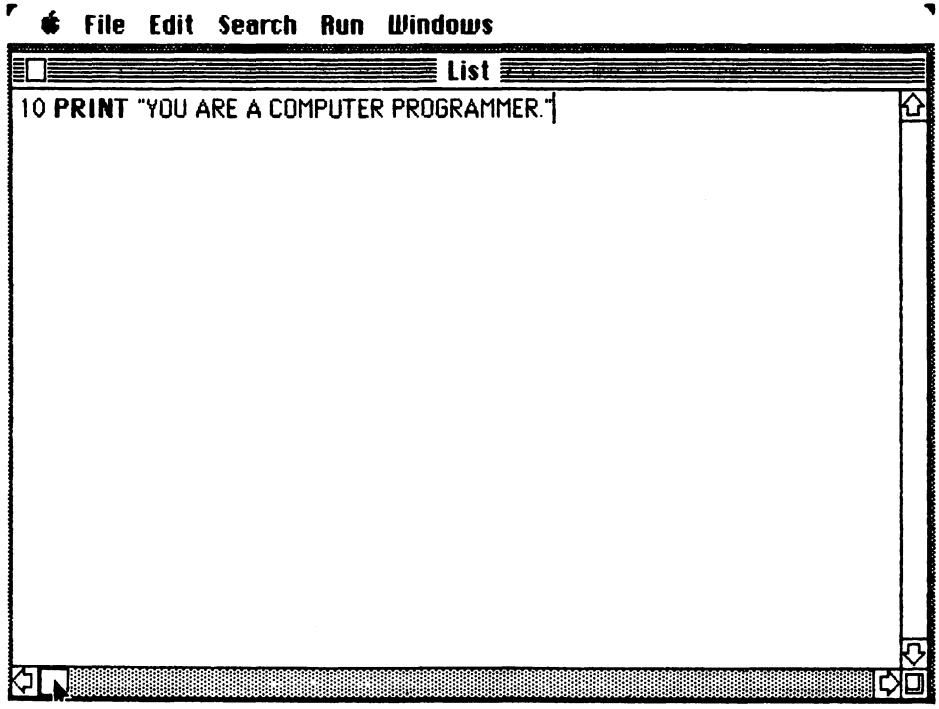
```
10 PRINT "YOU ARE A COMPUTER PROGRAMMER."
```

Do *not* hit the Return key yet!

Notice that the program Line is too long to fit inside the List window. To see the entire Line, place the pointer in the List title bar and double-click the button. The List window now fills the entire screen. Place the arrow back on the title bar and double-click again to return the List window to its original size.

We could have enlarged the window by moving it and using the size box, but then we would have to resize it each time we type in a longer program Line. Best to see the entire program at once.

Enlarge the List window again to check your program. Slide the horizontal scroll bar to the left to see the entire Line.



If you made a typing error, don't worry. Just use the Backspace key. Each time you press this key, the rightmost character will be erased. If the error was at the beginning of the Line, erase way back to that point, and then retype the rest of the Line. (If you hold the Backspace key down longer than a second, it will erase many letters very quickly.)

Study *very carefully* what you typed:

1. Is everything after the word PRINT enclosed in quotation marks?

2. Are there any extra quotation marks?

If everything's okay, press Return. The flashing | cursor will move to the left edge of the List window, telling us, "I got the message." The Line you typed in will be displayed with PRINT in bold type.

If It's Too Late

If you found an error after pressing Return, the Backspace key cannot correct it. The best way to fix it, for now, is by "pulling down" the File menu and selecting New. The Computer will ask if you want to save "the current


GO AHEAD, POKE AWAY - I WON'T BYTE! "YOU ARE A COMPUTER PROGRAMMER" SOUNDS GRATE!

program" before proceeding. Answer by clicking the No box. When the List window reappears, type in the one-Line program. In the next Chapter we'll learn how to "Edit" out errors instead of retyping entire Lines.

"Allow Me To Introduce You"

Let's tell the Computer to execute, or RUN, our program. The BASIC command for this is simple. Move the pointer to the Run menu, hold down the button, drag down and release the button when Start is outlined in black.

Wow, that was fast! We didn't have much of a chance to look at the Output window before it was covered over by the List window. Reduce the size of the List window by double-clicking the title bar. This time Run the program by moving the pointer to the Command window, clicking the mouse and typing:

```
RUN          Return
```

You can also RUN the program by pressing the ⌘ and R keys while in either the List or Command windows.

Return to the List window by typing:

```
LIST         Return
```

in the Command window, selecting Show List from the Windows menu or by simply pressing ⌘ L.

> In the future when you are asked to Run the program, you can either type RUN in the Command window, press ⌘ R or select Start from the Run menu.

If we made no mistakes, the line in the Output window will read:

```
YOU ARE A COMPUTER PROGRAMMER.
```

If it doesn't work, try to Run it again. If Run still doesn't produce the statement, there's something wrong in your program. Choose New from the File menu or type NEW Return in the Command window to clear it out, then type it in and Run again.

When selecting New from the File menu (or typing it in the Command window), the Computer may ask if you want to Save the current program before proceeding. Select the **No** box. Later we will start Saving our more important programs.

If it did work -- let out a yell!

"I are now a REAL computer programmer!"

This is very important because you have tasted success with computer programming, and it may be the last you are heard from in some time.

In Summary

Note that the word PRINT is not displayed, nor is the Line number nor the quotation marks. They are part of the BASIC Language program's *instructions*, and we didn't intend for them to be printed. Everything inside the quote marks is printed, including blank spaces and the period.

From the Run menu choose Start again.

Run to your heart's content, watching the magic machine do as it's told. When you feel you've got the hang of all this, get up and stretch, walk around the room, look out the window -- the whole act. You'll soon be absorbed in programming and won't have time for such things.

When typing in a program, we can choose direct commands like Start from the Run menu or we can type them in at the Command window, but remember to hit Return to tell the Computer to look at what we *typed*, then act accordingly.

Quitting BASIC Before Turning Off Mac

If you want to stop here, remember to turn down the brightness control or if you're planning a really long break, turn off the Computer completely. First get out of BASIC by selecting Quit from the File menu. Don't bother to save this program. It's so short, it can easily be retyped when it's needed. Eject the disk, and flick the power switch to off.

Learned In Chapter 2

Commands	Statements	Miscellaneous	Menus
NEW	PRINT	Entering BASIC	File
RUN	Return	| cursor (insertion point)	New
LIST		Backspace key	Quit
		" " quotation marks	Run
		List window	Start (⌘ R)
		Command window	Windows
		Output window	Show List (⌘ L)
		Enlarging the List window	

Commands (like RUN) are executed as soon as we type them in the Command window and press Return.

Statements (like PRINT) that are typed in the List window are executed only after we press ⌘ R, select Start from the Run menu or type RUN Return in the Command window.

> **Special message for people who can't resist the urge to play around with the Computer and skip around in this book.** (There always are a few!)
>
> It is possible to "lose control" of the Computer so it won't react to the mouse or keyboard. To regain control, just press ⌘ . (Command-period). If that doesn't work, push the INTERRUPT button. If that doesn't work, turn the Computer OFF for 10 seconds, then turn it back ON again.

Chapter 3

Using The Editor

An extraordinarily valuable capability of our BASIC is a feature called the Editor. Its purpose is as simple as its name. It lets us "EDIT," or make simple changes, in a program.

The Microsoft BASIC Editor gives us the ease and power of using a "word processor." It is so easy to use but so powerful you'll never again want to use a computer without one.

We now have a program in the Computer. (If you turned OFF your Computer at the end of the last Chapter, just retype the one-Line program in Chapter 2.) Enlarge the List window by double-clicking its title bar, and then we'll expand the program to read:

```
10 PRINT "YOU ARE A COMPUTER PROGRAMMER.  ARE
   YOU IN COMMAND?"
```

To do this, first place the pointer at the end of the program Line (to the right of the last quotation mark), and click the button. Notice that the *insert bar* is flashing at the end of the Line.

To get rid of the closing quotation mark, simply press Backspace like we did in the last Chapter. Now press the space bar twice, then type in the words:

```
ARE YOU IN COMMAND?"          Return
```

This program will run just fine. If, on the other hand, we wish to change the Line to something like:

```
10 PRINT "YOU ARE IN COMMAND.  YOU ARE THE
   COMPUTER PROGRAMMER."
```

then we need to do some Editing to Line 10.

Earlier we would have solved the problem by backspacing through the entire Line and retyping it, hoping we didn't make more mistakes than we eliminated. This particular example has so much to change, it might be just as easy to retype, but our purpose is to "exercise" the Editor.

Since we want the word A to be changed to THE in the sentence YOU ARE A COMPUTER PROGRAMMER., place the cursor on the right side of the letter A, and click the mouse. The listing shows:

```
10 PRINT "YOU ARE A| COMPUTER PROGRAMMER.  ARE
   YOU IN COMMAND?"
```

(If your screen doesn't look like this, move the cursor again and click until it does. It takes a steady hand.)

Press Backspace once to remove A. We now have to insert the word THE between ARE and COMPUTER. The insert bar (or insertion point) is already in position, so all we do is type the letters:

```
THE
```

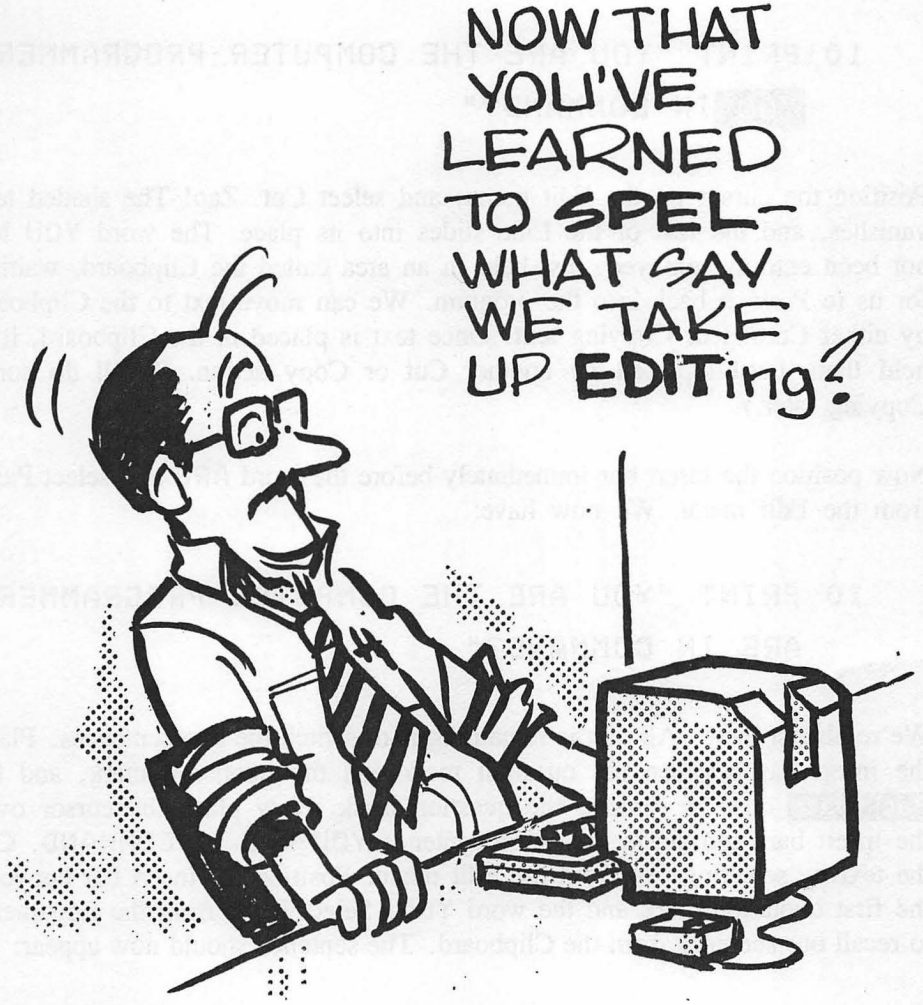
The screen now reads:

```
10 PRINT "YOU ARE THE| COMPUTER PROGRAMMER.  ARE
   YOU IN COMMAND?"
```

Notice that, as you typed in the letters, the remaining Line moved to the right to make room for the inserted letters.

If it seems we're going slowly, you're right! The Editor is simple but so important, we may as well learn to use it right the first time. You know the old story, "There's never time to do it right the first time, but always time to do it over."

We now have to reverse the words ARE and YOU in the sentence ARE YOU IN COMMAND?. To do this, we will use the Editor's *Cut* and *Paste* feature.


NOW THAT YOU'VE LEARNED TO SPEL- WHAT SAY WE TAKE UP EDITing?

Position the cursor to the left of the letter I in IN , and click the mouse. The bar cursor should be flashing on the left side of the word IN. With the cursor positioned over the bar cursor, press the button and drag left to shade the word YOU. The Line should show:

```
10 PRINT "YOU ARE THE COMPUTER PROGRAMMER.  ARE
   YOU IN COMMAND?"
```

Position the cursor on the Edit menu, and select Cut. Zap! The shaded text vanishes, and the rest of the Line slides into its place. The word YOU has not been entirely removed, just held in an area called the Clipboard, waiting for us to Paste it back into the program. We can move text to the Clipboard by either Cutting or Copying text. Once text is placed in the Clipboard, it is held there until replaced by another Cut or Copy action. (We'll do some Copying later.)

Now position the insert bar immediately before the word ARE and select Paste from the Edit menu. We now have:

```
10 PRINT "YOU ARE THE COMPUTER PROGRAMMER.  YOU |
   ARE IN COMMAND?"
```

We're almost there. All that is remaining is to switch the two sentences. Place the insert bar between the question mark and the quotation mark, and hit Backspace once to remove the question mark. Now place the cursor over the insert bar, and shade the entire sentence YOU ARE IN COMMAND. Cut the text by selecting Cut from the Edit menu. Position the insert bar between the first quotation mark and the word YOU. Select Paste from the Edit menu to recall our sentence from the Clipboard. The sentence should now appear:

```
10 PRINT "YOU ARE IN COMMAND|YOU ARE THE COMPUTER
   PROGRAMMER. "
```

Add the finishing touches to the Line by typing a period and two spaces. Then move the cursor in front of the last quotation mark, and press Backspace to remove the extra space at the end of the sentence.

Whew, finally done. But wait -- the insert bar is still sitting inside the program Line. Move the cursor directly below the program Line, and click the mouse. Line 10 is displayed with PRINT in bold type, and the cursor is in position waiting for us to type in the next Line.

From here on, we should always use the Editor for making changes, especially in long Lines. Compare the time it would take to change only one character in a very long and complex Line by retyping it, with the speed of doing it with the Editor.

EXERCISE 3-1: Choose New from the File menu, then use the Editor to change:

```
10 PAINT "WE CAN TAKE CREDIT FOR CONSUMER
   PROGRESS."
```

to:

```
10 PRINT "WE CAN EDIT COMPUTER PROGRAMS."
```

Try working this one out *on your own*. The answers to later Exercises will be provided in Section B, along with further comments.

Learned In Chapter 3

Menu	Miscellaneous
Edit	Editing features
Cut (⌘X)	
Paste (⌘V)	

Chapter 4

Expanded Program

After doing Exercise 3-1, we still have a program in the Computer. It's only a one-Liner, so let's expand it by adding a second Line.

One of the features available in this version of Microsoft BASIC which is not found in most other BASICs is its ability to write program Lines without assigning a number to every Line. We have the option of identifying program Lines with numbers or titles, or we can write program Lines leaving off both numbers and titles. The program's instructions are executed in order from the top Line to the bottom without regard to the order in which Lines are numbered. Later we will learn how these Line numbers or labels are necessary when branching from Line to Line within the program.

To help explain what is happening with each program Line used in this book, we will be assigning Line numbers in most of the examples.

Let's add the next Line, and leave out the Line number. Type:

```
PRINT "LINE NUMBERS ARE NOT REQUIRED."
```
Return

You did enlarge the List window, didn't you?

Check it carefully -- especially the quote marks. The program listing should show:

```
10 PRINT "WE CAN EDIT COMPUTER PROGRAMS."
PRINT "LINE NUMBERS ARE NOT REQUIRED."
```

Notice how PRINT is again displayed in boldface type. The Computer does this to help us pick out the BASIC statements, commands and functions from the other words.

Return the List window to its original size by double-clicking its title bar and Run the program.

If all was correct, the screen will read:

```
WE CAN EDIT COMPUTER PROGRAMS.
LINE NUMBERS ARE NOT REQUIRED.
```

Who Goofed?

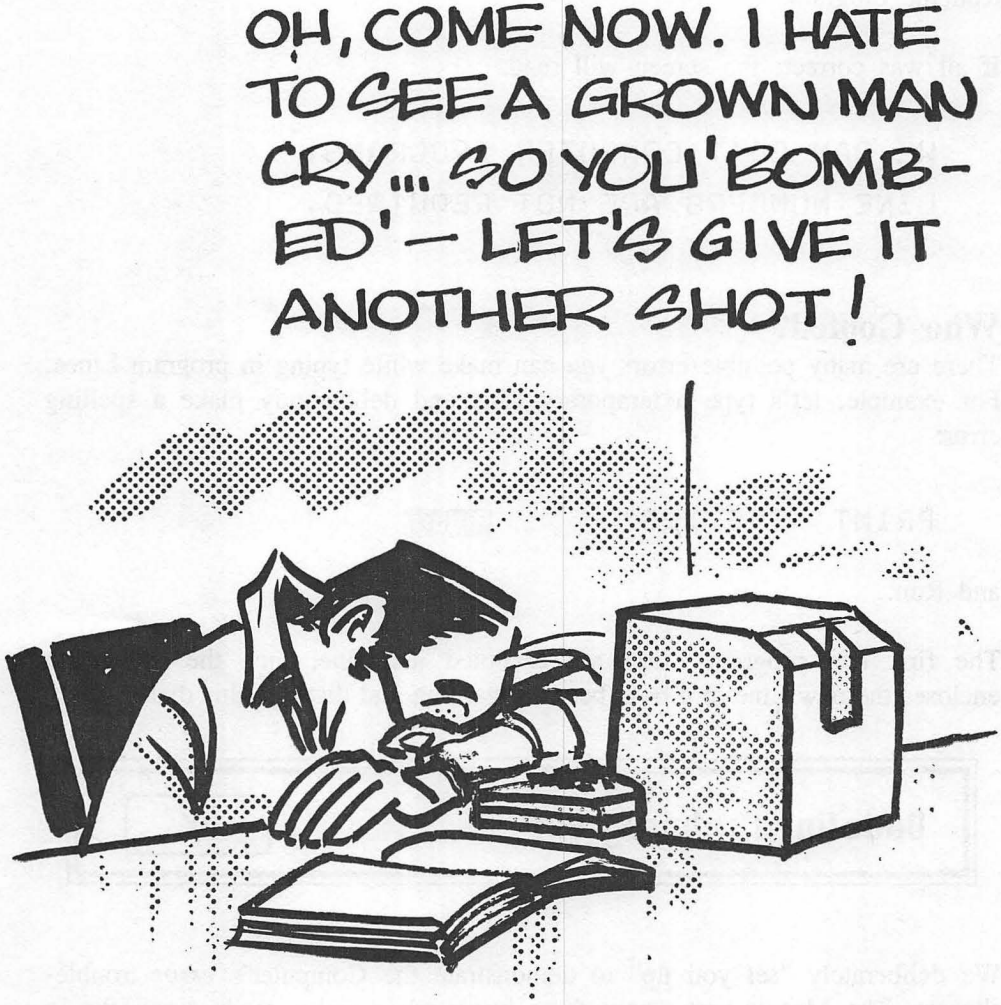
There are many possible errors *you* can make while typing in program Lines. For example, let's type a temporary Line and deliberately make a spelling error:

```
PRIMT "TESTING"          Return
```

and Run.

The first two program Lines are executed just fine, then the Macintosh encloses the new Line in a box, beeps a warning and displays this dialog box:

We deliberately "set you up" to demonstrate the Computer's *error* trouble-shooter. The Mac is smart enough to know when *we've* made a mistake in telling it what to do, and it PRINTs a clue as to the nature of the error. The Computer looked for a "subprogram" within this program called **PRIMT "TESTING"** and couldn't find it. Later we will see how smaller subprograms can be placed within our main program and how these subprograms can be labeled or assigned Line numbers.


OH, COME NOW. I HATE TO SEE A GROWN MAN CRY... SO YOU 'BOMB-ED' — LET'S GIVE IT ANOTHER SHOT!

To acknowledge the error, move the mouse and click it inside the **OK** box, or just hit the Return key.

The dialog box disappeared, but the temporary Line is still in the program, inside its error box. Click the mouse with the cursor positioned at the left side of the error box. Use the Editing skills learned in the last Chapter to Cut away the temporary Line.

And The Program Grows

In most BASICs where Line numbers are required, it is customary, traditional (and all that) to space the Lines ten numbers apart to leave room to insert new Lines between the old ones. Since Microsoft BASIC is not dependent on Line numbers, we can insert new Lines wherever needed with the Editor. However, when numbers are assigned to program Lines, we try to follow this rule of thumb.

Run again, and look at the Video Display. What if we'd rather not have the two Lines PRINTed so close together, but would like to have a space between them? Type in the new Line:

```
20 PRINT          Return
```

Our program Listing now looks like:

```
10 PRINT "WE CAN EDIT COMPUTER PROGRAMS."
PRINT "LINE NUMBERS ARE NOT REQUIRED."
20 PRINT
```

Now Run.

There doesn't seem to be any additional space PRINTed between the two lines. What happened? The Computer encountered Line 20 after it had already PRINTed the first two lines. To further illustrate this point, insert the number 30 at the beginning of the second Line:

```
10 PRINT "WE CAN EDIT COMPUTER PROGRAMS."
```

```
30 PRINT "LINE NUMBERS ARE NOT REQUIRED."
20 PRINT
```

and Run.

Even though the third PRINT statement has a Line number lower than that of Line 30, it is executed in order of appearance (from top to bottom).

Now, insert Line 20 between Lines 10 and 30 by Cutting Line 20 and Pasting it at the beginning of Line 30. Press Return to move Line 30 to the next Line so the Listing shows:

```
10 PRINT "WE CAN EDIT COMPUTER PROGRAMS."
20 PRINT
30 PRINT "LINE NUMBERS ARE NOT REQUIRED."
```

Then Run.

It now displays:

```
WE CAN EDIT COMPUTER PROGRAMS.

LINE NUMBERS ARE NOT REQUIRED.
```

Note: To make this book easier to read, we are using more space between all our program Lines than you actually see on the screen.

Looks neater, doesn't it? But what about Line 20? It says PRINT. PRINT what? PRINT nothing. That's what followed PRINT, and that's just what it PRINTed. Remember, we added Line 20 to keep Lines 10 and 30 from PRINTing so close together. Well -- in the process of PRINTing nothing, a space was automatically inserted between the PRINTing ordered in Lines 10 and 30. (Hmmm...so *that's* how we space between lines.)

Another important program statement is REM, which stands for REMark. It is often convenient to insert REMarks into a program.

Why? So you or someone else can refer to them later, to help remember complicated programming details, or even what the program's for and how to use it. It's like having a scratch-pad or notebook built into the program. When we tell the Computer to execute the program, it skips right over any Line which begins with a REM. *A REM statement has no effect whatsoever on the program.* Insert the following at the beginning of the program:

```
5 REM *THIS IS MY FIRST COMPUTER PROGRAM*  Return
```

You might be wondering why the asterisks(*) in Line number 5? The answer is ... they're just for decoration. Let's give this operation some class! Remember, *anything* on a Line that follows REM is ignored by the Computer.

Then Run.

The "video printout" reads just like the last one, totally unaffected by the presence of Line 5. Did it work that way for you?

Microsoft BASIC allows us to view two List windows at the same time. We saw earlier how we can expand the List window to full size by double-clicking inside the title bar and return it to the original size by double-clicking it again. Now let's place a second List window over the first one.

If your List window is full size, return it to its reduced size. Pull down the Windows menu, and select Show Second List. Notice how nicely the windows stack on top of each other. This will come in handy when we have a large program in memory and want to look at different parts of it at the same time.

Each window acts independently of the other. Each can be enlarged to full size, moved around on the desktop and reshaped to suit our needs. If we completely remove both List windows from the desktop by clicking the mouse inside each of their close boxes, we can bring back either the first or second List window. The first List can be brought back by any one of 3 methods:

1. Select Show List from the Windows menu.

2. Type ⌘ L from the keyboard.

3. Type LIST in the Command window (select Show Command from the Windows menu, and type LIST Return).

The second List window can only be brought back by selecting Show Second List from the Windows menu.

Now, with all that information on List windows, experiment on your own. Learn how they work before continuing.

Where Is The END Of The Program?

The end of a program is, quite naturally, the last statement we want the Computer to execute. Many computers require placing an END statement at this point so the computer will know when to stop. But with Microsoft BASIC, an END statement is optional. Remember though, if you want to Run BASIC programs on fussier computers, they may need END statements.

When we get into more complex programs, we'll use END statements to *force* execution to END at specified points.

Let's take a closer look at END. By the rules governing its use, most dialects of BASIC which require END insist that it be the last statement in a program, telling the computer "That's all, folks." By tradition, it is given the number 99, or 999, or 9999 (or larger), depending on the largest number the specific computer will accept. Macintosh accepts Line numbers up to 65529.

With one List window (default size) displayed, let's add an END statement to our program:

Type:

```
99 END          Return
```

Then Run.

The sample Run should read:

```
WE CAN EDIT COMPUTER PROGRAMS.

LINE NUMBERS ARE NOT REQUIRED.
```

Question: "Why didn't the word END PRINT?" **Answer:** Because nothing is PRINTed unless it is the "object" of a PRINT statement. So, how could we

make the Computer PRINT THE END at the end of the program execution? Think for a minute before reading on, then insert the next Line between Lines 30 and 99.

```
98 PRINT "THE END"          Return
```

...and Run.

This assumes that Line 98 is the last PRINT statement in the program. We now have an END statement (Line 99) and a PRINT "THE END" statement (Line 98). 98 says it; 99 does it.

Erasing Without Replacing

Just for fun, let's move the END statement from Line 99 to the largest usable Line number our Microsoft BASIC will accept, 65529.

Using the Editor, shade the number 99, and Cut it out. Now type in the number 65529, move the pointer directly below that Line and click the mouse button.

The List window should show the program with Lines 5, 10, 20, 30, 98 and 65529. Now Run the program to see if moving the END statement changed anything. Did it? It shouldn't have.

Other Uses For END

Using the Editor, move END from number 65529 to Line number 15, then Run.

What happened? It ENDed the Run after PRINTing Line 10. Run it several times.

Now move END to Line 8, and Run.

Do you see the effect END has, depending where it is placed (even temporarily) in a program? Feel like you are really gaining control over the machine? You ain't seen nothin' yet!

Learned In Chapter 4

Commands	Statements	Miscellaneous	Menu
LIST	PRINT (Space)	Error Messages	Windows
	REM	Line Numbering	Show Command
	END	List windows	Show List (⌘ L)
		Title bar	Show Second List

PART 2

BASIC FUNDAMENTALS

Chapter 5

Math Operators

But Can It Do Math?

Yes, it can. Basic arithmetic is a snap for Microsoft BASIC. So are highly complex math calculations -- when we write special programs to perform them -- and we will.

The BASIC Computer language uses the 4 fundamental arithmetic operations, plus 4 more complex ones which are just modifications of the others:

1. ADDITION, using the symbol +

2. SUBTRACTION, using the symbol − *(See -- nothing to this -- just like grade school. I wonder whatever happened to old Miss... Well, ahem -- anyway...)*

3. MULTIPLICATION, using the special symbol * *(Oh drat, I knew this was too easy to be true!)*

4. DIVISION, using the symbol / *(Well, at least it's simpler than the ÷ symbol.)*

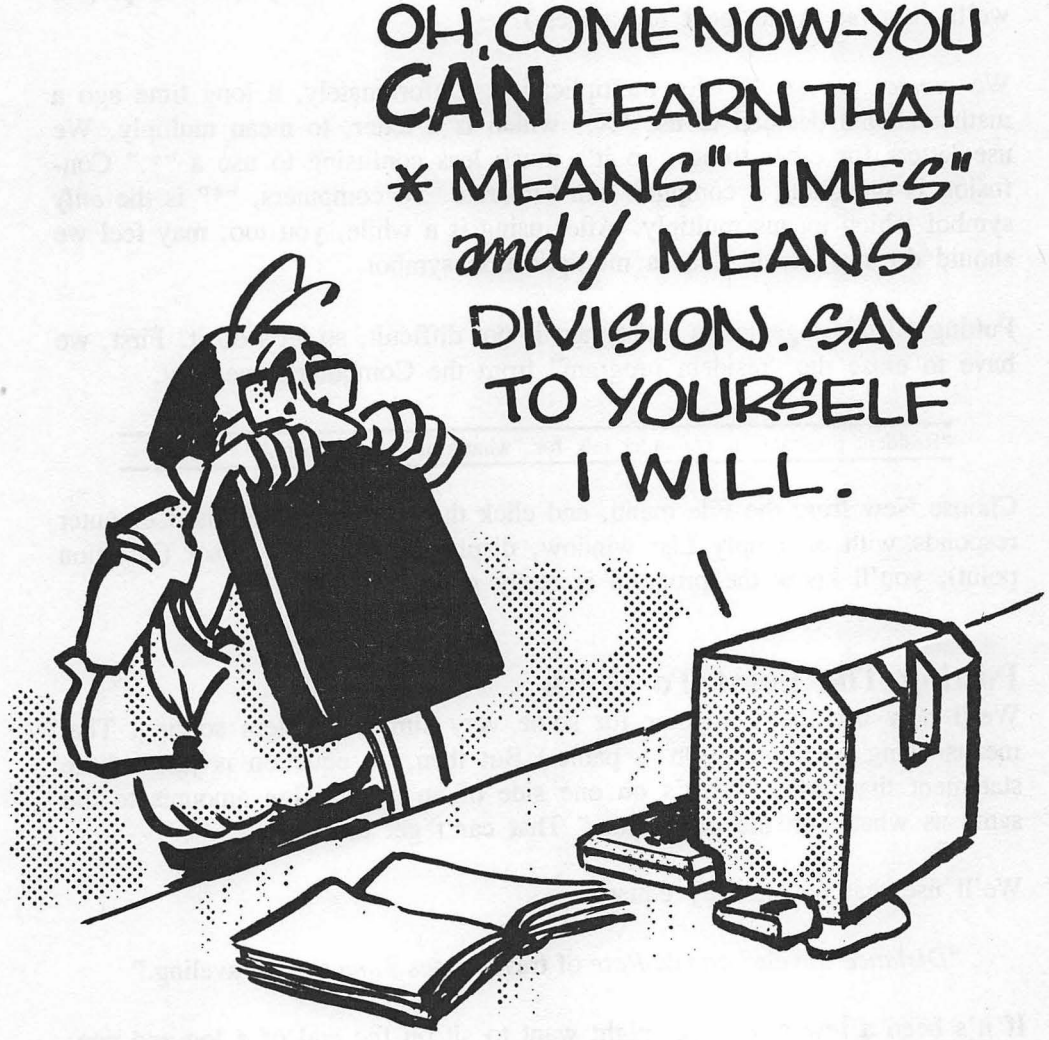
and

5. EXPONENTIATION, using ^ (unveiled in Chapter 28)

6. NEGATION (meaning "multiply times minus one"), using the − symbol

7. MODulo, of interest primarily to pure math-computer types (We'll discuss it in Chapter 28.)

8. INTEGER DIVISION, using the backslash \ (Taught in Chapter 16)


OH, COME NOW-YOU CAN LEARN THAT * MEANS "TIMES" and / MEANS DIVISION. SAY TO YOURSELF I WILL.

Of course, we also need that old favorite, the equals sign (=). But wait! The BASIC language is very particular about how we use this sign! Math expressions (like 1 + 2 * 5) can only go on the *right-hand* side of the equals sign; the left-hand side is reserved for the *result* of the math equation. We say 4 = 2 + 2. (This may seem a little strange, but it's really quite simple, as we'll discover in the next few pages.)

We *cannot* use an "X" for multiplication. Unfortunately, a long time ago a mathematician decided to use "X," which is a letter, to mean multiply. We use letters for other things, so it's much less confusing to use a "*." Confusion is one thing a computer can't tolerate. To computers, "*" is the *only* symbol which means multiply. After using it a while, you too, may feel we should do away with X as a multiplication symbol.

Putting all this together in a program is not difficult, so let's do it. First, we have to erase the "resident program" from the Computer's memory.

"Resident program" is computer talk for "what's already in there."

Choose New from the File menu, and click the **No** box. When the Computer responds with an empty List window, displaying only the cursor (insertion point), you'll know the program is really gone.

Putting The Beast To Work

We'll now use the Computer for some very simple problem solving. That means using equations. (Oh -- panic.) But then, an equation is just a little statement that says, "What's on one side of an equals sign amounts to the same as what's on the other side." That can't get too bad.

We'll use that old standby equation,

"*Distance* traveled equals *Rate* of travel times *Time* spent traveling."

If it's been a few years, we might want to sit on the end of a log and contemplate that for awhile.

To shorten the equation, let's choose letters (called variables) to stand for the 3 quantities. Then we can rewrite the equation as a BASIC statement accept-

able to the Computer. Type in:

```
40  D = R * T          Return
```

Remember, we have to use a * to specify multiplication.

What's that 40 doing in our equation? That's the program Line Number. Remember, this version of Microsoft BASIC does not require the use of Line numbers, but it's easier to make reference to specific Lines by numbering them. We chose 40, but any other number would have done just as well.

Here's what Line 40 means to the Computer: "Take the values of R and T, multiply them together, and assign the resulting value to the variable D." So until further notice, D is equal to the result of R times T.

We *could not* reverse the equation and write: R*T=D. It has no meaning to the Computer. Remember, the left-hand side of the equation is reserved for the Line number and the value we are *looking for*. The right-hand side is the place to put the values we *know*.

Any of the 26 letters from A through Z can be used to identify the values we know, as well as those we want to figure out. Whenever possible, it's a good idea to choose letters that are abbreviations of the things they stand for -- like the D, R, and T for the Distance, Rate, Time equation.

To complicate this very simple example, there's an optional way of writing the equation, using the BASIC statement LET:

```
40  LET D = R * T
```

This use of LET reminds us that making D equal R times T was *our* choice, rather than an eternal truth like 2 = 1 + 1. Some computers are fussy and always require the use of LET with programmed equations. Our Macintosh says, "Whatever you want."

Okay -- let's complete the program.

Assume:

> Distance (in miles) = Rate (in miles per hour) multiplied by Time (in hours). How far is it from San Diego to London if a jet plane traveling at an average speed of 500 miles per hour makes the trip in 12 hours?
>
> (Yes, I know you can do that one in your head, but that's not the point!)

Type in the following below Line 40:

```
10 REM  * DISTANCE, RATE, TIME PROBLEM *  Return
20 R = 500    Return
30 T = 12     Return
```

Now use the Editor to Cut and Paste Line 40 to the end of the program where it belongs. After you have cleaned it up, it should look like:

```
10 REM  * DISTANCE, RATE, TIME PROBLEM *
20 R = 500
30 T = 12
40 D = R * T
```

Check the program carefully, then:

> Run.

Hum de dum...ho-hum...(this sure is a slow computer).

All it does is clear the screen, then reList the program. **The Computer Doesn't Work!**

Yes, it does. *It worked just fine.* The Computer multiplied 500 times 12 just like we told it and came up with the answer of 6000 miles. But *we* forgot to tell it to give *us* the answer. Sorry about that.

EXERCISE 5-1: Can you finish this program without help? It only takes one more Line. Give it a good try before reading on for the answer. That way, the answer will mean more to you. (Hint: We've already used PRINT to PRINT messages in quotes. What would happen if we said `50 PRINT "D"`? ... No, we want the *value* of D, not "D" itself. Hmmmm, what happens when we get rid of the quotes?)

Don't Read Beyond This Point Until You've Worked On The Above Exercise!

Look in Section B of this Manual for an answer to this Exercise.

Well, the answer 6000 is correct, but its "presentation" is no more inspiring than the readout on a hand calculator. This inevitably leads us back to where we first started this foray into the unknown -- the PRINT statement.

Did you find out the hard way that a space must be placed between the PRINT and the variable D? It *can't* be eliminated.

Note that we said `50 PRINT D`. There were no quotes around the letter D like we used before. The reason is simple but fairly profound. If we want the Computer to PRINT *the exact words* we specify, we enclose them in quotes. If we want it to PRINT the *value* of a variable, in this case D, we leave the quotes off. That simple message is worth serious thought before continuing on.

Did you think seriously about it? Then on we go!

Now suppose we want to include both the *value* of something *and* some exact words on the same Line. Pay attention, as you will be doing more and more program designing yourself, and PRINT statements give beginners more trouble than any other single part of computer programming. Use the Editor

to Cut out Line 50, then type in the following:

```
50 PRINT "THE DISTANCE (IN MILES) IS",D
```

Then:

Run.

The Display says:

```
THE DISTANCE (IN MILES) IS          6000
```

How about that! The message enclosed in quotes is PRINTed exactly as we specified, and the letter gave us the value of D. The comma told the Computer that we wanted it to PRINT two separate items on the *same* line.

With this in mind, see if you can Edit Line 50 so the Computer finishes the program with the following message:

```
THE DISTANCE IS   6000              MILES.
```

Answer: Break up the message words into two parts, and put the number variable in between them on the same PRINT Line.

```
50 PRINT "THE DISTANCE IS",D,"MILES."
```

Why is there all that extra space on both sides of the 6000 in the PRINTout? When a PRINT statement contains two or more items separated by commas, the Computer automatically PRINTs them in adjacent PRINT zones. *Automatic zoning* is a very convenient method of outputting TABular information, and we'll explore the subject in detail later on.

It's possible to eliminate the extra spaces in the display. Edit the last version of Line 50, substituting semi-colons (;) for the 2 commas.

(Careful -- don't replace the period with a semi-colon.)

Run.

Perfection, at last:

```
THE DISTANCE IS 6000 MILES.
```

Look carefully at the new Line 50. There is no blank space between the S in IS, the D, and the M in MILES. But in the display printout, there *is* a space between IS and 6000, and another space between 6000 and MILES. Why?

Reason: When a *number* is PRINTed (the *value* of D), leading and trailing blank spaces are automatically inserted. As we do more programming, this feature will become very important.

WHEW!

Well, we have already covered more than enough Commands, Statements and Math Operators to solve a myriad of problems.

Math Operators? They're the = + - * ^ / and \ symbols we mentioned earlier.

Now, let's spend some time actually writing programs to solve problems. There is no better way to learn than by doing, and *everything* covered so far is fundamental to our success in later Chapters. Don't jump over these exercises! They will plunge you right into the thick of programming, where you belong. Sample answers are in Section B, along with further comments.

EXERCISE 5-2: Write a program which will find the TIME required to travel by jet plane from London to San Diego, if the distance is 6000 miles and the plane travels at 500 MPH.

```
20 D = 6000
30 R = 500
40 LET T = D/R
50 PRINT "The Time Required is"; T; "hours."
```

EXERCISE 5-3: If the circumference of a circle is found by multiplying its diameter times pi (3.14), write a program which will find the circumference of a circle with a diameter of 35 feet.

```
10 D = 35
20 PI = 3.14
30 LET C = D * PI
40 PRINT "The circumference of the circle is ";C;"feet"
```

EXERCISE 5-4: If the area of a circle is found by multiplying pi times the square of its radius, write a program to find the area of a circle with a radius of 5 inches.

```
20 PI = 3.14
30 R = 5
40 A = PI x R x R
50 PRINT "The circle's area is";A;" Square Inches."
```

EXERCISE 5-5: Your checkbook balance was $225. You've written three checks (for $17, $35 and $225) and made two deposits ($40 and $200). Write a program to adjust your old balance based on checks written and deposits made, and PRINT out your new balance.

```
LET B = 225 - 17 - 35 - 225 + 40 + 200
PRINT "My Balance is $";B
```

(Different - but it worked!)

Learned In Chapter 5

Statements	Math Operators	Miscellaneous
LET (Optional)	=	,
	+	;
	−	Variable Names
	*	
	/	

Remember, we can use any of the 26 letters as variables, not just D, R, and T (they were just convenient for our problem).

Chapter 6

Scientific Notation

Are There More Stars Or Grains Of Sand?

In this mathematical world we are blessed with very large and very small numbers. Millions of these and billionths of those. To cope with all this, our Computer uses "exponential notation," or "standard scientific notation," when the number sizes start to get out of hand. The number 5 million (5,000,000), for example, can be written "5E+06" (E for Exponential), which means, "the number 5 followed by six zeros."

Or technically, $5*10^6$ which is 5 times ten to the sixth power: 5*10*10*10*10*10*10.

If an answer comes out "5E-06," that means we must shift the decimal point, which is after the 5, six places to the *left*, inserting zeros as necessary. Technically, it means 5×10^{-6}, or 5 millionths (.000,005).

In our BASIC, that's 5/10/10/10/10/10/10.

It's really pretty simple once you get the hang of it and makes it very easy to keep track of the decimal point. Since the Computer *insists* on using it with very large and very small numbers, we can just as well get used to it right now.

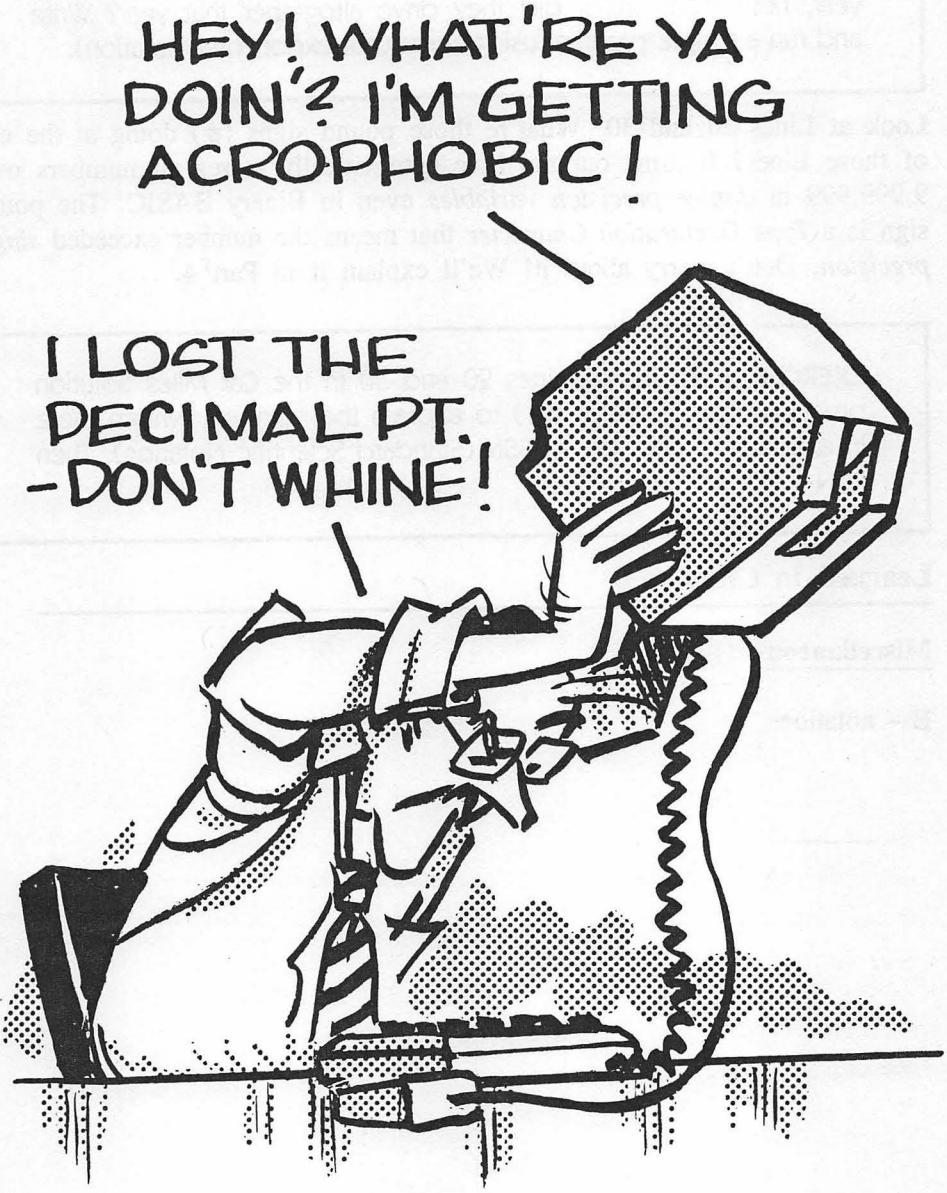
Type the following in the Command window:

```
PRINT 5*10^7
```

Return (The caret ^ is located above the 6 key.)

The answer is:

```
5E+07
```


HEY! WHAT'RE YA DOIN'? I'M GETTING ACROPHOBIC!
I LOST THE DECIMAL PT. -DON'T WHINE!

Select New before performing the following exercises.

> **EXERCISE 6-1:** If 100 million cars drove 10 million miles in a certain year, how many miles did they drive altogether that year? Write and run a simple program using zeros (not exponential notation).

Look at Lines 20 and 30. What're those pound signs (#) doing at the end of those Lines? It turns out the Mac automatically stores all numbers over 9,999,999 in *double precision variables* even in Binary BASIC. The pound sign is a *Type Declaration Character* that means the number exceeded *single precision*. Don't worry about it! We'll explain it in Part 4.

> **EXERCISE 6-2:** Change Lines 20 and 30 in the Car Miles Solution program (from Exercise 6-1) to express the numbers written there in exponential notation, or SSN (Standard Scientific Notation). Then RUN it.

Learned in Chapter 6

Miscellaneous

E - notation

Chapter 7

Using ( ) And The Order Of Operations

Parentheses play an important role in computer programming, just as in ordinary math. They are used here in the same general way, but there are important exceptions.

1. In BASIC, parentheses can enclose operations to be performed. Those operations which are within parentheses are performed before those *not* in parentheses.

2. Operations buried deepest within parentheses (that is, parentheses inside parentheses) are performed first.

To be sure equations are calculated correctly, use ( ) around the operations which must be performed first.

---

3. When there is a "tie" as to which operation the Computer should perform *after* it has solved all problems enclosed in parentheses, it works its way along the program Line from *left to right* performing the *multiplication* and *division*. It then starts at the left again and performs the *addition* and *subtraction*.

Recall the old memory aid, "My Dear Aunt Sally"? In math we do Multiplication and Division first (from left to right), then come back for Addition and Subtraction (left to right). Microsoft BASIC follows the same sequence.

INT, RND and ABS functions are performed before multiplication and division. (We haven't used them yet, but just to be completely accurate...)

---

4. An operation written as (X)(Y) will *not* tell the Computer to multiply. X * Y is the only scheme recognized for multiplication.


MY DEAR AUNT SALLY- I DIDN'T KNOW YOU WERE PART OF THIS!
THIS IS MY S-L-O-W NEPHEW!

EXAMPLE: To convert temperature expressed in degrees Fahrenheit to Celsius (Centigrade), the following relationship is used:

The Fahrenheit temperature equals 32 degrees plus nine-fifths of the Celsius temperature. Or, maybe you're more used to the simple formula:

$$F = \frac{9}{5} * C + 32$$

Assume we have a Celsius temperature of 25. Type in this New program and Run it:

```
10 REM * CELSIUS TO FAHRENHEIT CONVERSION *
20 C = 25
30 F = (9/5)*C + 32
40 PRINT C;"DEGREES (C) =";F;"DEGREES (F)."
```

SAMPLE RUN:

```
25 DEGREES (C) = 77 DEGREES (F).
```

Remember what the semi-colons are for?

Notice first that Line 40 consists of a PRINT statement followed by 4 separate expressions -- 2 variables and 2 groups of words in quotes called "literals," or "strings." Notice also that everything within the quotes (including spaces) is PRINTed.

Next, note how the parentheses are placed in Line 30. With the 9/5 securely inside, we can multiply its quotient times C, then add 32.

Now, remove the parentheses in Line 30 and Run again. The answer comes out the same. Why?

1. On the first pass, the Computer started by solving all problems within parentheses, in this case just one (9/5). It came up with (but did not PRINT) 1.8. It then multiplied the 1.8 times the value of C and added 32.

2. On our next try, without the parentheses, the Computer simply moved from left to right performing first the division problem (9 divided by 5), then the multiplication problem (1.8 times C), then the addition problem (adding 32). The parentheses really made no difference in *this* example.

Next, change the `+32` to `32+`, and move it to the front of the equation in Line 30 to read:

```
30 F = 32 + 9/5*C
```

Run it again.

Did it make a difference in the answer? Why not?

Answer: Execution proceeds from left to right, multiplication and division first, then returns and performs addition and subtraction. This is why the 32 was *not* added to the 9 before being divided by 5. **Very Important!** If they had been added, we would, of course, have gotten the wrong answer.

EXERCISE 7-1: Write and Run a program which converts 65 degrees Fahrenheit to Celsius. The rule tells us that "Celsius temperature is equal to five-ninths times what's left after 32 is subtracted from the Fahrenheit temperature."

$$C = (F - 32) \times \frac{5}{9}$$

EXERCISE 7-2: Remove the first set of parentheses in the Ex. 7-1 answer and Run again.

EXERCISE 7-3: Replace the first set of parentheses in program Line 30 and remove the second pair of parentheses, then Run. Note how the answer comes out -- correctly!

EXERCISE 7-4: Insert parentheses in the following equation to make it correct. Write a program to check it out on the Macintosh.

30 - 9 - 8 - 7 - 6 = 28

Learned In Chapter 7

Miscellaneous

( )
Order of Operations

Chapter 8

Relational Operators

If you liked the preceding Chapters, **then** you're going to love the rest of this book!

...because we're really just getting into the good stuff like IF-THEN and GOTO statements that let the Computer make decisions and take, um, er, executive action. But first, a few more operators.

Relational Operators allow the Computer to compare one value with another. There are only 3:

1. Equals, using the symbol =
2. Is greater than, using the symbol >
3. Is less than, using the symbol <

Combining these 3, we come up with 3 more operators:

4. Is not equal to, using the symbol <>
5. Is less than or equal to, using the symbol <=
6. Is greater than or equal to, using the symbol >=

Example: A<B means A is less than B. To help distinguish between < and >, just remember that the *smaller* (pointed) part of the < symbol points to the *smaller* of the two quantities being compared.

By adding these 6 *relational* operators to the *math* operators we already know, plus new *statements* called IF-THEN and GOTO, we create a powerful system of comparing and calculating that becomes the central core of everything that follows.

The IF-THEN statement, combined with the 6 relational operators above, gives us the *action* part of a system of logic. Enter and Run this New program:

```
10 A = 5
20 IF A = 5 THEN 50
30 PRINT "A DOES NOT EQUAL 5."
40 END
50 PRINT "A EQUALS 5."
```

The screen displays:

```
A EQUALS 5.
```

This program is an example of using an IF-THEN statement with only the most fundamental relational operator, the equals sign.

The Autopsy

Let's examine the program Line by Line.

Line 10 establishes the fact that A has a value of 5.

Line 20 is an IF-THEN statement which directs the Computer to GOTO Line 50, skipping over whatever might be in between Lines 20 and 50, *if the value of A is exactly 5*. Since A *does* equal 5, the Computer jumps to Line 50 and does as it says, PRINTing A EQUALS 5. Lines 30 and 40 were not used at all.

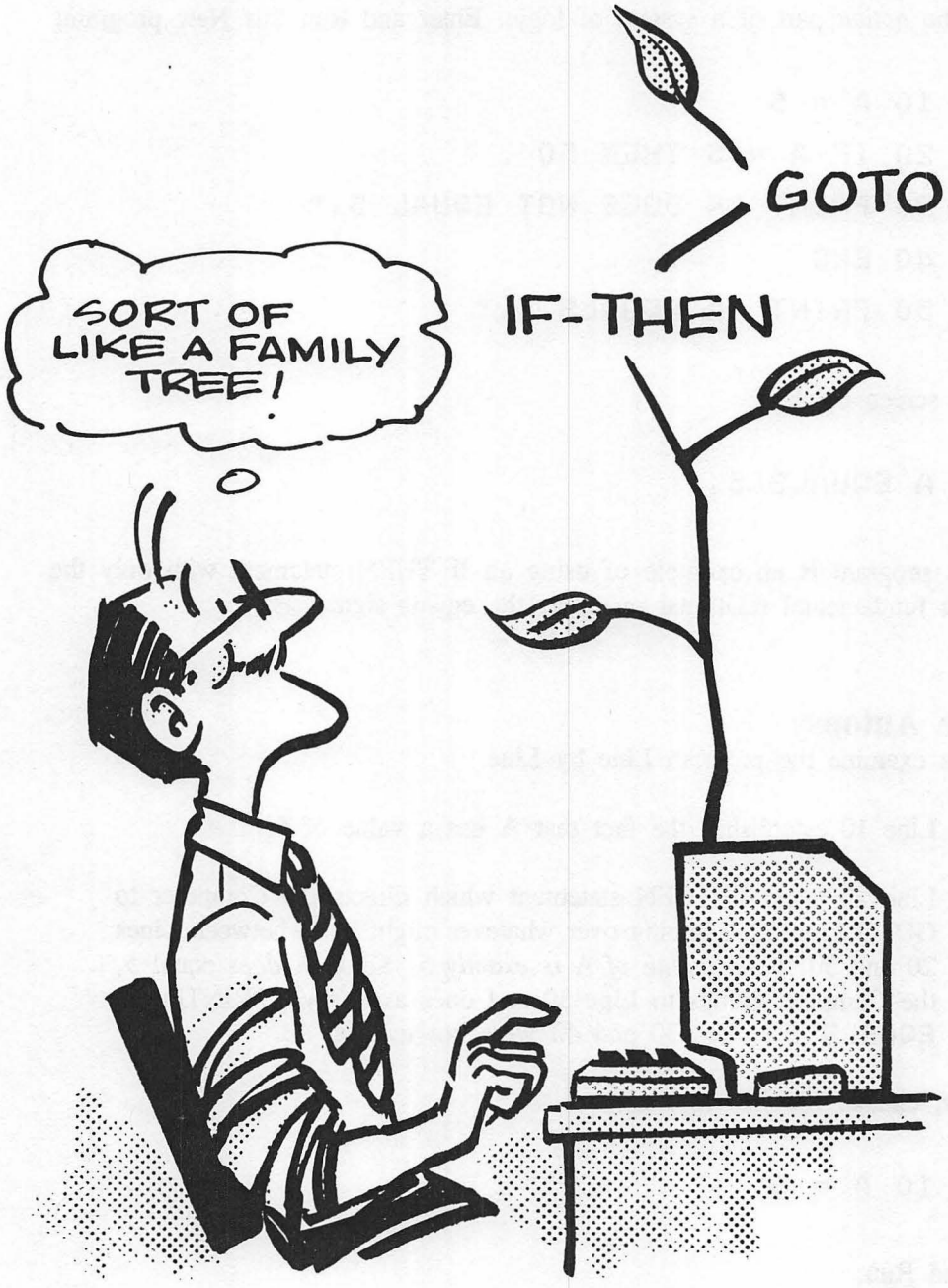
Now, change Line 10 to read:

```
10 A = 6
```

...and Run.

The screen says:

```
A DOES NOT EQUAL 5.
```


GOTO
IF-THEN
SORT OF LIKE A FAMILY TREE!

Taking it a Line at a time:

Line 10 establishes the value of A to be 6.

Line 20 tests the value of A. IF A equals 5, THEN the Computer is directed to GOTO Line 50. But "the test fails," that is, A does *not* equal 5, so the Computer proceeds as usual to the next Line, Line 30.

Line 30 directs the Computer to PRINT the fact that `A DOES NOT EQUAL 5`. It does not tell us what the *value* of A is, only that it does *not* equal 5. The Computer proceeds to Line 40.

Line 40 ENDs the program's execution. Without this statement separating Lines 30 and 50, the Computer would charge right on to

Line 50 and PRINT its contents, which obviously are in conflict with the contents of Line 30.

IF-THEN Vs. GOTO

IF-THEN is what is known as a *conditional* branching statement. The program will "branch" to another part of the program *on the condition that* it passes the IF-THEN test. If it fails the test, program execution simply passes to the next Line.

GOTO is an *unconditional* branching statement. If we were to replace Line 40 with:

```
40 GOTO 99
```

and add Line 99:

```
99 END
```

...whenever the Computer hit Line 40 it would *unconditionally* follow orders and GOTO 99, ENDing the Run. Change Line 40 as discussed above, and add Line 99 to the end of the program so the entire program appears as:

```
10 A = 6
```

```
20 IF A = 5 THEN 50
30 PRINT "A DOES NOT EQUAL 5."
40 GOTO 99
50 PRINT "A EQUALS 5."
99 END
```

...and Run.

Did the program work OK as changed? Did you try it with several values of A? Be sure you do! We will find many uses for the GOTO statement in the future.

Optional THEN With GOTO

When the IF-THEN statement is used with a GOTO statement, either THEN or GOTO or both can be used. This can be useful in long program Lines. For example, either of these Lines will work in place of Line 20 in our program:

```
20 IF A = 5 THEN GOTO 50
```

or

```
20 IF A = 5 GOTO 50
```

EXERCISE 8-1: Change the value of A in Line 10 back to 5 then rewrite the resident program using a "does-not-equal" sign in Line 20 instead of the equals sign. Change other Lines as necessary, so the same results are achieved with your program as with the one in the example.

_____

_____

_____

_____

_____

EXERCISE 8-2: Change Line 10 to give A the value of 6. Leave the other four Lines from Exercise 8-1 as shown. Add more program Lines as necessary so the program will tell us whether A is larger or smaller than 5 and Run.

EXERCISE 8-3: Change the value of A in Line 10 at least three more times, Running after each change to ensure that your new program works correctly.

No sample answers are given since you are choosing your own values of A. It will be obvious whether or not you are getting the right answer.

Learned In Chapter 8

Statements	Relational Operators	Miscellaneous
IF-THEN	=	Conditional branching
GOTO	>	Unconditional branching
	<	
	<>	
	<=	
	>=	

Chapter 9

It Also Talks And Listens

By now you have probably become tired of having to Edit Line 10 each time you wish to change the value of A. The INPUT statement is a simple, fast and more convenient way to accomplish the same thing. It's a biggie, so don't miss any points.

Enter this New program:

```
10 PRINT "THE VALUE I WISH TO GIVE A IS"
20 INPUT A
30 PRINT "A =";A
```

...and Run.

The Computer prints:

```
THE VALUE I WISH TO GIVE A IS
?
```

See the question mark on the screen? It means, "It's your turn -- and I'm waiting..."

Type in a number, and press Return to see what happens. The program responds exactly the same way as when we changed values within a program Line. Run several more times to get the feel of the INPUT statement.

Pretty powerful, isn't it?

Let's add a touch of class to the INPUT process by changing Line 10 as follows:

```
10 PRINT "THE VALUE I WISH TO GIVE A IS";
```

Look at that Line very carefully. Do you see how it differs from the earlier Line 10? It is different -- a *semi-colon* was added at the end.

Think back a bit. We used semi-colons before in PRINT statements, but only in the *middle,* to hook several together to PRINT them on the same line. In this case, we put a semi-colon at the *end,* so the *question mark* from the Line 10 will PRINT on the *same* display line rather than on a second line. After changing Line 10 as above, Run. It should read:

```
THE VALUE I WISH TO GIVE A IS?
```

We cannot use a semi-colon indiscriminately at the end of a PRINT statement. It is only meant to hook two lines together, *both* of which will PRINT something. The INPUT Line PRINTs a question mark. We will later connect two long Lines starting with PRINT by a "trailing semicolon" so as to PRINT everything on the same line.

The Microsoft BASIC *interpreter* speaks "The King's BASIC" as well as a variety of dialects. The first of the many "short-cuts" we will learn combines PRINT and INPUT into one statement.

INTERPRETER -- is the program we loaded in from disk which allows us to "rap" with the Computer in the English language. The program is called BASIC, which stands for Beginners All-purpose Symbolic Instruction Code.

Sometimes the word "dialect" is used when talking about the different variations of a computer language. Just as with dialects in "human" languages, there are differences in the way different computers use BASIC words. That's why I wrote *The BASIC Handbook, Encyclopedia of the BASIC Language* available at better Computer and Bookstores everywhere in English, and translated into French, German, Swedish, Norwegian, Dutch, Italian, Spanish and Hebrew.

Change Line 10 to read:

```
10 INPUT "TYPE IN A VALUE FOR A";A
```

delete Line 20 by cutting it out with the Editor

...and Run.

The results come out exactly the same, don't they? Here is what we did:

1. Changed PRINT to INPUT

2. Placed both statements on the same Line

3. Eliminated an unnecessary Line

In the long programs which we will be writing, Running and converting, this shortcut will be valuable.

Endless Love

Up to now, all our programs have been strictly one-shot affairs. You Run it; the Computer executes it, PRINTS the results (if any), and comes back with a flashing cursor in the List window. To repeat the program, we have to Run it again. Can you think of another way to make the Computer execute a program two or more times?

No -- don't enlarge the program by repeating its Lines over and over again -- that's not very creative!

We'll answer that question by upgrading our Celsius-to-Fahrenheit conversion program (Chapter 7). If you think GOTO is a powerful statement in everyday life, wait 'til you see what it does for a computer program!

Select New and type the following:

```
10 REM * IMPROVED (C) TO (F) CONV. PROGRAM *
20 INPUT "WHAT IS THE TEMP IN DEGREES (C)";C
30 F = (9/5)*C + 32
40 PRINT C;"DEGREES (C) =";F;"DEGREES (F)."
50 GOTO 20
```

...and Run.

Use [Shift] [.] to exit the program loop, and [Shift] [L] to List the program.

The Computer will keep asking for more until we get tired, and stop it or the power goes off (or some other event beyond its control). This is the kind of


I'LL KEEP ASK-
IN' FOR MORE
UNTIL YOU HIT
BY GEORGE!
I THINK I'VE
GOT IT!!

thing a computer is best at -- doing the same thing over and over. Modify some of the other programs to make them self-repeating. They're often much more useful this way.

These have been 4 long and "meaty" lessons, so go back and review them all, repeating those assignments where you feel weak. We are moving out into progressively deeper water, and complete mastery of these *fundamentals* is your only life preserver.

Learned In Chapter 9

Statements	Miscellaneous
INPUT and INPUT with built-in PRINT	; Trailing semi-colon

Chapter 10

Calculator Or Immediate Mode

Two Easy Features

Before continuing exploration of the nooks and crannies of the Computer acting as a *computer*, we should be aware that it also works well as a *calculator*. If we enter the Command window by either selecting it from the Windows menu or by clicking inside the Command window, the Computer will execute certain statements and commands and display the answer on the screen. What's more, it will work as a calculator even when another computer program is loaded, *without disturbing that program.* All we need, to be in the calculator mode, is to be in the Command window with the flashing cursor.

We won't be using the List window for these examples so remove it by clicking in the List window close box.

EXAMPLE: How much is 3 times 4? With the flashing cursor in the Command window, type in:

```
CLS          Return
PRINT 3 * 4          Return
```

...the answer comes back in the Output window:

```
12
```

CLS CLears the Output window. It is a very unfussy statement which you will want to use to make room for new output. Later you'll see how using CLS will make your displays neat and impressive.

EXAMPLE: How much is 345 divided by 123?

Type:

```
PRINT 345/123          Return
```

...the answer is:

```
2.804878
```

Spend a few minutes making up routine arithmetic problems of your own, and use the calculator mode to solve them. Any arithmetic expression which can be used in a program can also be evaluated in the calculator mode. This includes parentheses and chain calculations like A*B*C.

Try the following:

```
PRINT (2/3)*(3/2)          Return
```

The answer is:

```
1
```

Calculator Mode For Troubleshooting

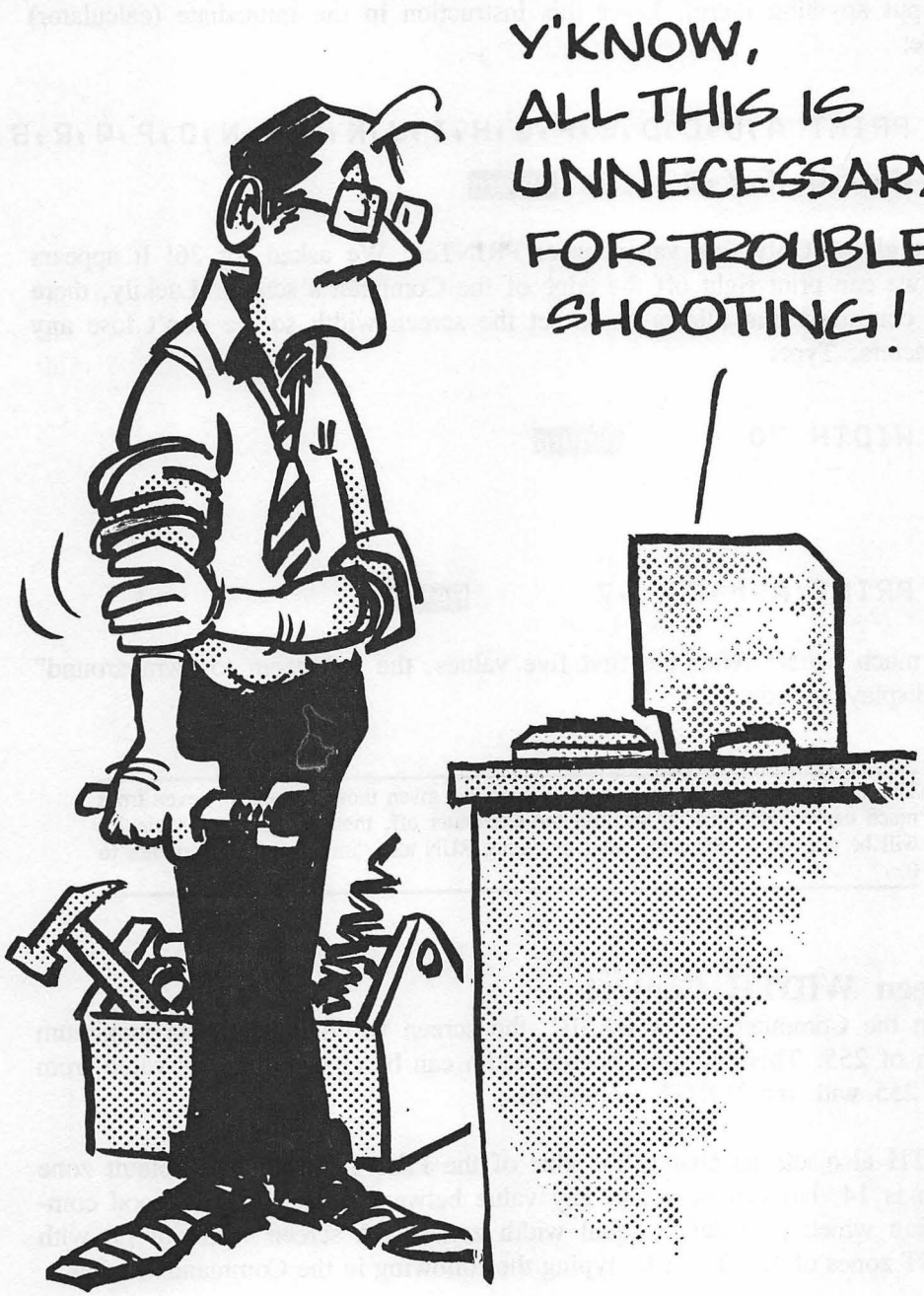
Suppose a program isn't giving the answers we expect. How can we troubleshoot it? One way is to ask the Computer to tell us what it knows about the variables used in the resident program.

EXAMPLE: If our program uses the variable X, we can ask the Computer to:

```
PRINT X          Return
```

The Computer will PRINT the present value of X.

Keep this handy tip in mind as you get into more complex programs.


Y'KNOW,
ALL THIS IS
UNNECESSARY
FOR TROUBLE
SHOOTING!

Another thought: *Something* is stored in every memory cell (even if *you* have not put anything there). Enter this instruction in the immediate (calculator) mode:

```
PRINT A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,
U,V,W,X,Y,Z          Return
```

What gives? Only five values were PRINTed! We asked for 26! It appears that we can print right off the edge of the Computer's screen. Luckily, there is a command that allows us to set the screen width so we don't lose any characters. Type:

```
WIDTH 70          Return
```

and

```
PRINT A,B,C,...,Z          Return
```

Ah, much better! After the first five values, the rest seem to "wraparound" the display window.

The answers you got depend on the values last given those variables -- even from much earlier programs. If we turn the Computer off, then on again, all variables will be reset to 0. Selecting Start or typing RUN also "initializes" all variables to 0.

Screen WIDTH Control

When the Computer enters BASIC, the screen width is set at its maximum width of 255. This "normal" screen width can be changed to any value from 1 to 255 with the WIDTH command.

WIDTH also lets us change the size of the PRINT zones. The default zone width is 14, but can be set to any value between 1 and 255. A good combination which provides 5 equal width zones is a screen width of 60 with PRINT zones of 12. Try it by typing the following in the Command window:

```
WIDTH 60,12          Return
```

Now test it by typing a Line containing more than 60 characters. Notice how the Computer refuses to display a line longer than 60 columns? Reset the WIDTH to the default (maximum) size by typing:

```
WIDTH 255          Return
```

The FRE(0) Function

Since programs do occupy space in the Computer's memory and program size is limited to how much memory is installed, it may be important to know how much memory is left. That's what the FRE(0) Function is for.

In a "128K" computer there are about 128,000 different memory locations available to store and process programs. "128K" is just a shortcut phrase for the exact amount of memory, which is 131072.

> This manual is meant to be for the computer operator and programmer, so we are studiously avoiding computer electronics theory -- when possible.

The Computer uses some of the memory for program control. To see the actual amount of memory available for our use, type NEW in the Command window or select New from the File menu. Then activate the Command window, and type:

```
PRINT FRE(0)          Return
```

> 0 is a "dummy" value used with FRE. Any number or letter can be used.

...and the answer is:

```
21000
```

With no program loaded, it means there are 21000 memory locations available for use. The difference in memory space between 21000 and 131072 is used by the BASIC language interpreter and overall management and "monitoring" of what the Computer is doing.

Activate the List window, and type in this simple program:

```
10 A = 25          Return
```

then measure the memory remaining by entering the Command window and typing:

```
PRINT FRE(0)       Return
```

The answer is:

```
20984
```

The program we entered took 21000 - 20984 = 16 bytes of space. Here is how we account for it:

1. Each Line number and the space following it (regardless of how small or large that Line number is) occupies 6 bytes. The "carriage return" at the end of the Line takes 4 more bytes, even though it does not print on the display. Thus, memory "overhead" for each Line, short or long, is 10 bytes.

2. In the above program, `10 A = 25` takes a total of 16 bytes. That's 10 bytes for overhead plus 6 bytes for the characters (10 + 6 = 16 bytes).

BYTE -- is the basic unit of storage for the Macintosh and most other microcomputers. In the Macintosh it is a string of sixteen binary digits (bits). Thus a byte = 16 bits.

We will be studying memory requirements in more detail later.

Obviously, the short learning programs we have written so far are not taking up much memory space. This changes quickly, however, as we move to more sophisticated programming. Make a habit of typing `PRINT FRE(0)` when completing a program to develop a sense of its size and memory requirements.

Learned In Chapter 10

Functions	Commands	Miscellaneous
FRE(0)	WIDTH	Calculator (Direct Command) Mode
		Memory
		Byte

Chapter 11

SAVEing And LOADing Using Disk

A big advantage of having disk drives is that programs can be SAVEd on or Opened (LOADed) from disk very quickly and reliably.

With a one-drive system, programs are automatically SAVEd on the diskette inside the Macintosh, unless the disk is locked (write-protected). With a two-drive system, the diskette inside the Macintosh serves as an application disk (i.e. has BASIC on it) and the one in the external drive serves as the data disk to which information would be written.

The Macintosh can have 2 drives; a second external drive or a high speed internal "hard disk" drive can be added.

Type in this short New BASIC program:

```
10 REM * SAVE THIS PROGRAM *
20 PRINT "HELLO THERE, DISKETTE!"
99 END
```

then, Save it on disk As:

```
PROGRAM1
```

by selecting Save As from the File menu, typing `PROGRAM1` when the dialog box appears, and clicking the pointer in the **Save** box (or simply pressing Return).

Don't confuse Save As with the Save option. Save is used when saving a program with the same name that is shown on the Title Bar (no, "Untitled" should not be used).

Well, something seemed to happen. Our little program is now Saved on the disk under the name PROGRAM1.

Now, let's recall the program from the disk. First, from the File menu, choose New to clear the program out of memory.

Notice that no trace remains in the List window. Good thing we Saved it on diskette. Hope it's really there.

To see what is on the disk, display the disk FILES. Select Show Command from the Windows menu, and simply type:

```
FILES          Return
```

Yes, there it is, the last program listed.

Programs can also be SAVEd from the keyboard by simply entering the Command window, and typing:

```
SAVE "program name"          Return
```

Quotation marks are required around file names only when SAVEing or LOADing a file from the keyboard. If they're omitted, a "Type mismatch" error results.

To copy the program from the diskette back into memory, choose Open... from the File menu, click the mouse over the file name PROGRAM1, then click in the **Open** box.

Programs can also be opened by double-clicking on the program name.

and in it comes.

To LOAD a program from the keyboard, just enter the Command window, and type:

```
LOAD "program name"          Return
```

Don't forget the quotation marks.


WELL, OUR SAVINGS PROGRAM DOES NOT INCLUDE DISKS!
TELLER

File names can contain as few a 1 character or as many as 63. They can include any characters (even blank spaces) except the colon and quotation marks.

While we're on the subject, you can Save a program to a different disk by selecting the **Eject** box in the Save As... dialog box. The Computer ejects the diskette, and after you insert another and click the **Save** box, it prompts you to swap disks until the program is Saved.

There are three format options available at the bottom of the Save As... box. The Computer normally Saves programs in the *Compressed* mode to conserve disk space. There are times when we will sacrifice a little disk space for the luxury of Saving a program "character for character" by selecting the *Text* format, or what is often referred to as the "ASCII format."

Programs *must* be Save in the ASCII format if they are to be used with other application software. Later we will MERGE programs that have been Saved in ASCII format.

The third format option allows us to Protect the program. By selecting this option, the program cannot be LISTed or RUN. Doesn't leave much for us to do with it except to Trash it.

Learned In Chapter 11

Commands	Miscellaneous	Menu
LOAD	File names	File
FILES	Text format	Save As...
SAVE	Compressed format	Save
	Protected programs	Open...

Chapter 12

FOR-NEXT Looping

A major difference between a Computer and a calculator is the Computer's ability to do the same thing over and over an outrageous number of times! This single capability (plus, a larger display) more than any other feature distinguishes between the two.

The FOR-NEXT loop is of such overwhelming importance in putting our Computer to work that few of the programming areas we explore from here on will exclude it. Its simplicity and variations are the heart of its effectiveness, and its power is truly staggering.

Type NEW in the Command window and then type the following program in the List window:

```
20 PRINT "HELP! MY COMPUTER IS BERSERK!"
40 GOTO 20
```

...and Run.

The Computer is PRINTing:

```
HELP! MY COMPUTER IS BERSERK!
```

and will do so indefinitely, until we tell it to STOP. When you have seen enough, Stop (⌘.). This "breaks" the program Run. Now, List (⌘L) the program.

Endless Loop

We created what is called an "endless loop." Remember our earlier programs which kept coming back for more INPUT? They were in a very similar "loop."

Line 40 is an unconditional GOTO statement which causes the Computer to cycle back and forth ("loop") between Lines 20 and 40 forever, if not halted. This idea has great potential if we can harness it.

Modify the program to read:

```
10 FOR N = 1 TO 5
20   PRINT "HELP! MY COMPUTER IS BERSERK!"
40 NEXT N
60 PRINT "NO --- IT'S UNDER CONTROL."
```

...and Run it.

The line:

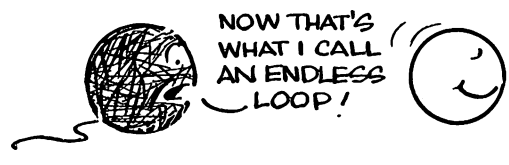
```
HELP! MY COMPUTER IS BERSERK!
```

was PRINTed 5 times, then:

```
NO --- IT'S UNDER CONTROL.
```

The FOR-NEXT loop created in Lines 10 and 40 caused the Computer to cycle through Lines 10, 20, and 40 exactly 5 times, then continue through the rest of the program. Each time the Computer hit Line 40, it saw "NEXT N." The word NEXT caused the value of N to increase (or STEP) by exactly 1. The Computer "conditionally" went back to the `FOR N =` statement that *began* the loop.

Execution of the NEXT statement is "conditional" on N being less than or equal to 5 because Line 10 says FOR N = 1 TO 5. After the 5th pass through the loop, the built-in test fails, the loop is broken and program execution moves on. The FOR-NEXT statement harnessed the endless loop!



The Step Function

There are times when it is desirable to increment the FOR-NEXT loop by some value other than 1. The STEP function allows it. Change Line 10 to read:

```
10 FOR N = 1 TO 5 STEP 2
```

...and Run.

Line 20 was PRINTed only 3 times (when N=1, N=3, and N=5). On the first pass through the program, when NEXT N was hit, it was incremented (or STEPped) by the value of 2, instead of the default value of 1. On the second pass through the loop, N equaled 3. On the third pass N equaled 5.

FOR-NEXT loops can be STEPped by any decimal number, even negative numbers. Why we would want to STEP with negative numbers might seem vague at this time, but that too will be understood with time. Meanwhile, change the following Line:

```
10 FOR N = 5 TO 1 STEP -1
```

...and Run.

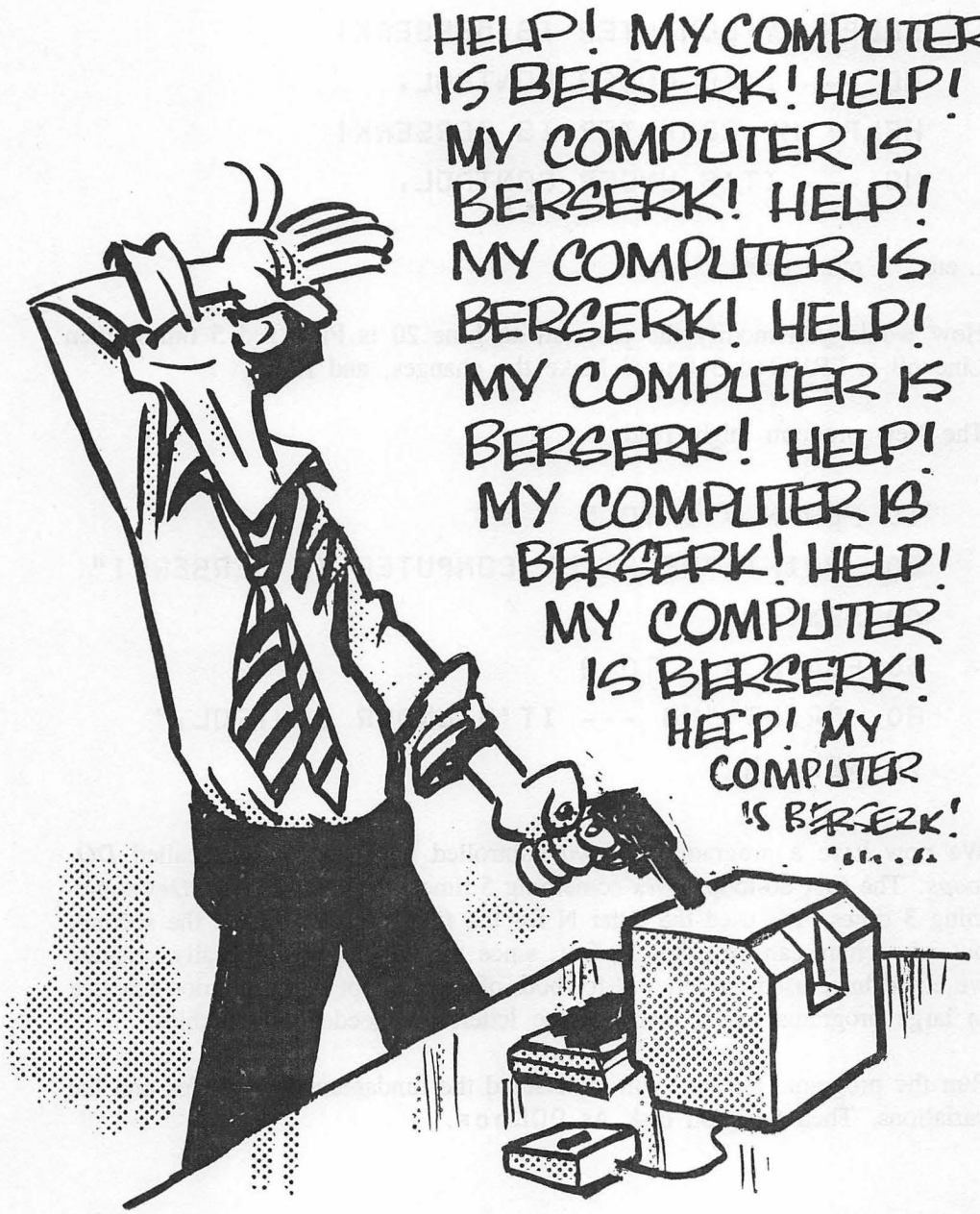
Five passes through the loop stepping *down* from 5 to 1 is exactly the same as stepping *up* from 1 to 5. Line 20 was still PRINTed 5 times. Change the STEP from -1 to -2.5 and Run again.

Amazing! It PRINTed exactly twice. Smart Computer. Change the STEP back to -1.

Modifying The FOR-NEXT Loop

Suppose we want to PRINT both Lines 20 and 60 five times, alternating between them. How will you change the program to accomplish it? Go ahead and make the change.

HINT: If you can't figure it out, try moving the NEXT N Line to another position.


HELP! MY COMPUTER IS BERSERK! HELP! MY COMPUTER IS BERSERK! HELP! MY COMPUTER IS BERSERK! HELP! MY COMPUTER IS BERSERK! HELP! MY COMPUTER IS BERSERK! HELP! MY COMPUTER IS BERSERK!

Right -- we moved Line 40 to Line 70, and the screen reads:

```
HELP! MY COMPUTER IS BERSERK!
NO --- IT'S UNDER CONTROL.
HELP! MY COMPUTER IS BERSERK!
NO --- IT'S UNDER CONTROL.
```

...etc., 3 more times.

How would you modify the program so Line 20 is PRINTed 5 times, then Line 60 is PRINTed 3 times? Make the changes, and Run.

The New program might read:

```
10 FOR N = 1 TO 5
20  PRINT "HELP! MY COMPUTER IS BERSERK!"
40 NEXT N
50 FOR M = 1 TO 3
60  PRINT "NO --- IT'S UNDER CONTROL."
70 NEXT M
```

We now have a program with *two* controlled loops, sometimes called *DO-loops*. The first do-loop *DOes* something 5 times; the second one *DOes* something 3 times. We used the letter N for the first loop and M for the second, but any letters can be used. In fact, since the two loops are totally separate we could have used the letter N for both of them -- not an uncommon practice in large programs where many of the letters are needed as variables.

Run the program. Be sure you understand the fundamental principles and the variations. Then Save on disk As `DOLoop`.

Incremental Looping

There is nothing magic about the FOR-NEXT loop; in fact, you may have already thought of another (longer) way to accomplish the same thing by using

features we learned earlier. Stop now, and see if you can figure out a way to construct a workable do-loop substituting something else in place of the FOR-NEXT statement.

Answer:

```
10 N = 1
20 PRINT "HELP! MY COMPUTER IS BERSERK!"
30 N = N + 1
40 IF N < 6 THEN 20
60 PRINT "NO --- IT'S UNDER CONTROL."
```

Line 10 *initializes* the value of N, giving it an *initial*, or beginning, value of 1. Without initializing, N could have been any number from a previous program or program Line. Note that selecting Run, or typing RUN, automatically resets all variables back to 0 before the program executes.

Initialize: initially, or at the beginning, establishes the value of a variable.

Line 30 *increments* it by 1, making N one more than whatever it was before. Line 40 uses one relational operator, <, to check that the new value of N is within the bounds we have established. If not, the test fails and the program continues.

Increments: STEPs (increases or decreases) values by specific amounts: by 1's, 3's, 5's, or whatever.

Note that in this system of *incrementing* and testing we do not send the program back to Line 10 as was the case with FOR-NEXT. What would happen if we did?

Answer: We would keep re-initializing the value of N to equal 1 and would again form an endless loop.

The opposite of *incrementing* is *decrementing*. Change the program so Line 30 reads:

```
30 N = N - 1
```

To *decrement* is to make smaller.

...then make other changes as needed to make the program work.

The changed Lines read:

```
10 N = 6
30 N = N - 1
40 IF N>1 THEN 20
```

Putting FOR-NEXT To Work

It isn't very exciting just seeing or doing the same thing over and over. The FOR-NEXT loop has to have a more noble purpose. It has many, and we will be learning new ones for a long time.

Suppose we want to PRINT out a chart showing how the time it takes to fly from London to San Diego varies with the speed at which we fly. (Remember, the formula is D = R*T.) Let's PRINT out the flight time required for each speed between 100 mph and 1000 mph, in increments of 100 mph. The program might look like this:

```
10 REM * TIME VS RATE FLIGHT CHART *
20 D = 6000
30 PRINT "  LONDON TO SAN DIEGO"
40 PRINT " DISTANCE =";D;"(MILES)"
50 PRINT "RATE (MPH)","TIME (HOURS)"
60 PRINT
70 FOR R=100 TO 1000 STEP 100
```

```
80    T = D/R
90  PRINT R,T
100 NEXT R
```

Type in the program, select Show Output from Windows menu to remove the List window from the display, then Run.

How about that...? Try doing that one on the old slide rule or hand calculator!

It is really solving the D = R*T problem 10 times in a row, for different values and PRINTing out the result. The screen should look like this:

```
      LONDON TO SAN DIEGO
   DISTANCE = 6000 (MILES)
RATE (MPH)       TIME (HOURS)

 100                60
 200                30
 300                20
 400                15
 500                12
 600                10
 700                8.571428
 800                7.5
 900                6.666667
 1000               6
```

Analyzing The Program

Press ⌘L, and look through the program. Observe these many features before we do some exercises to change it:

1. The REM statement identifies the program for future use.

2. Line 20 *initializes* the value of D. D will remain at its initialized value.

3. Lines 30 through 60 PRINT the chart heading.

4. Line 50 uses *automatic zone spacing* (the comma) to place those column headings, and Line 60 PRINTs a blank line.

Remember zone spacing? The comma (,) in a PRINT statement automatically starts the PRINTing in the next PRINT zone. We define the WIDTH of that zone. It is the second value in the WIDTH command, i.e., in WIDTH 60,12, 60 is the screen width, and 12 is the zone width.

The WIDTH command can be built right into a program. Try adding:

```
15 WIDTH 30,10
```

...and RUN. Then:

```
15 WIDTH 70,35
```

Experiment with different values, ending up with:

```
15 WIDTH 60,12
```

---

5. Line 70 established the FOR-NEXT loop complete with a STEP. It says, "Initialize the rate (R) at 100 mph, and make passes through the 'do-loop' with values of R incremented by values of 100 mph until a final value of 1000 mph is reached." Line 100 is the other half of the loop.

6. Line 80 contains the actual formula which calculates the answer.

7. Line 90 PRINTs the two values. They are positioned under their headings by automatic zone spacing (the commas).

8. Lines 80 and 90 are indented from the rest of the program text. This is a simple programming technique which highlights the do-loop and makes reading and troubleshooting easier. *Try to adopt good programming practices like this* as you do the exercises. Indenting does take up a little memory space and, on long programs, is sometimes omitted.

Take a deep breath, and go back over any points you might have missed in this lesson. Save the program onto disk As `LONDON1` because we will use it in the next Chapter, continuing our study of FOR-NEXT loops.

Learned In Chapter 12

Statements	Miscellaneous	Menu
FOR-NEXT	Increment	Windows
STEP	Decrement	Show Output
	Initialize	
	"Do-Loop"	
	Indenting program Lines	

Chapter 13

Son Of FOR-NEXT

This is heady stuff. If you turned the Computer off between Chapters, LOAD in the LONDON1 program which we SAVEd in the last Chapter.

Modify the program so the rate and time are calculated and PRINTed for every 50 mph increment instead of the 100 mph increment presently in the program.

...and Run.

Answer: `70 FOR R = 100 TO 1000 STEP 50`

Trouble In The Old Corral

What a revolting development! The PRINTout goes so fast we can't read it, and by the time it stops, the top part is cut off. *Aught'a known you can't trust these computers!*

Solutions For Sale

Several solutions are available:

1. Placing the pointer on the Edit, Run or Windows menu and pressing the mouse button will stop the execution of a program or the listing of files until the button is released.

2. Choosing Suspend from the Run menu (or pressing the ⌘ and the S keys *at the same time)* will halt program execution or LISTing. Pressing almost any other key will start it again. Run the program several times, and practice stopping and starting using this method.

There's another solution we must try. While the program is RUNning, -- choose Stop (or press the ⌘ and . keys) from the Run menu. While Suspend (⌘S) can be thought of as just pressing in the clutch, a Stop (⌘.) is more like turning off the engine.

To restart execution after a Stop, either select Start from the Run menu (or type RUN in the Command window) to start all over again from the beginning, or choose Continue to continue execution from the "break-point." Choosing Continue (or typing CONT in the Command window) does not reset all variables back to zero, which can be an important consideration.

3. For a classy display we can build a "pause" *into the program.* The screen will fill, pause a moment, then automatically continue if we don't interrupt execution.

The Timing Loop

It takes time to do everything. Even Macintosh takes time to do some things, though we may be awed by its speed.

We are going to write and experiment with a timing program using Lines 1-9 without erasing the one already resident. The new one must END without plowing ahead into the LONDON1 program, thus, Line 9. Insert the cursor at the front of Line 10, and type Lines 1-9 hitting Return at the end of each.

```
1 REM * TIMER PROGRAM *
4 PRINT "DON'T GO AWAY"
5 FOR X = 1 TO 22000
6 NEXT X
7 PRINT "TIMER PROGRAM ENDED."
9 END
```

...and Run.

Remember back when we learned *not* to do this (number Lines in tight sequence)? Well ... if we *hadn't* followed that rule with our LONDON1 program, we wouldn't have this nice space to demonstrate the point.

How long did it take? Well, it did take time, didn't it? About 10 seconds


SHHHH
DON'T DISTURB
'IM! HE'S IN
THE MIDDLE
OF A LOOP!

from the time the Computer displayed DON'T GO AWAY until it displayed TIMER PROGRAM ENDED. Microsoft BASIC can execute approximately 2200 FOR-NEXT loops per second. That means, by specifying the number of loops, we can build in as long a time-delay as we wish.

Change the program to create a 30-second delay. Time it against your watch or clock to see how accurate it is.

Answer:

```
5 FOR X = 1 TO 66000
```

EXERCISE 13-1: Using the space in Lines 1 through 7, design and Run a program which:

1) Asks us how many seconds delay we wish, allows us to enter a number, then executes the delay and reports back at the end that the delay is over and how many seconds it took. A sample answer is in Section B.

How To Handle Long Program LISTings

We now have **two** programs in the Computer. Double click inside the List window title bar to enlarge the List window to full size. My, my -- the program fills the entire List window, and the last Lines of the second program are chopped off. Now what do we do?

Rather than wring our hands about the problem, try each of the following solutions and watch the screen very carefully as each does its thing.

To LIST a program beginning with a specific Line:
Type LIST and the Line number in the Command window. The program, beginning with the specified Line number, will appear in the List window. (This LISTing can be done without activating the List window.)

To LIST a particular portion of the window:
Point to the scroll box, hold down the button and drag the scroll

box down (or up). When you release the button, a portion of the program will pop into view. Scrolling the box toward the top, center, or bottom of the scroll bar will LIST respectively the top, center, or bottom portion of the program.

To scroll the LISTing one Line at a time:
Move the pointer to and click the down arrow. The program will scroll upwards one Line at a time. Click the up arrow to scroll the program downwards.

Is There No End To This Magic?

To RUN the first program resident in the Computer -- we just type RUN. To RUN the second one we have a variation of RUN called:

```
RUN ###
```

The #'s represent the number of the Line we want the RUN to start with.

...and as you might suspect, it is similar to LIST ###. To RUN the program starting with Line 10, select Show Command window, and type:

```
RUN 10
```

...and that's just what happens.

Don't forget the space between RUN and 10. The Macintosh is fussy about some of these things.

Will wonders never cease? If there are 20 or 30 programs in the Computer at the same time, we can RUN just the one we want, provided we know its starting Line number. What's more, we can start any program in the middle (or elsewhere) for purposes of troubleshooting -- something we will do as our programs get longer and more complicated.

Remember: Using RUN reinitializes all variables to zero. If you want to preserve the current values, use GOTO ###.

Meanwhile, Back At The Ranch

We got into this whole messy business trying to find a way to slow down our RUN on the flight times from London to San Diego. In the process we found out a lot more about the Computer and learned to build a timer loop. Now let's see if we can build a pause right into the Distance program. First, erase the test program by typing the command:

```
DELETE 1-9          Return
```

Don't forget the space after DELETE.

Wow! How's that for power? It DELETEd those Lines, without having to cut out each individual Line Number with the Editor.

Wrong Way Computer

One way to STOP the fast parade of information is to put in a STOP. Insert:

```
75 IF R = 600 THEN STOP
```

...and Run.

We know R is going to increment from 100 to 1000. 600 is a little more than half the way to the end. See how the chart PRINTed out to 550 mph, then hit the STOP as 600 came racing down to Line 75? The Output window displays the first half of the chart, then the Computer beeps, flashes:

```
Program stopped          (in the upper right)
```

and draws a block around the contents of Line 75. This means the program is STOPped, or broken, in Line 75. To restart the program merely choose

Continue (from the Run menu) or enter the Command window and type:

```
CONT          Return
```

It automatically picks up where it left off and PRINTs the rest of the chart, or executes until it hits another STOP.

It may be desirable to change the size of the Command window and the Output window a bit to make the London1 desktop as large as possible.

At Last

The ultimate plan is to build a timer into the program so as not to completely STOP execution, but merely delay it for study.

Insert:

```
73   IF R <> 600 THEN 80
74   FOR X = 1 TO 11000
75   NEXT X
```

Be sure to Cut out the old Line 75.

...and Run.

Hey! It really works! As long as R does *not* equal 600, the program skips over the delay loop in Lines 74 and 75. When R *does* equal 600, the test "falls through" and Lines 74 and 75 "play catch" 11000 times, delaying the program's execution for about 5 seconds.

Time For A Cool One

It's been a long and tortuous route with numerous scenic side trips, but we finally made it. You picked up so many smarts in these 2 lessons on FOR-NEXT, that it's your turn to put them to work.

EXERCISE 13-2: Modify the resident program so that in this heading, (MPH) appears *below* RATE, and (HOURS) appears *below* TIME. This one should be a breeze.

EXERCISE 13-3: Design, write and Run a program which will calculate and PRINT income at yearly, monthly, weekly and daily rates, based on a 40-hour week, a 1/12th-year month, and a 52-week year. Do this for yearly incomes between $5,000 and $20,000 in $1,000 increments. Document your program with REM statements to explain the equations you create.

Some of the exercise programs are becoming too long to leave work space for your ideas. From now on, use a pad of paper for working up the answers.

EXERCISE 13-4: Here's an old chestnut that the Computer really eats up: Design, write and Run a program which tells how many days we have to work, starting at a penny a day, so if our salary doubles each day, we know which day we earn at least a million dollars. Include columns which show each day's number, its daily rate, and the total income to-date. Make the program stop after PRINTing the first day our daily rate is a million dollars or more. (After that ... who cares?)

Answers to these exercises are found in Section B.

The "Brute Force" Method

(Subtitled: Get A Bigger Hammer)

Much to the consternation of some teachers, a great value of the Computer is its ability to do the tedious work involved in the "cut and try," "hunt and peck" or other less respectable methods of finding an answer (or attempting to prove the correctness of a theory, theorem or principle). This method involves trying many possible solutions to see if one fits, or to find the closest one, or establish a trend. Beyond that, it can be a powerful learning tool by providing reams of data in chart or graph form which would simply take too

long to generate by hand. For example:

EXERCISE 13-5: You have a 1000 foot roll of fencing wire and want to make a *rectangular* pasture.

Using all of the wire, determine what length and width dimensions will allow you to enclose the maximum number of square feet? Use the brute force method; let the Computer try different values for L and W and PRINT out the Area fenced by each pair of L and W.

The formula for area is Area = Length times Width, or A=L*W.

EXERCISE 13-6: *Extra credit problem for "electronics types"*

As a further example (more complex and tends to prove the point better) try this final (optional) assignment. It involves a problem confronted by every electricity student who has studied SOURCES (batteries, generators) and LOADS (lights, resistors).

The *Maximum D.C. Power Transfer Theorem* states,

"Maximum DC power is delivered to an electrical load when the resistance of that load is equal in value to the internal resistance of the source."

And then the arguments begin...

> "Use a HIGH resistance load because it will drop more voltage and accept more power." ($P=V^2/R$)
>
> "No, use a LOW resistance load so it will draw more current and accept more power." ($P=I^2*R$)
>
> "Use a load value somewhere in between." ($P=I*V$)

Don't necessarily shy away from this problem if electricity doesn't happen to be your bag. Enough information is given to write the program. The principle, the optimizing of a value, is applicable to many fields of endeavor and is little short of profound.

With the values given in the schematic, design, write and Run a program which will try out values of load resistance ranging from 1 to 20 ohms, in 1 ohm increments, and PRINT the answers to the following:

1. Value of Load Resistance (from 1 to 20 ohms)

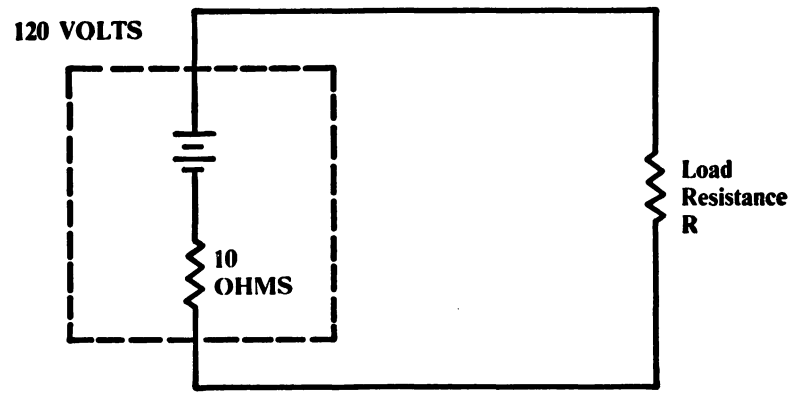
2. Total circuit power (circuit current squared, times circuit resistance) = $I^2 * (10 + R)$

3. Power lost in source (circuit current squared, times source resistance) = $I^2 * 10$

4. Power delivered to load (circuit current squared, times load resistance) = $I^2 * R$

Note: Circuit current is found by dividing source voltage (120 volts) by total circuit resistance (load resistance + 10 ohms source resistance). Everything follows Ohms Law (V=I*R) and Watts Law (P=I*V).

GOOD LUCK! Don't look at the answer until you've got it whipped.



Learned In Chapter 13

Commands	Miscellaneous	Menu
LIST ###	Timer Loop	Run
RUN ###	"Brute Force" method	Suspend (⌘ S)
DELETE ###		Stop (⌘ .)
CONT		Continue

Chapter 14

Formatting With TAB

After those last few Chapters, it's time for an easy one.

We already know 3 ways to set up our output PRINT format.

We can:

1. Enclose what we want to say in quotes, inserting blank spaces as necessary.

2. Separate the objects of the PRINT statement with semi-colons so as to PRINT them tightly together on the same line.

3. Separate the objects of the PRINT statement with commas to PRINT them on the same line in the different PRINT "zones."

Macintosh will default to 5 PRINT zones unless reset using the WIDTH statement.

A 4th way is by using the TAB function, which is similar to the TAB on a regular typewriter. TAB is especially useful when the output consists of columns of numbers with headings. Type in the following NEW program and Run:

```
10 PRINT TAB(5);"THE";TAB(20);"TOTAL";
   TAB(35);"SPENT"
20 PRINT TAB(5);"BUDGET";TAB(20);"YEAR'S";
   TAB(35);"THIS"
30 PRINT TAB(5);"CATEGORY";TAB(20);"BUDGET";
   TAB(35);"MONTH"
```

The Run should appear:

```
THE                TOTAL              SPENT
BUDGET             YEAR'S             THIS
CATEGORY           BUDGET             MONTH
```

EXERCISE 14-1: EDIT the above program using the 3 ways we know (so far) to format PRINTing. Here is a start:

```
10 PRINT"THE          TOTAL              SPENT"
20 PRINT"BUDGET","YEAR'S","THIS"
30 PRINT TAB( );"CATEGORY";TAB( );
   "BUDGET";TAB( );"MONTH"
```

Use ordinary spacing for the first Line of the heading, zone spacing for the second Line and TABbing for the third Line.

A semi-colon is traditionally used following TAB, as shown above. Most newer BASIC interpreters permit a blank, quote marks or even no symbol, instead.

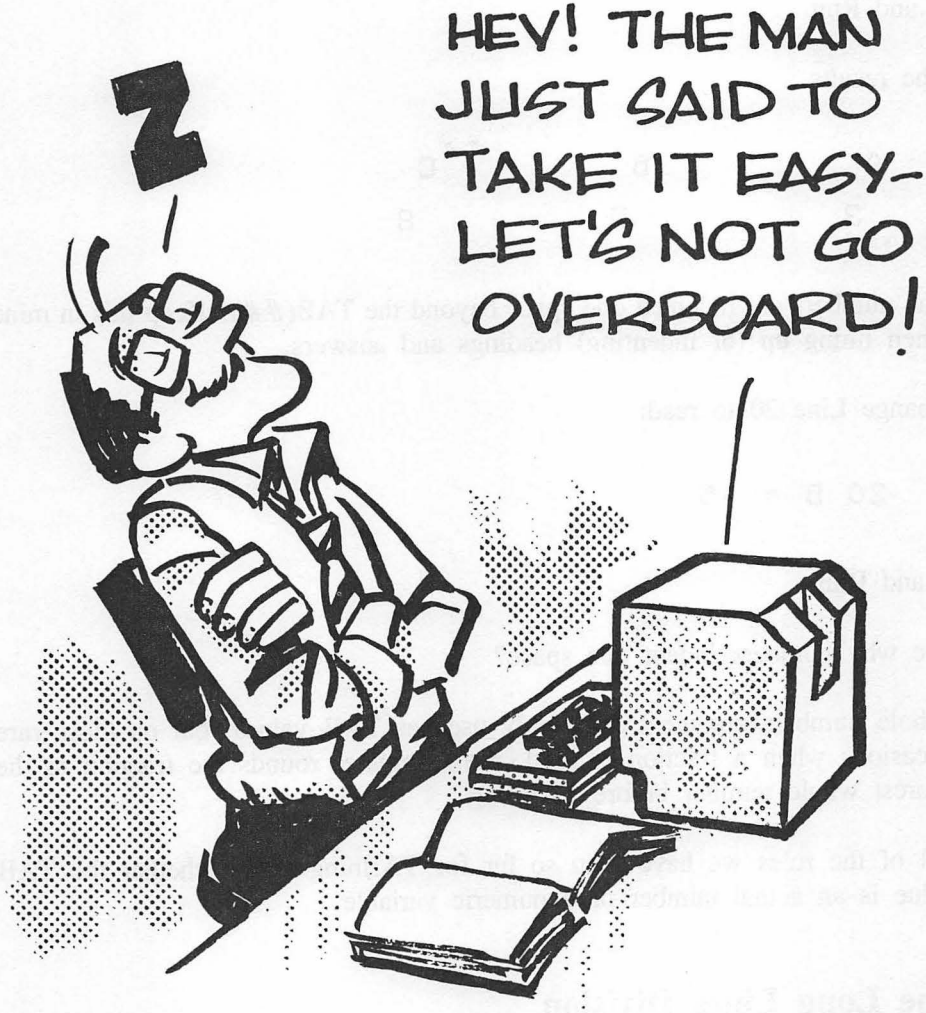
```
10 PRINT TAB(10) "OOPS, NO SEMICOLON!"
```

Runs just fine, but leave out semi-colons at your own peril.

The Computer will start PRINTing TAB(##) spaces to the right of the left margin. It is important to remember when using TABs that whenever numbers or numeric variables are PRINTed, the Computer inserts one additional space to the left of the number to allow for the − or + sign.

Type this NEW program:

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


Z
HEY! THE MAN JUST SAID TO TAKE IT EASY- LET'S NOT GO OVERBOARD!

```
40 PRINT TAB(10);"A";TAB(20);"B";TAB(30);"C"
50 PRINT TAB(10);A;TAB(20);B;TAB(30);C
```

...and Run.

The results...

```
A             B             C
 3             5             8
```

The numbers are indented one space beyond the TAB(##). Keep this in mind when lining up (or indenting) headings and answers.

Change Line 20 to read:

```
20 B = -5
```

...and Run.

See why numbers indent one space?

Whole numbers are most commonly used as TAB values, but on those rare occasions when a fraction is used, the Computer rounds the fraction to the nearest whole number before TABbing.

All of the rules we have seen so far for TABbing apply whether the TAB value is an actual number or a numeric variable.

The Long Lines Division

Have you ever wondered what would happen if we had to PRINT a great number of headings or answers on the same line -- but didn't have enough room on the program Line to neatly hold all the TAB statements? You have? Really? You're in luck because it's easy. Type and Run the following New program. It stretches the "leaving out of semi-colons" to the limits of prudence.

```
10 A = 0
```

```
20 B = 1
30 C = 2
40 D = 3
50 E = 4
60 F = 5
70 G = 6
80 PRINT "A"TAB(10)"B"TAB(20)"C"TAB(30)"D";
90 PRINT TAB(40)"E"TAB(50)"F"TAB(60)"G"
100 PRINT A;TAB(10)B;TAB(20)C;TAB(30)D;
110 PRINT TAB(40)E;TAB(50)F;TAB(60)G
```

The trailing semi-colons (;) in Lines 80 and 100 do the trick. They make the end of one PRINT Line continue right on to the next PRINT Line without activating a carriage return. The combination of TAB and trailing semi-colon allows us almost infinite flexibility in formatting the output.

Finally, to see the program crash when one too many liberties are taken with semicolons, remove the last one in Line 110 and Run.

The program Runs fine until the Computer encounters the second TAB instruction in Line 110. The Computer stops and displays the error message:

```
Subscript out of range
```

Click the **OK** box, and insert a semicolon before the last TAB in Line 110. Run again to make sure that fixed the problem.

POS(N)

An additional and sometimes useful statement allows the Computer to report back the horizontal POSition of the cursor. This simple New program exercises the POS function.

```
5 WIDTH 60
```

```
10 INPUT "ANY NUMBER BETWEEN -9 AND 45";A
20 PRINT TAB(10 + A)
30 PRINT POS(0);
40 PRINT " IS NUMBER OF NEXT PRINT COLUMN"
```

...and RUN.

Line 5 sets the screen WIDTH to 60 characters.

Line 20 just TABs the cursor over 10 places from A.

Line 30, containing POS, is the key. The 0 inside the brackets is just a "dummy." Most any other number or variable would work as well -- but something has to be placed there. POS reports back the horizontal cursor POSition on the screen.

Remember, most Macintosh fonts are proportionally spaced. Characters may not always line up properly unless the monospace "Monaco" font is specified.

That's enough fooling around with Mother Nature.

EXERCISE 14-2: Rework the answer to Exercise 13-3 to include the *hourly* rate of pay in the PRINTout. Use the TAB Function to have the chart display all 5 columns side by side.

Learned In Chapter 14

Statements	Print Modifiers	Miscellaneous
POS	TAB	Trailing semi-colon

Chapter 15

Grandson Of FOR-NEXT

The FOR-NEXT loop didn't go away for long. It returns here more powerful than ever. Type this New program:

```
10 FOR A = 1 TO 3
20  PRINT "A LOOP"
30   FOR B = 1 TO 2
40    PRINT ,"B LOOP"
50   NEXT B
60 NEXT A
```

...and Run.

For good program readability, add 2 blank spaces in Line 20 before PRINT, 3 in Line 30 before FOR, 4 in 40 before PRINT, and 3 in 50 before NEXT.

The result is:

```
A LOOP
                    B LOOP
                    B LOOP
A LOOP
                    B LOOP
                    B LOOP
A LOOP
                    B LOOP
                    B LOOP
```

This display vividly demonstrates operation of the nested FOR-NEXT loop. "Nesting" is used in the same sense that drinking glasses are "nested" when stored to save space. Certain types of portable chairs, empty cardboard boxes, etc. can be nested. They fit one inside the other for easy stacking.

Let's analyze the program a Line at a time:

Line 10 establishes the first FOR-NEXT loop, called A, and directs that it be executed 3 times.

Line 20 PRINTs `A LOOP` so we will know where it came from in the program. See how this program Line is indented to make it stand out as being nested in the "A loop"?

Line 30 establishes the second loop, called B, and directs that it be executed twice. It is indented even more so we can instantly see that it is buried even deeper in the "A" loop.

Line 40 PRINTs two items: "nothing" in the 1st PRINT zone, then the comma kicks us into the 2nd PRINT zone where `B LOOP` is PRINTed. Makes for a clear distinction on the screen between A loop and B loop, eh?

Line 50 completes the "B" loop and returns control to Line 30 for as many executions of the "B" loop as Line 30 directs. (So far we have PRINTed one "A" and one "B.")

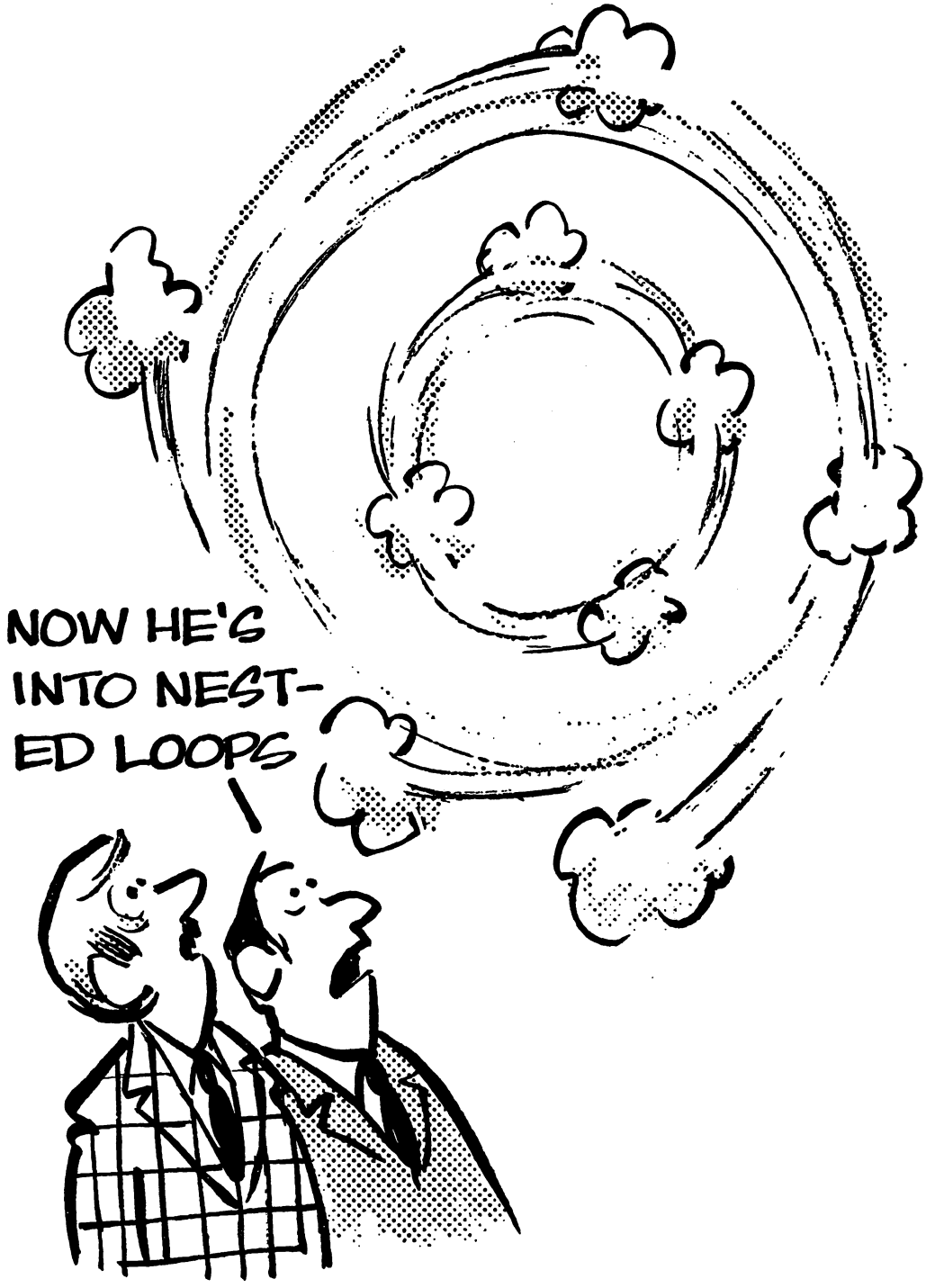
Line 60 ends the first pass through the "A" loop and sends control back to Line 10, the beginning of the A loop. The A loop has to be executed 3 times before the program RUN is complete, PRINTing "A" 3 times and "B" six times (3 times 2).

Study the program and the explanation until you completely comprehend. It's simple but powerful magic.

Okay, to get a better "feel" for this nested loop (or loop within a loop) business, let's play with the program. Change Line 10 to read:

```
10 FOR A = 1 TO 5
```

...and Run.


NOW HE'S INTO NEST-ED LOOPS

Right! A was PRINTed 5 times, meaning the "A" loop was executed 5 times, and B was PRINTed 10 times -- twice for each pass of the "A" loop. Now change Line 30 to read:

```
30 FOR B = 1 TO 4
```

...and Run.

Nothing to it! A was PRINTed 5 times, and B PRINTed 20 times. Do you remember what to do if the A's and B's whiz by too fast? Press the ⌘S or choose Suspend from the Control menu to temporarily freeze the display. Press most any other key to continue.

How To Goof-Up Nested FOR-NEXT Loops

The most common error beginning programmers make with nested loops is improper nesting. Change these Lines:

```
50  NEXT A
60 NEXT B
```

...and Run.

The Computer displays a dialog box saying:

```
NEXT without FOR
```

and blocks Line 10.

Looking at the program, we quickly see that the B loop is *not* nested within the A loop. The FOR part of the B loop is inside the A loop, but the NEXT part is outside it. That doesn't work! A later chapter deals with something called "flow charting," a means of helping us plan programs to avoid this type of problem. Meanwhile, we just have to be careful.

Breaking Out Of Loops

Improper nesting is illegal, but breaking out of a loop when a desired con-

dition has been met is OK. Click the **OK** box, then add and change these Lines:

```
50    NEXT B
55   IF A = 2 GOTO 100
60 NEXT A
99 END
100 PRINT "A EQUALED 2. RUN ENDED."
```

...and Run.

As the screen shows, we "bailed out" of the A loop when A equaled 2 and hit the Test Line at 55. The END in Line 99 is just a precautionary block set up to STOP the Computer from executing into Line 100 unless specifically directed to go there. That would never happen in this simple program, but we will use *protective ENDs* from time to time to remind us that Lines which should be reached only by specific GOTO or IF-THEN statements must be protected against accidental "hits."

We'll be seeing a lot of the *nested* FOR-NEXT loop now that we know what it is and can put it to use.

EXERCISE 15-1: Re-enter the original program found at the beginning of this Chapter. It contains a B loop nested within the A loop. Make the necessary additions to this program so a new loop called "C" will be nested within the B loop and will PRINT "C LOOP" 4 times for each pass of the B loop.

EXERCISE 15-2: Use the program which is the answer to Exercise 15-1. Make the necessary additions to this program so a new loop called "D" will be nested within the C loop and will PRINT "D LOOP" 5 times for each pass of the C loop.

WHILE - WEND

A more obscure variation on the FOR-NEXT idea is the WHILE-WEND statement. WHILE is the beginning statement in a series which is executed repeatedly until a certain WHILE condition becomes *false*.

The loop which begins with WHILE must be closed by a WEND. Type in this NEW program:

> When writing programs, be sure to indent Lines to highlight nesting or program flow. It helps when reading them -- and is a great aid when debugging (troubleshooting) problems. End of message.

```
10 X = 1
20 WHILE X<>0
30    INPUT X
40    S = S + X
50 WEND
60 PRINT "SUM =";S
```

...and Run.

INPUT several non-zero numbers, then INPUT a 0. As long as X does not = 0, WEND keeps returning execution to WHILE. When X is INPUT as 0, the WHILE statement in Line 20 interprets the 0 as its "bail-out" cue and exits the loop via WEND. Line 60 PRINTs the sum of the numbers INPUT.

And with that, let's WEND our way towards the next Chapter.

Learned In Chapter 15

Statements	Miscellaneous
WHILE-WEND	Nested FOR-NEXT loops
	Protective END blocks

Chapter 16

The INTeger Function

Integer? "I can't even pronounce it, let alone understand it." Oh, come, come. Don't let old nightmares of being trapped in Algebra class stop you *now*. It's pronounced (IN-teh-jur) and simply means a *whole* number like -5, 0, or 3, etc. How difficult can that be? Come to think of it, some folks make a whole career of complicating simple ideas. We try to do just the opposite.

The INTeger function, INT(X), allows us to "round off" any number, large or small, positive or negative, into an INTeger, or *whole* number.

Careful -- we're not talking about ordinary rounding. Ordinary rounding gives us the *closest* whole number, whether it's larger or smaller than X. INT(X), on the other hand, gives us the ***largest whole number which is less than or equal to*** *X*. As you'll see in this Chapter, this is a very versatile form of rounding -- in fact, we can use it to produce the other "ordinary" kind of rounding.

Select New from the File menu to clear out any old programs, then type:

```
10 X = 3.14159
20 Y = INT(X)
50 PRINT "Y =";Y
```

...and Run.

The display reads:

```
Y =  3
```

Oh -- success is so sweet! It rounded 3.14159 off to the whole number 3. Change Line 10 to read:

```
10 X = -3.14159
```

...and Run.

Good Grief! It rounded the answer *down* to read:

```
Y = -4
```

What kind of rounding is this? Easy. The INT function *always* rounds *down* to the next *lowest whole number*. Pretty hard to get that confused! It makes a positive number less positive and makes a negative number more negative (same thing as less positive). At least it's consistent.

Taking it a Line at a time:

Line 10 set the value of X (or any of our other alphabet-soup variables) equal to the value we specified, in this case pi.

Line 20 found the INTeger value of X and assigned it to a variable name. We chose Y.

Line 50 PRINTed an identification label (Y =) followed by the value of Y.

Not Content To Leave Well Enough Alone

We can do some foxy things by combining a FOR-NEXT loop with the INTeger function.

Change the program to read:

```
10 X = 3.14159
20 Y = INT(X)
30 Z = X - Y
40 PRINT "X =";X
```

```
50 PRINT "Y =";Y
60 PRINT "Z =";Z
```

Save As INTEGER1 ... and Run.

AHA! I don't know what we've discovered, but it must be good for something. It reads:

```
X = 3.14159
Y = 3
Z = .1415901
```

We've split the value of X into its INTeger (whole number) value (calling it Y) and its decimal part (calling it Z).

Lines 40, 50, and 60 merely PRINTed the results.

Hold The Phone

Oh - oh! Why doesn't Z equal the exact difference between X and Y? Where did that "01" in the decimal value come from? What gives?

The slight difference has nothing to do with the INT function. You have discovered the Computer's limit of accuracy. Just like a calculator (or a person), a computer can never be perfectly accurate all the time. For short arithmetic expressions, the Mac is accurate to six digits. In longer, more complex expressions, such a minute error in the sixth digit can be magnified to where it becoms significant. All programmers have to cope with this kind of built-in error.

There *is* a way to control the accuracy of our results. It involves artificially rounding the fraction to the desired number of decimal places and then forcing the Computer to PRINT out only those digits which are "properly rounded."

For example, suppose we need pi accurate to only 3 decimal places. (Of course, we can specify it as 3.142, but that's not the point.) Select New, then enter and Run the following program:

```
10 X = 3.14159
```

```
20 X = X + .0005
30 X = INT(X * 1000)/1000
40 PRINT X
```

Adding .0005 in Line 20 gives our fraction a "push in the right direction." If this fraction has a digit greater than 4 in its 10-thousandths-place, then adding .0005 will effectively increase the thousandths-place digit by 1. Otherwise, the added .0005 will have no effect on the final result. This results in what's called "4/5 rounding."

Try using other values than pi for X (just make sure X*1000 isn't too large for the INT function to handle).

It's easy to change the program to round accurately to a number of decimal places. For example, to round X off at the hundredths-place (2 digits to the right of the decimal point), change Lines 20 and 30 to read:

```
20 X = X + .005
30 X = INT(X * 100)/100
```

...and Run, using several values for X.

This trick is very useful when PRINTing out dollars-and-cents. It prevents $39.995 type prices.

HMMMM!!!

Do you suppose there is any way to separate each of the digits in 3.14159, or in any other number? Do you suppose we would have brought it up if there wasn't? After all (mumble, mumble).

It's really your turn to do some creative thinking, but we'll get it started and see if you can finish this idea. First, wipe out the resident program and reOpen `INTEGER1`.

Now, if we multiply Z by 10, then Z will become a whole number plus a decimal part: 1.4159. We can then take *its* INTeger value and strip off the decimal part, leaving the left hand digit standing alone. Let's label the Left-hand digit L and see what happens. Enter:

```
70 Z = Z * 10
80 L = INT(Z)
90 PRINT "L =";L
```

...and Run.

Hmmm! It reads:

```
X = 3.14159
Y = 3
Z = .1415901
L = 1
```

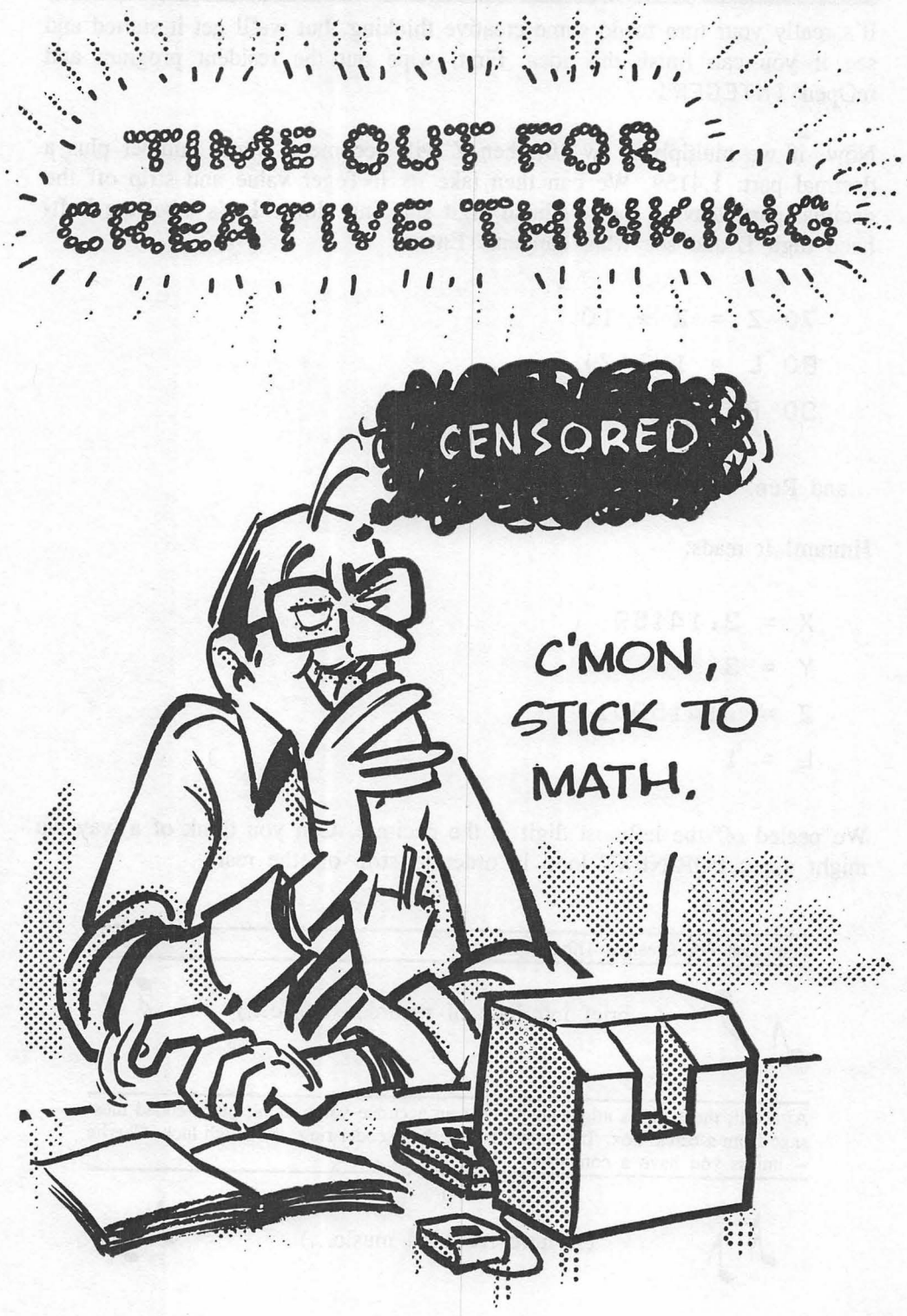
We peeled off the leftmost digit in the decimal. Can you think of a way we might use a FOR-NEXT loop in order to strip off the rest?

Time Out For Creative Thinking!

(...brief interlude of recorded music...)

After all, these digits might not be just an accurate value of pi, but a coded message from a cereal box. If you don't have the decoder ring, it's tough luck, Charlie -- unless you have a computer!

(...more recorded music...)


TIME OUT FOR
CREATIVE THINKING
CENSORED
C'MON,
STICK TO
MATH.

Enough thinking there on company time! Add these Lines:

```
75 FOR A = 1 TO 5
100 Z = Z - L
110 Z = Z * 10
120 NEXT A
```

Save As INTEGER2 and Run.

VOILA! The "PRINTout" reads:

```
X = 3.14159
Y = 3
Z = .1415901
L = 1
L = 4
L = 1
L = 5
L = 9
```

Let's analyze the program.

Line 75 began a FOR-NEXT loop with 5 passes, one for each of the 5 digits right of the decimal.

Line 100 creates a new decimal value of Z by stripping off the INTeger part. (Plugging in the values, Z = 1.4159 − 1 = .415901.)

Line 110 does the same as Line 70 did, multiplying the new decimal value times 10 so as to make the left-hand digit an INTeger and vulnerable to being snatched away by the INT function. (Z = .415901 * 10 = 4.15901.)

Line 120 sends control back to Line 75 for another pass through the clipping program, and the rest is history.

Is This Too Hard To Follow?

No -- it isn't hard to follow, and we could go through and calculate every intermediate value just like I did before, and it would be perfectly clear (to coin a phrase). Let's instead learn a way to let the Computer help us understand what it is doing.

We can insert temporary PRINT Lines anywhere in any program to follow every step in its execution. The Computer can actually overwhelm us with data. By carefully indicating exactly what we want to know, it will display the inner details of any process. Start by adding this Line:

```
72 PRINT "#72 Z =";Z
```

...and Run.

The essentials of this "test" or "debugging" or "flag" Line are:

1. It PRINTs something.
2. The PRINT tells the *Line number* for analysis and easy location for later erasure.
3. It tells the *name* of the variable we are watching at that point in the program.
4. It gives the *value* of that variable at *that point.*

This "flagging" is such a wonderful tool for troubleshooting stubborn programs that you will want to make a habit of never forgetting to use it when the going gets tough.

It can be very helpful when inserted in FOR-NEXT loops -- so:

```
77 PRINT "#77 A =";A
```

...and Run.

Wow! The information comes thick and fast! It tells what is happening during

each pass of the loop. Hard to keep track of so much, and we've barely begun. Is there some way to make it more readable?

Yes, there are lots of ways. Indenting is one simple way to separate the answers from the troubleshooting data. Change Lines 72 and 77 as follows:

```
72 PRINT ,"#72 Z =";Z
77 PRINT ,,"#77 A =";A
```

...and Run.

Ahh. How sweet it is. That is so easy to read, let's monitor one more point in the program. Type in:

```
105 PRINT ,,,"#105 Z =";Z
```

Save As INTEGER3 ... and Run.

Very nice.

Well, there it is. All the data we can handle (and then some). By using Suspend or the ⌘S keys to temporarily halt execution, we can study the data at every step to understand how the program works (or doesn't). Do it. Understand this program and all its little lessons completely. When you are satisfied, go back and erase the "flags."

INTeger Division

And if that isn't quite enough to keep the mind reeling, there *is* another way to get the INTeger value of the result of an equation without using the INT function! It is called "INTeger division," and instead of using the normal slash /, we use a backslash \.

Choose New. Then enter this example:

```
10 X = 23.987
20 Y = 2.567
30 PRINT "X/Y =";X/Y
```

```
40 PRINT "INT(X/Y) =";INT(X/Y)
50 PRINT "X\Y =";X\Y
```

...and RUN. It should produce:

```
X/Y = 9.344371
INT(X/Y) = 9
X\Y = 8
```

8? Is that right? Yep. INTeger division actually modifies the value of each variable in the equation *before the calculation is made*. In this case, both X and Y are rounded to the nearest whole numbers, 24 and 3, then division is performed producing the INTeger value of 8. Hmmm, did that sink in?

Take a breather. You have learned quite enough in this Chapter.

EXERCISE 16-1: Enter this straightforward New program for finding the area of a circle.

```
10 P = 3.14159
20 PRINT "RADIUS", "AREA"
30 PRINT
40 FOR R=1 TO 10
50  A = P * R * R
60  PRINT R,A
70 NEXT R
```

...and Run.

Area equals pi times the radius squared (that is, the radius times itself).

Pretty routine stuff -- huh? Problem is, who needs all those little numbers to the far right of the decimal point. *Oh, you do?* Well, there's one in every crowd. The rest of us can do without them. Modify the resident program to suppress all the numbers to the right of the decimal point.

EXERCISE 16-2: Now, knowing just enough to be dangerous, and in need of a lot of humility, change Line 55 so that each value of *area* is rounded (down) to be accurate to one decimal place. For example:

RADIUS	AREA
1	3.1

EXERCISE 16-3: Carrying the above Exercise one step further, modify the program Line 55 to round (down) the value of area to be accurate to 2 decimal places.

Learned In Chapter 16

Functions	Math Operators	Miscellaneous
INT(X)	\	Flags INTeger Division

Chapter 17

More Branching Statements

It Went That-A-Way

Enter this New program:

```
10 INPUT "TYPE A NUMBER BETWEEN 1 AND 5";N
20 IF N = 1 GOTO 100
30 IF N = 2 GOTO 120
40 IF N = 3 GOTO 140
50 IF N = 4 GOTO 160
60 IF N = 5 GOTO 180
70 PRINT "THE NUMBER YOU TYPED WAS"
80 PRINT "NOT BETWEEN 1 AND 5!"
90 END
100 PRINT "N = 1"
110 END
120 PRINT "N = 2"
130 END
140 PRINT "N = 3"
150 END
160 PRINT "N = 4"
170 END
180 PRINT "N = 5"
```

Save As ONGOTO1 and Run it a few times to feel comfortable and to be sure it is "debugged." Be sure to try numbers outside the range of 1-5, including 0 and a negative number.

Debugged is an old Latin word which, freely translated, means "getting all the errors out of a computer program."

This program works fine for examining the value of a variable, N, and sending the Computer off to a certain Line number to do what it says there. If there are lots of possible directions in which to branch, however, we will want to use a greatly improved test called ON-GOTO which cuts out lots of Lines of programming.

DELETE Lines 20, 30, 40, 50 and 60. Remember how? (Type DELETE 20-60 in the Command window.)

Enter this new Line:

```
20 ON N GOTO 100,120,140,160,180
```

Save As ONGOTO2 and Run a few times, as before.

Works fine until a negative number or a number greater than 255 is entered. Then the Mac responds with an "Illegal function call" error and blocks the contents of Line 20.

Using the ON-GOTO statement is really pretty simple, though it looks hard. Line 20 says:

IF the "rounded" value of N is 1, THEN GOTO Line 100.

IF the "rounded" value of N is 2, THEN GOTO Line 120.

IF the "rounded" value of N is 3, THEN GOTO Line 140.

IF the "rounded" value of N is 4, THEN GOTO Line 160.

IF the "rounded" value of N is 5, THEN GOTO Line 180.


AW, LET'S NOT DRAG OUT THAT OL' CHESTNUT FOR "DE-BUGGING"!

IF the "rounded" value of N is not one of the numbers Listed above, THEN move on to the next Line, Line 70.

The ON-GOTO statement has a built-in standard rounding system. If the number INPUT is less than halfway between 2 INTegers, rounding is downward to the *lower* INTeger. If it is halfway or larger, rounding is to the next *higher* INTeger.

Run again, and type in the following values of N to prove the point:

```
2.4
1.5
3.7
4.5001
4.5
0.6
```

Get the picture?

Variations On A Theme

Lots of tricks can be played to milk the most from ON-GOTO. For example, if we wanted to branch out to 15 different locations but didn't want to type that many different numbers on a single ON-GOTO Line, we could use several Lines, like this (don't bother to do it):

```
20 ON N GOTO 100,120,140,160,180
30 ON N-5 GOTO 200,220,240,260,280
40 ON N-10 GOTO 300,320,340,360,380
```

and, of course, fill in the proper responses at those Line numbers.

In Line 30, it was necessary to subtract 5 from the number being INPUT as N, since each new ON-GOTO Line starts counting again from the number 1.

In Line 40, since we had already provided for INPUTs between 1 and 10, we subtract 10 from N to cover the range from 11 through 15.

We could have used any letter after "ON," not just N. N can be the value of a letter variable or a complete expression, either calculated in place or calculated in a previous Line.

Give Me A SGN(X)

Using ON-GOTO along with a new function called SGN (it's pronounced "sign") plus a modest amount of imagination produces a useful little routine. But first, let's learn about SGN.

The SGN function examines any number to see whether it is negative, zero, or positive. It tells us the number is negative by giving us a (−1). (In computer language, "it returns a −1.") If the number is zero, it returns a (0). If positive, it returns a (+1). SGN is a very simple function.

In order to sneak easily into the next concept, we will simulate the built-in SGN function with a SUBROUTINE.

So What Is A Subroutine?

Funny you should ask. A subroutine is a short but very specialized program (or routine) which is built into a large program to meet a specialized need. The BASIC interpreter incorporates many of them which we never see.

As an example of how to create functions that are *not* included in our BASIC, we will use a 5-Line subroutine instead of the "SGN" function to accomplish the same thing. (Even though Microsoft BASIC has its own "SGN" function, you should complete this Chapter to be sure you learn about subroutines. We don't want to turn out computer illiterates, you know.)

Until now we have assigned a number to each program Line to help identify them for later study. Let's try typing in this program without Line numbers. "Scratch" the program now in memory by choosing New, then -- very carefully, type in this SGN subroutine:

```
END
SIGN:
  REM * SGN(X) * INPUT X, OUTPUT T=-1,0, OR +1
  IF X < 0 THEN T = -1
  IF X = 0 THEN T = 0
```

```
IF X > 0 THEN T = +1
RETURN
```

SIGN: is the label assigned to the subroutine, and it must be followed by a colon (:). We indented the program's Lines to help them stand out from the other Lines. Remember, indenting isn't mandatory, just a way to make programs easier to read.

We can assign any name we want to the routine as long as we do not use a name that is reserved for use as a BASIC statement, command or function. The label can contain any combination of letters and numbers, although it must begin with a letter and cannot be more than 40 characters long.

A list of reserved words can be found in Appendix B.

"CALLING" A Subroutine

(Sort of like calling hogs...)

GOSUB directs the Computer to go to a Line number or a subprogram label, execute what it says there and in the Lines following, and when done, RETURN back to the Line containing that GOSUB statement. The RETURN statement is always at the end of a subroutine.

RETURN is to GOSUB what NEXT is to FOR.

One advantage to writing subroutines (or subprograms) without Line numbers is that the subroutine can be placed anywhere within the main program without interfering with the existing Line numbering sequence. Notice that we placed a protective END block in the first Line before our subroutine so the Computer doesn't come crashing into it. Of course, this won't be necessary if the routine happened to be placed ahead of the main program.

Getting Down To Business

Okay, now let's combine GOSUB with the SGN subroutine to see what all this fuss is about. Add:

```
10 INPUT "TYPE ANY NUMBER";X
```

```
20 GOSUB SIGN
30 ON T+2 GOTO 50,70,90
40 END
50 PRINT "THE NUMBER IS NEGATIVE."
60 END
70 PRINT "THE NUMBER IS ZERO."
80 END
90 PRINT "THE NUMBER IS POSITIVE."
```

...and Run.

Try entering negative, zero and positive numbers to be sure it works. Most of the program workings are obvious, but here is an analysis:

Line 10 INPUTs any number.

Line 20 sends the Computer to the subroutine labeled SIGN via a GOSUB statement. This is different from an ordinary GOTO, since a GOSUB will return control to the originating Line like a boomerang when the Computer hits a RETURN. The call to GOSUB is not complete and will not move on to the next program Line until a RETURN is found.

Three Lines in the subroutine contain the simple logic routine.

The last Line in the subroutine holds RETURN, which sends control back to Line 20, which silently acknowledges the return and allows execution to move to the next Line.

Line 30 is an ordinary ON-GOTO statement, but adds 2 to the value of its variable, in this case T. Line 30 really says,

> "If T is -1, THEN GOTO Line 50. If it is zero, THEN GOTO Line 70, and if it is +1, GOTO Line 90."

By adding 2 to each of the values from SGN, we "matched" them up with the 1, 2, and 3 series which is built into the ON-GOTO statement.

Lines 40, 60, and 80 are routine protective END blocks.

By the way, many subroutines are not this simple -- as a matter of fact, they often contain very hairy mathematical derivations. We won't bother trying to explain any of them -- if you're heavily into Math, you go right ahead and play with the numbers.

ON-GOSUB

ON-GOSUB is a variation on the ON-GOTO and GOSUB schemes. It allows branching to a variety of *subroutines* from a single GOSUB statement. If we had 3 subroutines and had to choose which one to use based on the value of X, here is how the program might be structured. (Don't bother to type it in.)

```
10 INPUT X
20 ON X GOSUB 1000,2000,3000
30  REM - CALCULATIONS HERE
60  REM - PRINT RESULTS HERE
99 END
1000 REM - 1ST ROUTINE GOES HERE.
1099  RETURN
2000 REM - 2ND ROUTINE GOES HERE.
2099  RETURN
3000 REM - 3RD ROUTINE GOES HERE.
3099  RETURN
```

Or with labels:

```
10 INPUT X
20 ON X GOSUB FIRST,SECOND,THIRD
30  REM - CALCULATIONS HERE
40  REM - PRINT RESULTS HERE
99 END
```

```
FIRST: REM - 1ST ROUTINE GOES HERE.
  RETURN
SECOND: REM - 2ND ROUTINE GOES HERE.
  RETURN
THIRD: REM - 3RD ROUTINE GOES HERE.
  RETURN
```

Preview Of Coming Attractions?

Like so much of what we are learning, this is just the tip of the iceberg. The ON-GOTO and ON-GOSUB statements have many more clever applications, and they will evolve as we need them. As a hint for restless minds, note that the *value* of X (which we INPUT) was not used, but it didn't go away. All we did was find its SGN. Hmmm...

Routines Vs. Subroutines

In this Chapter we studied a special-purpose routine used as a SUBroutine. It was easy to understand. All routines, understandable or not, can be built directly into any program instead of being set aside and "called" as subroutines. The main value of subroutines is that they can be "called" repeatedly from different parts of a program, which is often desirable. Ordinary routines are usually only used once, so use of GOSUB and RETURN with them often doesn't make good programming sense.

One value of using routines as subroutines is that some are exceedingly complex to type without error, and if each is typed once and SAVEd on disk, it can be quickly and accurately LOADed back into the Computer as the first step in creating a new program, or added to an existing one.

We'll have more to say in a later Chapter. When you see just how powerful subroutines are, you'll feel like your Macintosh is even smarter than it thinks it is.

Now, it's your turn.

EXERCISE 17-1: Remove all traces of the subroutine from the resident program. Use the SGN function to accomplish the same thing we have been doing with a subroutine. Hint: T = SGN(X)

Learned In Chapter 17

Functions	Statements	Miscellaneous
SGN(X)	ON-GOTO	Debugging
	GOSUB	Calling a subroutine
	ON-GOSUB	Routines
	RETURN	Labels

Chapter 18

Random Numbers

At RANDOM

A *random* number is one with a value which is unpredictable. A "Random Number Generator" is a device which pulls *random numbers* "out of a hat." Our Computer has an RND generator, and it works this way:

```
N = RND(X)
```

where N is the random *number.*

RND is the symbol for *RaNDom* Function.

X is a dummy value, either negative, zero, or positive, which can be either placed between the parentheses or brought in as a variable from elsewhere in the program.

Type this NEW program:

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

...and Run. Did you observe:

1. A different number appeared each time?
2. All numbers were between 0 and 1?
3. *Very* small numbers were expressed in Exponential notation?

RND behaves exactly the same as RND(X), when X is a positive number. Since this is almost always how it is used, we almost always omit (X). Put a semi-colon behind the PRINT statement and increase the FOR-NEXT loop

to 40 passes to put more numbers on the screen at one time. Line 10 is added to keep the printout from running off the display.

```
10 WIDTH 60
20 FOR N = 1 TO 40
30  PRINT RND;
40 NEXT N
```

Close the List window to get it out of the way and Run.

The Computer uses an internal "seed number" to produce a "random number" series. The seed for RND is always the same.

You get the idea.

Now bring back the List window (⌘L), and add:

```
50 PRINT RND(0)
```

Close the List window again and Run.

The *last* RaNDom number PRINTed is repeated. Hmmm...

This Is Fairly Exciting!

Well, maybe so, but you ain't seen nothin' yet! Virtually all computer games are based on RND(X), and we'll soon play and design our own.

RND With Racing Stripes

In most real-life cases we need a Random INTeger, not a Random Number between 0-1. To create numbers larger than 1, we have to resort to mathematical chicanery.

Remove Line 50, and change Line 30 to read:

```
30 PRINT INT(RND * 15 + 1);
```

...and Run.

Wow! That's more like it -- real live random INTegers. They all have values between 1 and 15. Figured out the scheme? Pretty simple, isn't it?

This equation specifies the *range* of INTegers RND will output:

```
R = INT(RND * (B-A+1) + A)
```

where R = the RaNDom number,
B = the *largest* INTeger and
A = the *smallest* INTeger.

Pseudo-Random

Random numbers are unpredictable, properly functioning computers are not. So how do we get truly random numbers from the Computer? We usually don't; we get *pseudo-random* numbers.

Run the program several times, and study the screen. The numbers from each Run are the same as from the previous Run! They may be random, but are certainly predictable!

Change Line 30, and Run several times using negative seed numbers, like:

```
30 PRINT RND(-20);
```

We get a different set of numbers with each seed -- but all the numbers in any one set have the same value. Running again, the numbers are unchanged. Using a different negative seed with RND produces a similar result, but the value will be changed.

When Running game programs using RND, it's a good idea to set the *seed* to an unpredictable value. To ensure that a different pseudo-random number sequence is used each time the Computer uses RND(X), we need to find a source of unpredictable numbers somewhere in the Computer.

Enter the Command window, and type the following:

```
CLS          Return
PRINT TIME$          Return
```

Hmmm, that's interesting. If we could somehow separate the seconds from the rest of the time, we would have essentially unpredictable numbers between 0 and 59. That would give us 60 different seed numbers. Here's how to do it:

```
PRINT VAL(RIGHT$(TIME$,2))
```

The mechanics of that statement will be covered in detail in a later Chapter, but for those too curious to wait here is a short analysis: RIGHT$(TIME$,2) means, "Peel off the 2 right-most characters from TIME$." VAL means, "Make sure those 2 characters are numbers so we can use them in a numeric variable."

We now have the tools to write a subroutine for "randomizing" the INPUT to RND. Type the following:

```
10 GOSUB 10000          (to our own Randomizer)
20 FOR N = 1 TO 10
30  PRINT RND;
40 NEXT N
99 END
10000 S = VAL(RIGHT$(TIME$,2))
10010 FOR N = 1 TO S
10020  D = RND
10030 NEXT N
10040 RETURN
```

and here's how it works:

Line 10000 picks off a number between 0-59.

Lines 10010-10030 "burn off" the first "S" numbers in the RND series.

Line 10040 RETURNs execution to the main program where:

Line 30 continues RND and PRINTs the next 10 numbers.

If you don't believe any of this, insert a temporary Line:

```
10005 S = 25
```

which sets the number of burn-offs to a specific value. Then Run several times. The same 10 numbers appear each time, so it must be working.

Remove Line 10005, and Run a few more times. Ahhh! Now we've got it. Instead of only one, we now have 60 versions. We have developed a viable RANDOMIZER routine.

Randomizer

With a RANDOMIZE statement at the beginning of the program, the Computer will "shuffle," or "reseed," the series of random numbers. Type this New program:

```
10 RANDOMIZE
20 WIDTH 60
30 FOR N = 1 TO 10
40  PRINT RND;
50 NEXT N
```

...and Run.

Oh, Oh! More decisions needed. RANDOMIZE allows the selection of 65536 different seed values. Even so, whoever picks the seed controls the numbers series.

Variable Randomizer

To increase the possibility that a different seed number will be selected each time RANDOMIZE is encountered, we can let the Computer make that selection for us. The RANDOMIZE statement can be followed by a variable or numeric constant between −32768 and +32767.

Let's use a close relative to the TIME$ function. Enter the Command window and type:

```
CLS          Return
PRINT TIMER          Return
```

The number displayed is the number of seconds that have elapsed since midnight. (Just what we wanted to know!) We can use TIMER to set the random seed. Type in this short New program:

```
10 RANDOMIZE TIMER
20 FOR R = 1 TO 10
30   PRINT RND;
40 NEXT R
```

...and Run

...and Run again.

We get different numbers each time! But what if the value of TIMER is greater than 32767 (which it would be if it were after nine o'clock in the morning)? We would surely get an error if we used, say a constant of 35000, as a RANDOMIZE seed value. It turns out that the Macintosh does some tricky internal manipulation of the TIMER value so that Microsoft BASIC does not "crash" on us.

The "randomness" of this scheme is based on the unpredictability of the number of seconds that have passed since midnight. Care to guess that value after glancing at your watch?

The Old Coin Toss Gambit

We could toss a thousand heads in a row, and the odds on the next toss are *exactly 50/50* that a head will come up next. The outcome of every toss is totally independent of what happened before. **It is too!**

In the *long run*, however, the number of heads and tails should be exactly the same. (Casinos live off people who go broke waiting for their particular

scheme to pay off ... "in the long run.") The Computer can provide an education in "odds" and various games of chance and allow us to prove or disprove many ideas involving probability. This is known as computer "modeling," or "simulation."

Type in this coin toss simulation:

```
10 RANDOMIZE
20 INPUT "NUMBER OF COIN FLIPS";F
30 PRINT "STAND BY WHILE I'M FLIPPING."
40 FOR N=1 TO F
50  X = INT(RND * 2 + 1)
60  ON X GOTO 90,110
70  PRINT "WAS NEITHER A HEADS NOR TAILS."
80  END
90  H = H + 1
100  GOTO 120
110  T = T + 1
120 NEXT N
130 PRINT "HEADS","TAILS","TOTAL FLIPS"
140 PRINT H,T,F
150 PRINT INT(100*H/F);"%",INT(100*T/F);"%",
    "100%"
```

...and Run.

Seed the generator with the number 1, and "Flip the coin" 100 times. Run a number of times, changing the seed. When it's time for lunch, try 25,000 flips or more.

Line 10 establishes a Random seed value.

Line 20 INPUTs the number of flips desired.

Line 30 Prints a "Standby" statement.

Line 40 begins a FOR-NEXT loop that Runs "F" times.

Line 50 is the RND generator. We told it to generate INTegers between 1 and 2, and that restricts it to just the numbers 1 and 2. Heads is "1," and Tails is "2."

Line 60 has an ON-GOTO test sending X=1 to Line 90 where the "Heads" are counted and X=2 to Line 110 where the "Tails" are counted.

Lines 70 is the default Line. If X = other than 1 or 2, the error message will be PRINTed and execution will END. It will never happen, but here is the proof.

Line 90 sets up H as a counter. Each time the ON-GOTO tests sends control to this Line because X=1, H is incremented by one and keeps count of the "Heads."

Line 100 sends control to Line 120 where NEXT N is executed. When the N Loop has gone through all "F" number of passes, control drops to Line 130.

Until then, Line 50 generates another RaNDom number (1 or 2). If the next X = 2,...

Line 60 sends control to Line 110.

Line 110 keeps track of the "Tails."

Line 130 PRINTs the Headings.

Line 140 PRINTs the values of H, T and F.

Line 150 calculates and PRINTs the percentage of heads and percentage of tails.

Save this program As `COINTOSS`.

More Than One Generator At A Time

It is possible to generate more than one random number in a program by using more than one generator. This has special value when the ranges of the generators are different, but is helpful even if their ranges are the same.


I GUESS I CAN'T COMPLAIN – I ASKED FOR RANDOM NUMBERS

It could also be done with a single generator, but that wouldn't make the point.

To make the point, we will simulate the game of "Craps" -- where 2 dice are "rolled." Each "die" has six sides, and each side has 1,2,3,4,5 or 6 dots. When the 2 dice are rolled, the number of dots showing on their top sides are added. That sum is important to the game. Obviously, the lowest number that can be rolled is 2, and the highest number is 12. We will set up a separate Random Number Generator for each die, give each a range from 1 to 6, and call them die "A" and die "B."

Type NEW, then the following:

```
10 A = INT(RND*6+1)
20 B = INT(RND*6+1)
30 N = A + B
40 PRINT N,
50 GOTO 10
```

...Run.

Each number PRINTed falls between 2 and 12. We only need to PRINT N since the dice are both thrown at the same time, and only the *sum* of the 2 is what counts.

Remember to press [Ctrl] [Break] to stop the Computer.

Why would the following by wrong? It creates numbers between 2 and 12.

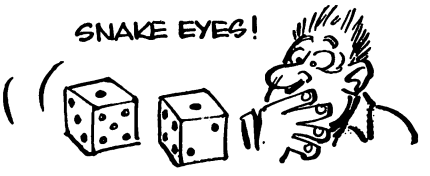
```
10 PRINT INT(RND*11+2)
```

Answer: Adding random numbers created by two generators, each picking numbers between 1 and 6, will create many more sums which equal 3,4,5,6,7,8,9,10 and 11 than a single generator which picks an equal amount of numbers 0 through 10, to which we add 2, to make the range 2 through 12. To simulate 2 dies, the generator range must be 1-6, twice.

Rules Of The Game

In its simplest form, the game goes like this:

1. The player rolls the two dice. If the sum is 2 (called "snake eyes"), a 3 ("cock-eyes"), or a 12 ("boxcars") on the first roll, he loses, and the game is over. That's "craps."



2. If the player rolls 7 or 11 on the first throw (called "a natural"), he wins, and the game is over.

3. If any other number is rolled, it becomes the player's "point." He must keep rolling until he either "makes his point" by getting the same number again to win or rolls a 7 and loses.

EXERCISE 18-1: You already know more than enough to complete this program. Do it. Put in all the tests, PRINT Lines, etc. to meet the rules of the game and tell the player what is going on. It will take you awhile to finish, but give it your best before turning to Section B for a sample solution. Good luck!

Learned In Chapter 18

Functions

RND(X)
RANDOMIZE R
TIME$
TIMER

Miscellaneous

Seed numbers
Pseudo-random

Chapter 19

READing DATA

We have learned how to insert numeric values into programs by two different methods. The first is by building them into the program:

```
10 A = 5
```

The second is by using an INPUT statement to enter them through the keyboard:

```
10 INPUT A
```

The third principal method uses the DATA statement.

Type in this New program:

```
10 DATA 1,2,3,4,5
20 READ A,B,C,D,E
30 PRINT A;B;C;D;E
```

...and Run.

The DATA statement is in some ways similar to the first method in that a Line holding the values is part of the program. It's different, however, since each DATA Line can contain many numbers, or pieces of data, each separated by a comma. Each piece of DATA must be read by a READ statement. Each READ Line can hold a number of READ statements, each separated by a comma.

The display shows that all 5 pieces of DATA in Line 10, the values 1,2,3,4 and 5, were READ by Line 20, assigned to variables A through E, and PRINTed by Line 30.

Keep in mind these important distinctions: DATA Lines can be read *only* by READ statements. If more than one piece of DATA is placed on a DATA Line, they must be separated by commas. INPUT statements are used to enter data directly from the keyboard.

DATA Lines are always read from left to right by READ statements; the first DATA Line first (when there is more than one), and *it does not matter where they are in the program.* This may seem startling, but do the following and see:

1. Move the DATA Line between Lines 20 and 30 (don't bother to change Line numbers), and Run. No change in the PRINTout, right?

2. Move the DATA Line to the end of the program. Same thing -- no change in the PRINTout.

DATA Line(s) can be placed anywhere in the program.

This fact leads different programmers to use different styles. Some place all DATA Lines at the beginning of a program so they can be read first in a LISTing and found quickly, to change the DATA.

Others place all DATA Lines at a program's end where they are out of the way. Still others scatter the DATA Lines throughout the program, next to the READ Lines. The style you select is of little consequence -- *but consistency is comfortable.*

The Plot Thickens

Since we now know all about FOR-NEXT loops, let us see what happens when a DATA Line is placed in the middle of a loop. Erase the old program by selecting New, and type in this program:

```
DATA 1,2,3,4,5
10 FOR N = 1 TO 5
20   READ A
30   PRINT A;
```


Y'KNOW SOMETHIN' FRIEND? YOU'RE NOT HALF AS SCARY AS IN THE BEGINNING.
RATS! MY COVER'S BLOWN!

```
40 NEXT N
```

...and Run.

That DATA Line is outside the loop. Now move it between Lines 10 and 20 and Run. What happened?

Nothing different! It is important to absorb this fact, or we wouldn't have gone to the trouble to prove it. We went through the N loop 5 times, READ the letter A 5 times, and the PRINT statement PRINTed A 5 times, but A's value was *different* each time. Its value was what it last READ from the DATA Line. The reason -- each piece of data in a DATA Line can only be read *once* each time the program is RUN. The next time a READ statement requests a piece of data, it will read the NEXT piece of data in the DATA Line or if that Line is all "used up," move on to the next DATA Line and begin READing it.

Change Line 10 in the program to read:

```
10 FOR N = 1 TO 6
```

...and Run.

The READ statement was instructed to read 6 pieces of DATA, but there were only 5. An error statement caught it, as the dialog box shows.

```
Out of DATA
```

and `READ A` is boxed.

Click the **OK** box, then change Line 10 so the number of READs is *less* than the DATA available.

```
10 FOR N = 1 TO 4
```

...and Run.

No problem. It works just fine even if we don't use all the available data. The point is, each piece of data in a DATA statement can only be READ once during each Run.

Exceptions, Exceptions!

Because it is sometimes necessary to read the same DATA more than once without RUNning the complete program over, a statement called RESTORE is available. Whenever the program comes across a RESTORE, *all* DATA Lines are RESTOREd to their original "unread" condition, both those that have been READ and those that have not, and all are available for reading again. Change Line 10 back to:

```
10 FOR N = 1 TO 5
```

and insert:

```
25 RESTORE
```

...and Run.

Oh-oh! The screen PRINTed five 1's instead of 1 2 3 4 5. Can you figure out why?

> Line 20 READ A as 1, but Line 25 immediately RESTOREd the DATA Line to its *original un*READ *condition*. When the FOR-NEXT loop brought the READ Line around for the next pass, it again read the first piece of data, which was that same 1. Same thing happened with the remaining passes.

READ and DATA statements are extremely common. RESTORE is used less often.

Do you begin to see some distant glimmer involving the storing of business or technical DATA in DATA Lines where it's easily changed or updated without affecting the rest of the program or its formulas?

String Variables

Who knows where some of these seemingly unrelated words come from? If they weren't so important, we could ignore them. We have been using the letters A through Z to hold number values. They are called *numeric variables*. We can use the same 26 letters to hold *string variables* by just adding a "$."

A$, for example, is called "A String." String variables can be assigned to indicate *letters, words* and/or *combinations* of letters, numbers, spaces and

other characters. Choose New, then type in:

```
10 INPUT "WHAT IS YOUR NAME";A$
20 PRINT "HELLO THERE, ";A$
```

...and Run.

Hey-hey! How's that for a grabber? If that, along with what we have learned in earlier Chapters doesn't make the creative juices flow, nothing will.

That's Two....

We now know two ways to PRINT words. The first, learned long ago, is to imbed words in PRINT statements (and is called "PRINTing a string"). The second is to bring word(s) through an INPUT statement (called "INPUTting a string"). If you can't think of the third way, go back and check the title at the top of this Chapter.

Select New, and type in this program:

```
10 READ A$
20 DATA APPLE MACINTOSH COMPUTER
30 PRINT "SEE MY ";A$
```

...and Run.

```
SEE MY APPLE MACINTOSH COMPUTER
```

Let's use 2 string variables to accomplish the same thing, seeing how they work with each other. Reword the program to read:

```
10 READ A$
15 READ B$
20 DATA APPLE, MACINTOSH COMPUTER
30 PRINT "SEE MY ";A$;" ";B$
```

...and Run.

Analyzing the program:

Line 20 contains two pieces of string Data, separated by a comma.

Line 10 READs the first one.

Line 15 READs the second one.

Line 30 contains 4 PRINT expressions:

The first one PRINTs "SEE MY ", leaving a space behind the "Y" since, unlike numeric variables, string variables do *not* insert leading and trailing spaces. This gives excellent control over PRINT spacing.

The second PRINT expression is A$, and it prints "APPLE".

The third inserts the space which is enclosed in quotes.

The fourth PRINT expression is B$ which PRINTs "MACINTOSH COMPUTER".

Together, they PRINT the entire message on the same line.

A semi-colon between STRING variables does *not* cause a space to be PRINTed between them. We have to insert a space using " " marks.

Learned In Chapter 19

Statements	Miscellaneous
READ	String Variables A$, B$,...
DATA	Numeric Variables
RESTORE	

PART 3

STRINGS

Intermediate BASIC

Intermediate Features Of Microsoft BASIC for the Apple Macintosh

Now that we've learned the rudiments of "Elementary" BASIC, we can get serious about "Intermediate" BASIC. The next Chapter is sort of a "catch up" and "catch all," explaining a lot of little unrelated features that didn't find convenient homes in the previous Chapters. Study each of them, do the sample programs and think about them. Each one is brief but important.

Chapter 20

Smorgasbord

Multiple Statement Lines : (Now he tells us!)

BASIC allows more than one consecutive statement on each program Line, separated by a colon (:). For example, a timer loop such as:

```
100 FOR N = 1 TO 500
110 NEXT N
```

can become...

```
100 FOR N = 1 TO 500 : NEXT N
```

Caveat Emptor *(Don't buy a used computer from a stranger.)*

Control yourself! It's easy to get carried away with this exciting feature. While we will use multiple statement Lines often from here on, you will quickly find that it's possible to pack the information so tightly it becomes hard to read, and also very hard to modify.

More Caveat *(or is it more Emptor?)*

Multiple statement Lines require careful understanding. Especially critical are statements of the IF-THEN variety.

Enter the following *incorrect* program:

```
10 INPUT "TYPE IN A NUMBER";X
20 IF X = 3 THEN 50 : GOTO 70
30 PRINT "HOW DID YOU GET HERE?"
40 END
```

```
50 PRINT "X=3"
60 END
70 PRINT "CAN'T GET FROM THERE TO HERE."
```

...and Run it several times with different INPUT values, including 3.

Line 20 has an error in logic. If the IF-THEN test passes, control moves to Line 50. That's OK.

If the test fails, however, control drops to the next Line in the program -- Line 30, not to the 2nd statement in Line 20. **There is no way the 2nd statement in Line 20 (GOTO 70) can ever be executed.**

The Message -- if you put an IF-THEN (or ON-GOTO) type-test in a multiple statement Line, it must be the *last* statement in that Line.

Next Message -- we cannot send control TO any point in a multiple statement Line except to its FIRST statement. Look at Line 20. There is no way to address the GOTO 70 portion. It shares the same Line number as the first statement in the same Line. Only the first statement is addressable by a GOTO or IF-THEN. Other statements in a Line are accessed in sequence, IF each prior test is passed.

Searching The Program

Now that we are beginning to develop larger and more complex programs, it becomes more difficult to find something buried deep within the program.

The Search menu has several methods that can be used to Search for individual characters, text or the cursor. It's also possible to Search for a letter or text and replace it with something else. This is useful when changing a name or variable used throughout a program to a different name or variable. For example, let's use the Search feature to find the location of each variable X in our resident program.

Place the cursor at the beginning of the program and select Find... from the Search menu or press ⌘F. A dialog box appears with the cursor flashing in the **Find next** box. Type in the letter X (upper or lower case -- it isn't fussy), and click the **OK** box or press Return.

The Computer found the first X in Line 10 and pointed it out by reversing the letter.

To Find the Next letter X, press ⌘ N (or select Find Next from the Search menu). Continue pressing ⌘ N, and notice that after the last X is found, the Computer returns to the first one in Line 10.

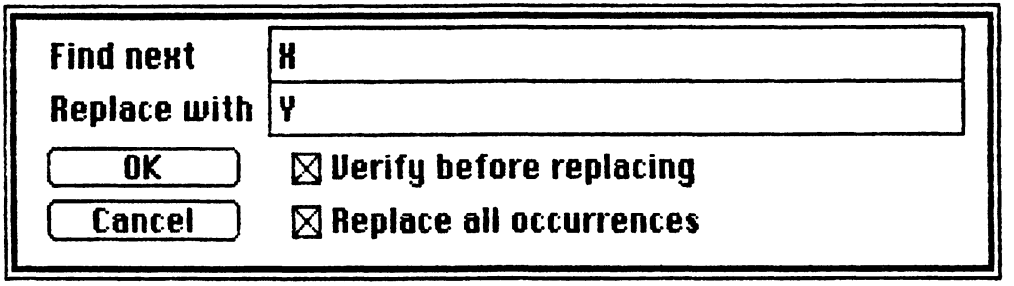
Now let's change variable X to Y.

Place the pointer at the beginning of the program, and click the mouse. This sets the letters to the normal font and places the cursor at the beginning of the program.

Select Replace... from the Search menu. Another dialog box appears. It looks like the Find dialog box except in this box we can specify a Replacement for what is listed in the **Find next** box.

The letter X is still sitting in the **Find next** box. Let's leave it there, and place a Y in the **Replace with** box. Position the pointer inside the **Replace with** box and click. Type the letter Y (use upper case -- it matters this time), but don't hit Return.

We also have the options of "Verifying before replacing" and "Replacing all occurrences." In this example, we want to change all X variables, so click the box for "Replace all occurrences." Then click the box for "Verify before replacing" so we can monitor the action. Your dialog box should look like this:



Select **OK**. The Computer should have found the first X in Line 10 and displayed the **Replace verify** box in the upper right-hand corner. Click **Yes** for the three occurrences of the variable X in the program. As long as the **Verify** box is present, there are more X's to be replaced.

Selective Searching

As with most other features in our Mac, there is a short cut to selecting and Searching text.

Place the cursor to the right of the 70 in Line 20. Press the mouse button and drag to the left until the number 7 and 0 are reversed. Be careful not to select more than the two numbers, then release the button. Now select Find Selected Text from the Search menu, and bingo, the Computer found the 70 in Line 70.

In a very large program, a GOTO or GOSUB Statement may be specifying a Line number or Label that is residing far down the program. With the Find Selected Text feature we can easily select any portion of text by shading it with the mouse and find other occurrences of it in the program.

Try finding other selected text such as the PRINT statements or the value 3 in Lines 20 and 50 until you get a handle on all the Search features.

After we have written a program that has more Lines than the screen can hold, we can use the Search menu to Find the Cursor. Selecting the Find the Cursor option causes the Computer to scroll down the program and display the program starting with the Line where the cursor is positioned. Again, these are features that will come in handy when working with very long programs.

New Numeric Variables

We know we can use the 26 letters of the alphabet as names for variables. We can also use the numbers 0 through 9 in conjunction with these letters:

```
A3 = 65
F9 = 37
```

etc.

Although the 26 letter variables are usually enough, addition of the numbers give us an additional 26*10 = 260. They can be very handy, particularly if we want to label a number of "sub" variables (D1,D2,D3,etc.) which combine to make a grand total which we can just call D.

```
PI = 3.14159
```

```
C = PI*D          (Circumference = 3.14159 * Diameter)
```

In addition, we can use any combination of upper- and lowercase letters, num-

bers and decimal points (or periods) for a name, up to 40 characters long. For example:

```
LEARNING.MICROSOFT.BASIC = 19.95
```

Now that really looks valuable.

If that doesn't provide enough variables to solve your problems, nothing will.

New String Variables

So far we've used only A$ and B$ as string variables. We actually have *all* the letters of the alphabet available for strings. And the numbers 0 through 9 too, plus any letter-number combination. These are valid string names:

X$

D8$

PI$

WHAT.A.GREAT.BOOK$

etc.

As with numeric variables, string variables can have any combination of up to 40 letters (upper- or lowercase), numbers and decimal points (periods) followed by the $ sign.

Upper- and lowercase letters can be used in both numeric and string variable names although the Computer cannot distinguish between the two. For example, Pi and PI would be treated as the same variable, as would x2$ and X2$.

Shorthand

There are several little "shorthand" tricks available.

The first is the use of ? in place of the very common word, PRINT. Select New, then type this Line:

```
10 ?"QUESTION MARK"          Return
```

Awwk! The pumpkin turned into a coach. The Computer rewrote it to read:

```
10 PRINT"QUESTION MARK"
```

It also works at the command level. Enter the Command window and try:

```
?3*4
```

and we get:

```
12
```

Try ?FRE(0).

The ' is shorthand for REM and is especially nice when documenting the purpose of a Line. It makes program Lines into multiple statement Lines as in ' = :REM.

```
50 X = Z*C/4 + 33        'THE SECRET EQUATION
```

The only place ' can't be used unaided is in a DATA Line, and that problem can be overcome by actually adding a :.

```
1000 DATA 102,3,9,105,10,1 : 'DATA IS IN
1010 DATA 108,7,3,111,6,1 : 'SEQUENCE
```

Use Of Quotes

Technically, it is not necessary to use quotes to close off many PRINT statements, or LOADs and SAVEs.

```
10 PRINT "WHERE IS THE END QUOTE?
```

Note lack of second ".

Runs just fine, but leave it off at your own peril.

A BASIC interpreter that is "too forgiving" is like an airplane that is "too forgiving." It allows us to become sloppy, and when we need all the skill we can muster, it is gone from the lack of practice. You are strongly encouraged *not* to take these and other "cheap" short-cuts.

INPUT

It's possible to INPUT several variables with a single INPUT statement. Type this program and respond with a cluster of 3 numbers separated by commas. It will "swallow" them all in one gulp.

```
10 INPUT A,B,C
```

...and Run.

However, if we fail to INPUT them all, separated by commas, the error:

```
?Redo from start
?
```

points out that more DATA must be INPUT. To see the Error Message, Run again, but only INPUT one number, Return, then type ⌘ . to bail out.

Run again and try to INPUT letters instead of numbers. Same Error Message.

There is extensive information in Appendix C dealing with Error Messages. Most often, *Redo* reminds us that we can't INPUT a string variable into a request for a numeric variable.

Optional NEXT

FOR-NEXT loops don't always have to specify which FOR we are NEXTing. This can be useful when dealing with nested loops.

Type this New program:

```
10 FOR A = 1 TO 2 : PRINT A
20    FOR B = 1 TO 3 : PRINT ,B
```

HE JUST WORKED 3 HRS
ON A PROGRAM and
THE COMPUTER SAID:

```
30       FOR C = 1 TO 4 : PRINT ,,C
40 NEXT : NEXT : NEXT
```

Run it several times to get the flavor. (Note how commas were used to place PRINTing in different zones.)

This method of NEXTing should not be used if the program contains tests which might allow a loop to be broken out of. Better then to be specific, or use this little short-cut:

```
40 NEXT C,B,A
```

IF-THEN-ELSE

ELSE is an interesting addition to our stable of conditional branching statements. It allows an option other than dropping to the next Line if a test fails. Try this New one:

```
10 INPUT "ENTER A NUMBER";N
20 IF N=0 THEN PRINT "0" ELSE PRINT "NOT 0"
```

...and Run.

255 Characters Per Line

Microsoft BASIC permits up to 255 characters in a single program Line. (Don't ask **me** to debug such a Line!)

Another Way Of Leaving BASIC

In the past, whenever we wanted to leave BASIC to return to the Finder, we pulled the File menu down and selected Quit. As you may have discovered, QUIT will not work from the keyboard. Instead, we need to use the SYSTEM command.

Learned In Chapter 20

Commands	Statements	Miscellaneous	Menu
SYSTEM	IF-THEN-ELSE	Multiple statement Lines	Search
		Variable Names	Find... (⌘F)
		Some Shorthand	Find Next (⌘N)
		Quotes	Find Selected Text
		Multiple INPUTting	Find the Cursor
		Optional NEXT	Replace...
		String Variables	

Chapter 21

The ASCII Set

The purpose of this Chapter is to learn how to use ASC and CHR$. Before doing so, however, we must learn about something called "the ASCII set."

ASCII is pronounced (ASK'-EE) and stands for American Standard Code for Information Interchange. Since a computer stores and processes only numbers, not letters or punctuation, it's important that there be some sort of uniform system to specify which numbers represent which letters and symbols. The ASCII Chart in Appendix A shows the relationship between the number system and symbols as used in the Macintosh. Take a minute to review the chart.

Type in this New program:

```
10 FOR N = 32 TO 217
20   PRINT "ASCII NUMBER";N;
30   PRINT "STANDS FOR";,CHR$(N)
40   FOR T = 1 TO 500 : NEXT T
50 NEXT N
```

Save As `ASCII` ... and Run.

Observe that the characters between ASCII code numbers 97 and 122 are lowercase duplicates of ASCII numbers 65 to 90. Numbers 128 to 217 call forth special graphics and foreign language characters.

The Option Key

We can use the keyboard to enter all those special characters by using the Option key. (There are two of them, one below each Shift key.) To use the Option key, hold it down while pressing another key on the keyboard, in the same manner as we would use the ⌘ key.

Instead of "poking around in the dark" to find out which Option sequences produce the different characters, there is a desktop accessory called "Key Caps" in the Menu. Use the mouse to pull down the menu, and select Key Caps.

Like all windows, we can move it around the desk top by pointing at the title bar while dragging the mouse.

With the Caps Lock key released, hold down the Shift key and notice the screen change. The two Shift keys are darkened, the lowercase letters are replaced with uppercase and the number keys are replaced by the special characters.

With the Caps Lock key off, hold down the Option key. Aha! There are some of the special characters and their keyboard placement. Note that some are blank (□). Now, while holding down Option, press Shift. The rest of the special characters are then displayed.

Above its picture of the keyboard, Key Caps has a small window available for displaying anything we type on our keyboard. Choose some of the special characters, and enter them. Use the Backspace key to clear the window.

After experimenting with Key Caps for a while, click the close box (small white box) on the title bar to remove it from the desk top.

Note that we can select Key Caps for more experimentation at any time, even during INPUT while Running a program. And if we feel really lazy, we can even use the mouse to click the keys on Key Caps keyboard!

ASCII Chart

Some of the ASCII numbers between 0 and 31 are used by the Macintosh for special control purposes:

Code	Function
7	Beep
8	Backspace and erase current character
9	TAB(9,17,25...)
10	Move cursor to beginning of next Line
12	Clear screen
13	Move cursor to beginning of next Line (Return)

Change Line 10 to read:

```
10 FOR N = 1 TO 31
```

and Run again.

There is very little uniformity internationally (or even within the U.S.) in the assignment of ASCII code numbers, except those used for the "Roman" letters and numbers. Fortunately, they handle most of our everyday needs. If we contemplate the problems faced by users of other languages which need special letters and characters, it's easy to see how good use can be found for the ASCII values between 128 and 216.

So What Is CHR$(N)?

We have used CHR$ (pronounced Character String) without describing it, but you undoubtedly figured it out anyway. CHR$(N) produces the ASCII character (or control action) specified by the code number N. It is a one-way converter from the ASCII *code number* to the ASCII *character* and allows us to throw characters around with the ease of throwing around numbers. The word "string" refers to any character or mixture of characters (letters, numbers or punctuation).

Enter this simple New program:

```
10 INPUT "TYPE ANY NUMBER (33-217)";N
20 PRINT CHR$(N)
30 PRINT : GOTO 10
```

...and Run.

Don't forget to press [Esc] [.] or select Stop from the Run menu to break out of the loop.

Almost all of our activity with ASCII numbers will be confined to this range.

EXERCISE 21-1: Using the ASCII chart (Appendix A) and the CHR$ function, create a program which will PRINT the name: MACINTOSH.

ASCII Applications

If we end up in the Big House serving time for computer fraud, the following little program will make up our license plate combinations, putting CHR$ to good use.

Enter this New program:

```
10 REM * LICENSE PLATE NUMBER GENERATOR *
20 FOR N=1 TO 3 : PRINT INT(RND*10);
30 NEXT N : PRINT "  ";
40 FOR N=1 TO 3
50  PRINT CHR$(INT(RND*26) + 65);"  ";
60 NEXT N : PRINT : GOTO 20
```

Save As `LICPLATE` ... and Run.

The RND generator in Line 20 PRINTs numbers between 0 and 9. Line 50 PRINTs those characters whose ASCII number falls between 65 and 90 by producing a RaNDom INTeger between 0 and 25 and then adding 65 to it. What do we see on the ASCII conversion chart between 65 and 90? Hmmmm???

What Then Is ASC($)?

ASC is the exact opposite of CHR$(N). ASC is a one-way converter from the ASCII *character* to its corresponding ASCII *number*.

Select New and type:

```
10 INPUT "TYPE NEARLY ANY CHARACTER";A$
20 PRINT "ITS ASCII NUMBER IS";ASC(A$)
30 PRINT : GOTO 10
```

...and Run.


NOW, WHAT'S ALL THIS BABBLING ABOUT "CHR$(N) and ASC($)?"

It will PRINT the ASCII number of almost all characters. (Try lower case letters and special code characters which use the Option key, too.) It doesn't work with the comma (,), the quotation mark ("), the space bar, and some others, but then strings can be a real mystery at times, as we will see.

To get around this and other problems, we use an advanced form of INPUT called LINE INPUT. LINE INPUT allows us to INPUT *any* character (that is assigned an ASCII code) as a string. Notice that the Computer will not insert a question mark when it asks for the text.

Change Line 10 to:

```
10 LINE INPUT "TYPE IN ANYTHING ";A$
```

...and Run. Check comma, quote, space bar, etc.

An obscure way to use ASC is to imbed the character within quotes, thus:

```
10 PRINT ASC("A")
```

but this latter method is rarely convenient.

Home Base

So far we have talked exclusively about *decimal* numbers, since most of us have just 10 fingers. But the Macintosh has two intrinsic functions which convert decimal numbers to numbers with Hexadecimal and Octal Bases. Whereas the Decimal system is built on the base 10 (10 digits, 0-9), Octal is base 8 (8 digits, 0-7), and Hexadecimal is base 16 (16 digits, 0-9 and A-F). We'll leave the mechanics of using other bases to other CompuSoft books, but just to be complete, select New, and type in this New program:

```
10 LINE INPUT "TYPE ANY LETTER, NUMBER OR
   CHARACTER: ";A$
20 A = ASC(A$) : PRINT
30 PRINT "ITS ASCII NUMBER IS";ASC(A$)
40 PRINT,A,"DECIMAL"
```

```
50 PRINT,HEX$(A),"HEXADECIMAL"
60 PRINT,OCT$(A),"OCTAL"
70 PRINT : GOTO 10
```

Save As BASECONV ... and Run.

Before we can really understand the importance of CHR$ and ASC, we must learn a lot more about strings. Before we could learn about strings, we had to learn something about ASCII. It's like "catch Macintosh."

> **EXERCISE 21-2:** Input a single character from the keyboard, and test its ASCII value to determine IF it is a number. If not, return program control to the INPUT statement. Hint: use two IF statements and ASC.

Learned In Chapter 21

Functions	Statements	Miscellaneous	Menu
CHR$	LINE INPUT	ASCII Codes	
ASC		Option key	Key Caps
HEX$		Special characters	
OCT$			

Chapter 22

Strings In General

It was not our intention to "string you along" in the previous Chapter, but we really can't understand how strings work without first understanding the ASCII concept of numbers standing for letters, numbers and other characters and controls.

Comparing Strings

One of the most powerful string handling capabilities is the ability to *compare* them. We compare the values of *numeric* variables all the time. How can we compare *strings* of letters or words? Well, why do you suppose we put the ASCII Chapter just before this one? **Right!** The Computer can compare the ASCII *code numbers* of letters and other characters. The net result is a comparison of what's in the corresponding strings.

Type in this New program:

```
10 INPUT "WHAT IS YOUR NAME";A$
20 IF A$ = "ISHKIBIBBLE" THEN 50
30 PRINT "SORRY. WRONG NAME!"
40 END
50 PRINT "FINALLY GOT IT!"
```

...and Run.

If the Computer can compare A$ against *that* name, it should be able to compare anything!

During the process of comparing what you enter as A$ in Line 10 to what's already in quotes in Line 20, the ASCII code numbers of each letter found in one string are compared, letter for letter, from left to right with those in the other. Every one must match, or the test fails.

Strings and "quotes" are inseparable. You know this from earlier Chapters where every PRINT "XXX" has its string enclosed in quotes.

PRINT "XXX" is called a string *constant*. A$ is a string *variable*.

Run the above program again, this time answering the question with "ISHKIBIBBLE," but enclosed in quotes.

Sure -- it ran OK.

READing Strings

A string can be INPUT with or without quotes. BASIC has become increasingly lenient about this matter, but every once in a while the rules come up from behind and bite us if we play fast and loose with them.

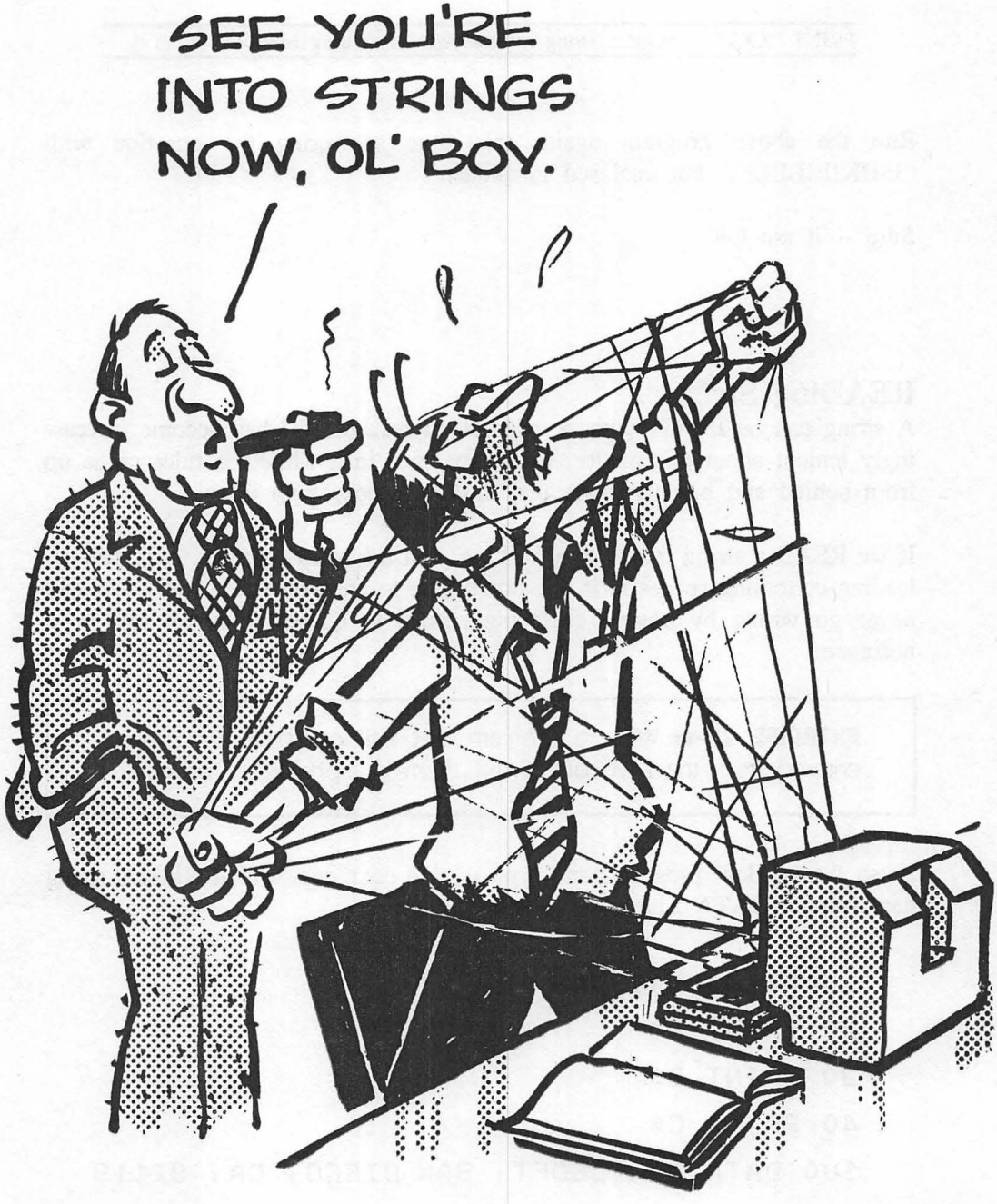
If we READ a string from a DATA Line, and it has no commas, semi-colons, leading or trailing spaces in it, we don't *have* to enclose it in quotes. We will never go wrong by *always* enclosing strings in quotes, but that can be a nuisance.

EXERCISE 22-1: Write a program that will compare two strings entered from the keyboard. PRINT them in alphabetical order.

Erase the resident program, and type in this next one, which READs string data from a DATA Line.

```
10 READ A$,B$,C$
20 PRINT A$
30 PRINT B$
40 PRINT C$
100 DATA COMPUSOFT, SAN DIEGO, CA, 92119
```

...and Run.


SEE YOU'RE INTO STRINGS NOW, OL' BOY.

Look carefully at the results. The screen shows:

```
COMPUSOFT
SAN DIEGO
CA
```

That's nice, but where is the ZIP Code? And why weren't SAN DIEGO and CA PRINTed on the same line? The answer, my friend, is blowing in the ... er, in the commas.

Because of the commas in the DATA Line, the READ statement sees 4 pieces of DATA, but only READs 3 of them. What do we have to do in order to PRINT a comma as part of a string? Right -- enclose it, or the string containing it, in quotes.

Change Line 100 to read:

```
100 DATA COMPUSOFT, "SAN DIEGO, CA", 92119
```

...and Run.

Aaaah! That's more like it. Notice that we didn't have to enclose *all* pieces of string DATA in separate quotes, but could have.

What would happen if we also enclosed the *entire* DATA Line in quotes, leaving the existing quotes in there? (Think about it, then try it. Every question raised has a specific purpose.)

Our Editor is so easy to use; let's make it read:

```
100 DATA "COMPUSOFT, "SAN DIEGO, CA", 92119'
```

...and Run.

Awwk! Disaster. A "Type mismatch" error in Line 10? Yes, there is no straight-forward way to READ quotes as part of a string, even by enclosing them inside another pair of quotes. The Computer just isn't smart enough to figure out which quote mark is which. The usual way to overcome this BASIC

language deficiency is to substitute ' for each " imbedded inside other quotes. Let's try it:

```
100 DATA "COMPUSOFT, 'SAN DIEGO, CA', 92119"
```

...and Run.

Ooops, "Out of DATA"? Of course. With quotes surrounding the whole works there is now just one piece of DATA, and we are trying to read 3 pieces. Change Line 10 to just read one piece:

```
10 READ A$
```

...and Run.

B$ and C$ are PRINTed as "blanks" since they are empty.

There we go. Might look a little strange, but it demonstrates the point and warns us a little about the "touchiness" of strings.

Learned In Chapter 22

Miscellaneous

String comparison
INPUTting strings
READing strings

Chapter 23

Measuring Strings

One of the most frequently needed facts about a string is its length. Fortunately, the LEN function makes it easy to find. Type:

```
10 INPUT "ENTER A STRING OF CHARACTERS";A$
20 L = LEN(A$)
30 PRINT A$;" HAS";L;"CHARACTERS"
90 FOR X = 1 TO 8000 : NEXT X : RUN
```

See how Run can be used *inside* a BASIC program? The delay loop in Line 90 gives us a chance to read the display before the Computer clears the screen at the next Run.

Enter your name and other combinations of letters and numbers. Try entering your name, last name first, with a comma after your last name.

AHA! Can't INPUT a comma. How about if we put it all in quotes? Try again.

Yep. Just like it said in the last Chapter.

LEN has only one significant variation, and it's not all that useful -- unless it's really needed. Change Lines 10-30 to read:

```
10 INPUT "ENTER A NUMBER";A
20 L = LEN(A)
30 PRINT A;" HAS";L;"CHARACTERS"
```

...and Run, entering any number.

Crash time again! "Type mismatch" means we tried to INPUT a *number* into LEN -- but it requires a *string*.

Letters cause a "?Redo from start" since they need to be INPUT by an A$ or equivalent. Run again, and INPUT a letter. Is there no justice here? OK, let's change LEN to make it a string:

```
20 L = LEN("A")
```

...and Run, entering a Number. Then try bigger numbers.

Hmmm. Doesn't seem to matter what number we INPUT, it always comes back saying that we have only 1 character.

The answer is, LEN evaluates the LENgth of what is actually between its parentheses (or quotes). At first we brought in a string from the "outside" and measured its length. That worked fine. We are now measuring the length of what's actually between the quotes, and that *length* doesn't change with the *value* of A. We are using A as a "literal string constant," not a variable string.

Like we said, this second way to use LEN has its limitations, but don't lose any sleep over it. (Change the resident program back to the way it appears at the beginning of the Chapter.)

DEFSTR -- For Thrill Seekers

Those among us who attract trouble will love this next one. As if handling strings isn't complex enough, this very powerful statement looks nice and clean but in long and complex programs can be the greatest source of heartburn since the horseradish pizza.

DEFSTR (pronounced "DEFine STRing") allows us to define *which* variables are to be *string* variables, so we don't have to use the $ any more. (Hmm ... Uncle Sam could put some of this DEFSTR business to good use.) Add this Line:

```
5 DEFSTR A
```

and change Lines 10 and 20 to:

```
10 INPUT "ENTER A STRING";A
```

```
20 L = LEN(A)
```

Then Run.

Works fine, doesn't it. A was declared by Line 5 to be a string variable. So what's all the fuss about?

Well, this is a very simple program, but let's change 5 to read:

```
5 DEFSTR A-Z
```

which makes *all* letters string variables.

...and Run, entering any character(s).

Crasho again! We got a "Type mismatch." **Too much** of a good thing. Because of Line 5, the L in Line 20 is now *also* a string. Since LEN gives us the length of a string as a number, it doesn't set at all well with L (really L string). Imagine the fun this can create in a long program.

Good thing we can learn by our errors!

DEFSTR is best used to define individual variables. For example:

```
DEFSTR A,N,Z
```

defines only A, N and Z as string variables. Rework Line 5 back to read:

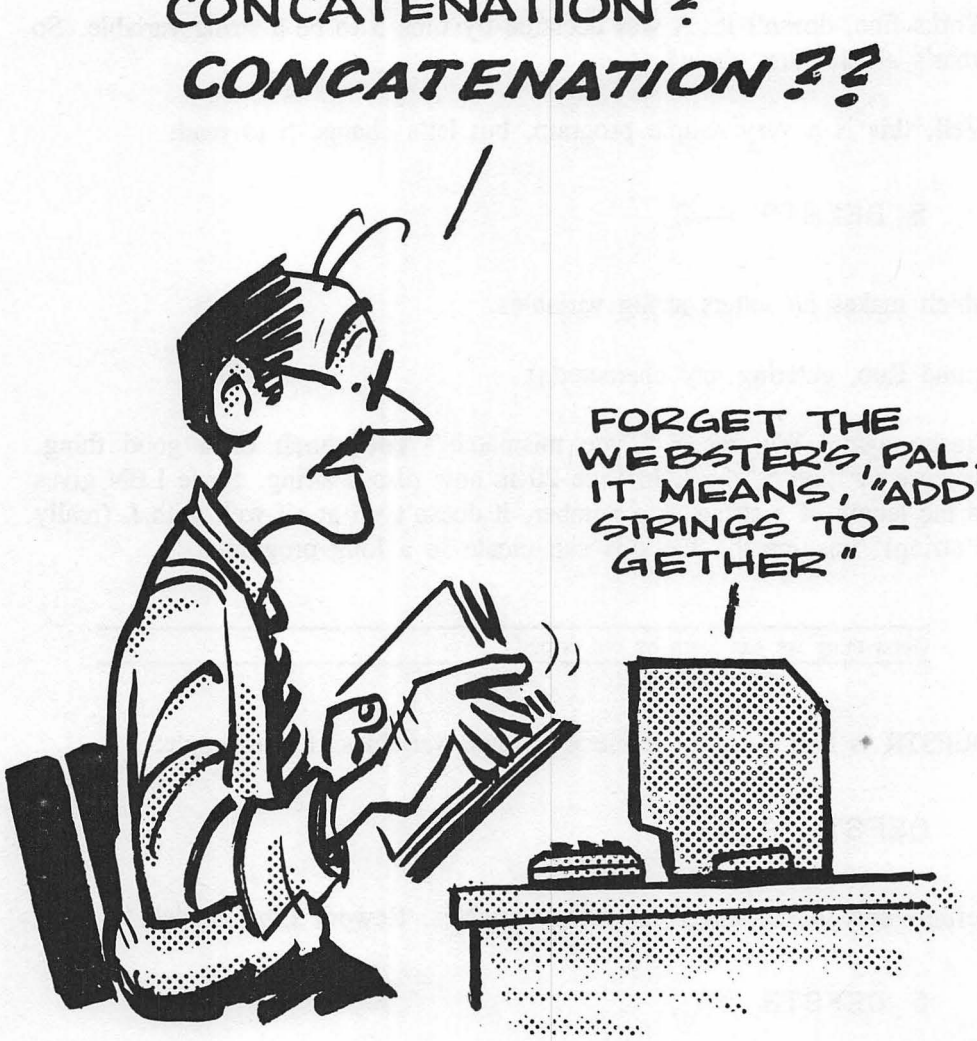
```
5 DEFSTR A
```

...and Run.

That's a short course in what DEFSTR is all about.

Concatenation

Concatenation? Concatenation??? Now what is that supposed to mean? Did you ever wonder who pays whom to sit around and think up such nondescrip-


CONCATENATION?
CONCATENATION??
FORGET THE WEBSTER'S, PAL. IT MEANS, "ADD STRINGS TO-GETHER"

tive words? It must have been done on a government grant. Wait till Senator Proxmire hears about it.

Concatenation (pronounced con-cat-uh-na'shun) is a national debt-sized word which means ADD, as in "add strings together." It's easier to do than to pronounce.

Type this New program:

```
10 FOR N = 1 TO 11
20  READ A$ : B$ = B$ + A$
30  PRINT B$
40 NEXT N
DATA ALPHA,BRAVO,CHARLIE,DELTA
DATA ECHO,FOXTROT,GOLF,HOTEL
DATA INDIA,JULIETTE,KILO
```

Check it carefully, but don't RUN it yet. The key Line is 20, which simply says B$ (a new variable) equals the old B$ (which starts out as nothing) plus whatever is in A$. The program cycles around and keeps adding what is in B$ to what is READ from DATA as A$. Now close the List window, and Run.

Anyhoo, the point of all this is *concatenation*. Line 20 just did it, and that's about all there is to it. We added strings together.

EXERCISE 23-1: Use the LEN function to check the length of a string INPUTted from the keyboard. PRINT a message telling us if the string exceeded 10 characters.

EXERCISE 23-2: INPUT a word from the keyboard, and compare it to a secret password. If there is a match, PRINT "CORRECT PASSWORD; YOU MAY ENTER." If not, PRINT "WRONG PASSWORD. TRY AGAIN!" Store the ASCII number for each letter of the password in a DATA Line. READ each value, and use CHR$ to build (concatenate) the password string.

Learned In Chapter 23

Statements	Functions	Miscellaneous
DEFSTR	LEN	Concatenation (+)

Chapter 24

VAL And STR$

The "hassle factor" can be very high when converting back and forth between strings and numerics.

By definition, if we convert a *numeric* variable (can hold only a number) to a *string* variable (can hold almost anything), the *contents* of that new string is still the original number. No letters or other characters were converted (except for a leading space) since they weren't in the numeric variable to start with.

Conversely, if we change a *string* variable to a *numeric* variable, we can't change any letters or other characters to numbers. Only the *numbers* in a string can be converted to a numeric variable. (Don't confuse this with ASCII conversions.)

If you'll keep the two previous paragraphs in mind, it'll save an awful lot of grief in dealing with strings.

VAL

Let's give string-to-numeric conversion a shot. The VAL function converts a *string* variable holding a *number* into a *number*, if the number is at the beginning of the string. Try this VAL program:

```
10 INPUT "ENTER A STRING ";A$
20 A = VAL(A$)
30 PRINT"THE NUMERIC VALUE OF ";A$;" IS";A
90 PRINT : GOTO 10
```

...and Run

Try lots of different INPUTs, such as:

```
12345
ASDF
123ASD
ASD123
1,2,3
A,B,C
```

and the same ones over again, but enclosed in quotes.

The screen tells all.

Use Stop from the Run menu or press ⌘. to break out of the program, then take the $ out of Lines 10, 20, and 30 and Run, INPUTting both numbers and letters.

What you're seeing is typical of the frustrations that bedevil string users who don't follow the rules. VAL only evaluates STRINGs, and we've put A, a numeric value, in where a string belongs. Does this remind you of the problems in the last Chapter with LEN?

Let's put that A in quotes and see what happens.

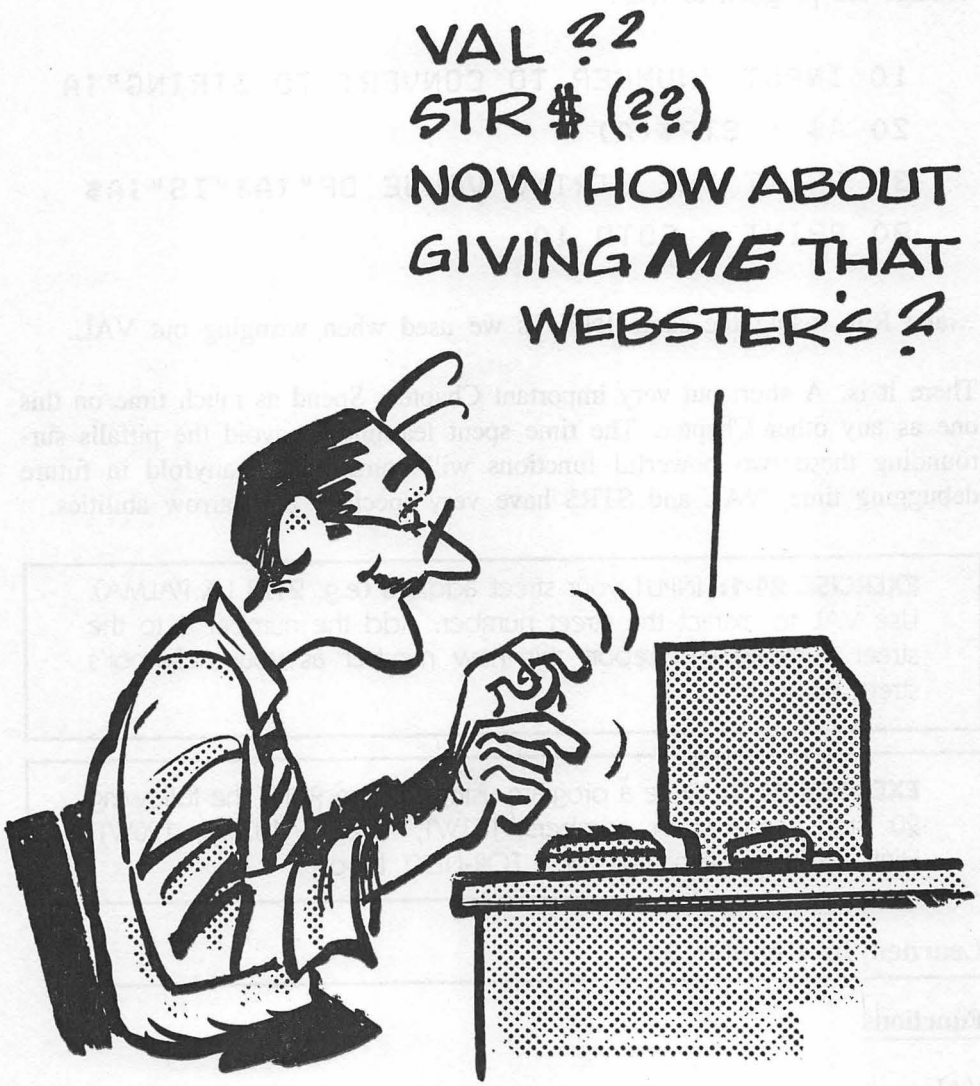
```
20 A = VAL("A")
```

...and Run.

No help at all! The rule remains unchanged.

Properly used, VAL converts a *string* holding a *number* into that *number.*

Looking at the screen you can see all the other uses we are finding for VAL are just not in the cards. Remember this irritating frustration and "The Rule" when you get in the thick of debugging a nasty string-loaded program.


VAL ??
STR$ (??)
NOW HOW ABOUT
GIVING ME THAT
WEBSTER'S ?

STR$

Now let's try the opposite, converting a *numeric* variable to a *string* variable. Change the program to read:

```
10 INPUT "NUMBER TO CONVERT TO STRING";A
20 A$ = STR$(A)
30 PRINT"THE STRING VALUE OF";A;"IS";A$
90 PRINT : GOTO 10
```

...and Run, using the same INPUTs we used when wringing out VAL.

There it is. A short but very important Chapter. Spend as much time on this one as any other Chapter. The time spent learning to avoid the pitfalls surrounding these two powerful functions will come back manyfold in future debugging time. VAL and STR$ have very specific, but narrow abilities.

EXERCISE 24-1: INPUT your street address (e.g. 2423 LA PALMA). Use VAL to extract the street number. Add the number 4 to the street number, and report this new number as your neighbor's street number.

EXERCISE 24-2: Write a program using STR$ to PRINT the following 20 store item stock numbers: 101WT, 102WT, 103WT,...120WT. Hint: Looks like a natural for a FOR-NEXT loop.

Learned In Chapter 24

Functions

VAL
STR$

Chapter 25

Having A Ball With String

LEFT$, RIGHT$, MID$

Three different, yet very similar, functions are used for playing powerful games with strings. They are LEFT$, RIGHT$ and MID$. Let's start with this program:

```
10 S$ = "KILROY WAS HERE"
40 PRINT LEFT$(S$,6),
50 PRINT MID$(S$,8,3),
60 PRINT RIGHT$(S$,4)
```

...and Run.

The screen says:

```
KILROY          WAS             HERE
```

(How about that one, nostalgia buffs?)

Learning to use these string functions is exceedingly simple. Study the program slowly and carefully as we go thru what happened.

LEFT$ PRINTed the LEFTmost 6 characters in the string named S$.

MID$ PRINTed 3 characters in the string named S$, starting with the 8th character from the left. (Count 'em.)

RIGHT$ PRINTed the 4 RIGHTmost characters in the string named S$.

The commas after Lines 40 and 50 are to PRINT everything on the same line.

Save this program As KILROY, then let's move some Lines around to exercise our new-found power. Move Line 50 to Line 30:

```
30 PRINT MID$(S$,8,3),
```

Run ... and we get:

```
 WAS                 KILROY                 HERE
```

Now move Line 60 to Line 20 and add a trailing comma.

```
20 PRINT RIGHT$(S$,4),
```

Run ... and we get:

```
HERE                 WAS                 KILROY
```

These 3 functions can really do wonders with strings. Use Cut and Paste to Edit the resident program to read:

```
10 S$ = "KILROY WAS HERE"
20 FOR N = 1 TO 15
30  PRINT "N =";N,
40  PRINT LEFT$(S$,N)
90 NEXT
```

...and Run.

The picture tells it faster than words. LEFT$ picks off "N" letters from the

LEFT side of S string. See how this string function could be used to strip off only the first 3 digits of a phone number or the first letter of a name when searching and sorting?

Change Line 20 to read:

```
20 FOR N = 1 TO 20
```

Save As LEFT ... and Run.

Even though there are only 15 characters in the string, the overRUN is ignored. Change Line 20 back to N = 1 TO 15.

RIGHT$ works the same way, but from the RIGHT:

Change Line 40 to read:

```
40  PRINT RIGHT$(S$,N)
```

Save As RIGHT ... and Run.

It's the mirror image of LEFT$.

Now let's exercise MID$ and see where it goes. Change Line 40 to:

```
40  PRINT MID$(S$,N,1)
```

Save As MID ... and Run.

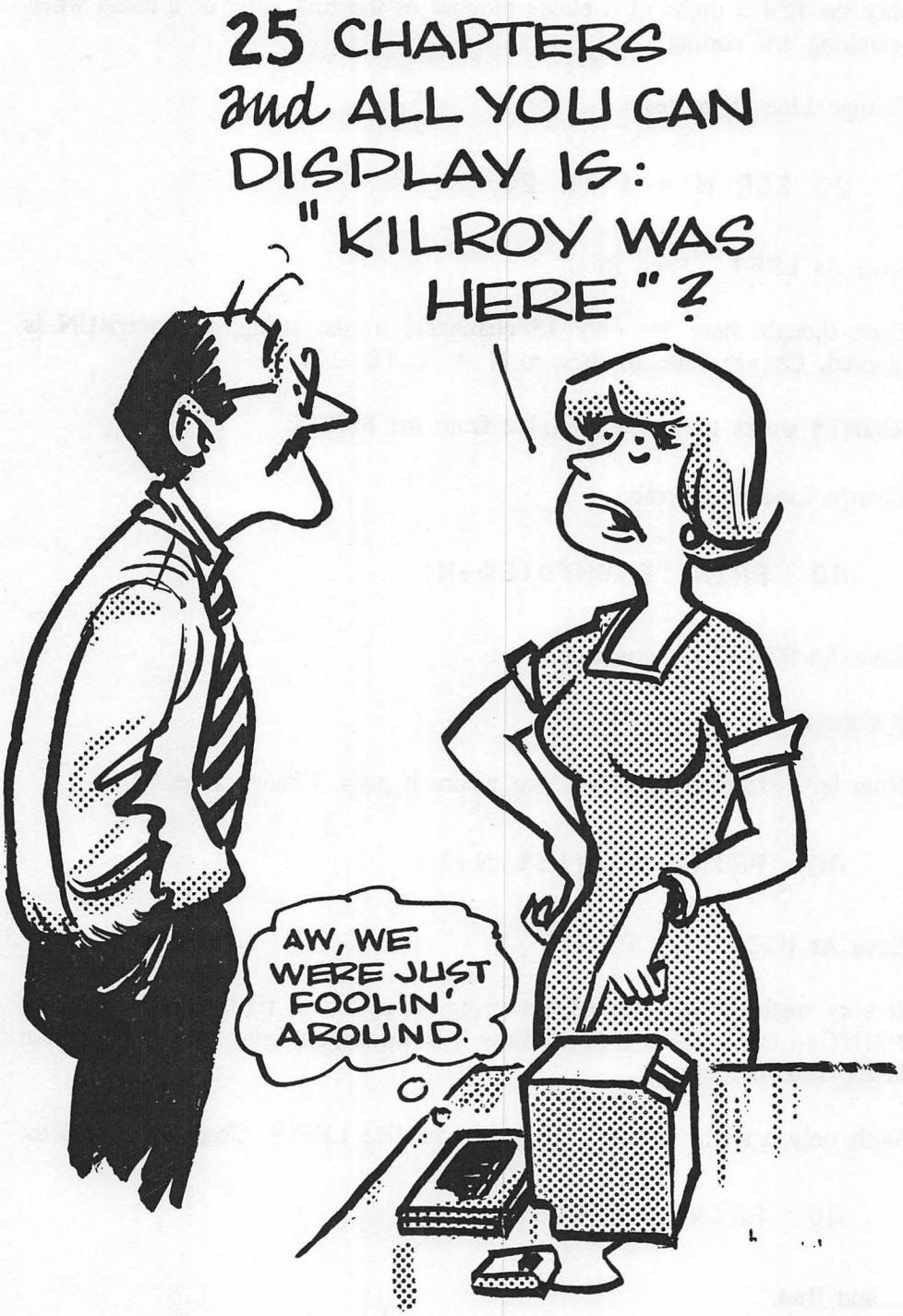
It very methodically scanned the string, from left to right, picking out and PRINTing one letter at a time. Slow it down with a delay loop if the action is too fast to follow.

With only a slight change, MID$ can act like LEFT$. Change Line 40 to:

```
40  PRINT MID$(S$,1,N)
```

...and Run.

It PRINTed N characters, counting from number 1 on the left.


25 CHAPTERS and ALL YOU CAN DISPLAY IS: "KILROY WAS HERE"?
AW, WE WERE JUST FOOLIN' AROUND.

MID$ can also simulate RIGHT$. Change Line 40:

```
40  PRINT MID$(S$,16-N,N)
```

...and Run.

Would you believe RIGHT$ backwards, one at a time?

```
40  PRINT MID$(S$,16-N,1)
```

...and Run.

How about a sort of "histogram" type graph:

```
40  PRINT MID$(S$,N,N)
```

...and Run.

Make notes for future reference. If all these examples don't spark some ideas for your future use, I give up.

Suppose we want to PRINT the character in a specific position in the string. Make the program read:

```
10 S$ = "KILROY WAS HERE"
20 INPUT "CHARACTER # TO PRINT";N
30 PRINT MID$(S$,N,1)
```

...and Run.

If it's not obvious, we can assign any of these statements to a variable. That variable can in turn be used in tests against other variables. Change:

```
30 V$ = MID$(S$,N,1)
40 PRINT V$
```

...and Run.

A short book could be written about these three powerful functions, but I think the point's been made. They are used *very* frequently in complex sort and select routines. If we dissect them into these simple components, they are easy to keep track of. The next section has some good examples.

> **EXERCISE 25-1:** Write a program that asks the question, "ISN'T THIS A SMART COMPUTER." Input a YES or NO answer. If the first character in the answer is a Y, PRINT "AFFIRMATIVE." If the first character is an N, PRINT "NEGATIVE." Otherwise PRINT "THIS IS A YES OR NO QUESTION," and send control back to the INPUT statement.

> **EXERCISE 25-2:** READ in the following part numbers: N106WT, A208FM, AND Z154DX. Use MID$ to find the numbers. PRINT the number with the largest value.

Searching With INSTR

INSTR (pronounced, "In-string") is a function that can be of value when searching for a needle in a haystack. It compares one string against another to see if they have anything in common.

Suppose we have a list of names and want to see if another name (or part of that name) is in our list. It's the "part of" which makes this operation very different from a straight comparison of name-against-name, which we already know how to do using ordinary string-against-string comparisons. Here we learn how to locate a name (and similar names) by asking for just a small part of it.

Start the New program by entering this list of Names:

```
DATA SMITH, JONES, FAHRQUART, BROWN
DATA JOHNSON, SCHWARTZ, FINKELSTEIN
DATA BAILEY, SNOOPY, JOE BFTSPLK, *
```

That was the easy part.

How do we READ these names, one at a time, and compare them, or parts

of them, with the name or part of a name which we INPUT? Add these Lines:

```
10 INPUT "WHAT LETTER(S) IS WANTED";N$
20 PRINT
30 READ D$
40 IF D$ = "*" THEN GOTO 99
50 IF INSTR(1,D$,N$) = 0 THEN 30
60 PRINT ,N$;" IS PART OF ";D$
70 GOTO 30
99 PRINT : PRINT "END OF SEARCH" : END
```

Save this program As INSTR. We'll be needing it later.

Now this takes a bit of explaining:

Line 10 INPUTs the name or part of the name we are trying to locate.

Line 20 PRINTs a blank space for easier reading to help give this book some class.

Line 30 READs a single name from the DATA file.

Line 40 tests to see if D$ is READing the last item in the DATA file; IF so, execution branches to Line 99.

Line 50 uses the INSTRing to do all the searching. INSTR looks at D$, starting with the 1st character, to see if the characters INPUT in N$ match characters in D$. If INSTR returns the value of 0, it means there is no "match," and the program should READ the next piece of DATA. If there is a match, INSTR returns a number which is the number of characters it counted in N$ before a match was found. Since this number is not 0, execution drops to:

Line 60 which PRINTs both what we're looking for and the match.

Line 70 starts the process over again.

Run, trying various letters, names and parts of names to get the hang of what's going on. It's pretty impressive!

Now that wasn't too bad, was it? ('Twarnt nothin', really.) It doesn't matter how hard a program seems; when broken down to its individual parts, it isn't very hard. Like we've pointed out before, "The BASICs Are Everything." A little time beside the pool reflecting on the logic will do wonders.

For those with only a silver fingerbowl, but no pool, these changes will show the inner machinations of INSTR.

```
50 L = INSTR(1,D$,N$)
55 IF L = 0 THEN 30
60 PRINT ,N$;" IS CHARACTER#";L;"IN ";D$
```

Run it through a number of times trying different letters. It really does make sense!

To see the effect the starting number following INSTR has on our program, change Line 50 to:

```
50 L = INSTR(2,D$,N$)
```

INSTR now looks at D$, starting with the 2nd character.

Run and type in the letter S. See how it skipped SMITH, SCHWARTZ and SNOOPY? Play around with the starting number in INSTR until you have a good handle on what it does.

EXERCISE 25-3: ReLOAD the "INSTR" program, and change the DATA Lines to:

```
DATA P-RUTH, OF-MANTLE, SB-MORGAN
DATA SS-LEOTHELIP, P-KOUFAX
DATA C-CAMPANELLA, P-FELLER,*
```

What string would we enter to LIST the pitchers only?

A. P
B. PITCHER
C. P-
D. None of the above

Save As BASEBALL and Run. Practice sorting by team positions.

Snarled STRING

In the last Chapter we learned about STR$, which lets us convert a numeric variable to a string variable. For the purpose of confusion (no doubt), there is another "string-string" that does something completely different. Fortunately, it is written differently.

STRING$(N,A) is a specialized PRINT *modifier* which allows us to PRINT a single ASCII character, represented by A, a total of N times. Quite simple, really, and very useful.

Select New and type:

```
10 PRINT STRING$(23,42);
20 PRINT "STRING$ FUNCTION";
30 PRINT STRING$(23,42)
```

...and Run.

Wow! That really moves. It PRINTed ASCII character 42, which is a *, 23 times, then PRINTed the phrase STRING$ FUNCTION, then PRINTed * 23 more times. This just has to have some good applications.

Suppose we need to type a "header" across the top of a report -- let's say the first line of it is to be solid dashes. What is the ASCII code for a dash? Forgot? *Me too*. Everybody back to Appendix A to find the code number.

45 it is. We want to PRINT, 70 times, the character represented by ASCII code 45. That will print dashes across the full width of our screen. The New program should look something like:

```
20 PRINT STRING$(70,45)
```

...Run it.

An even easier way to use STRING$ is to replace the ASCII code of the character we wish to PRINT with the actual character itself. (It must be enclosed in quotes.) This works fine with characters that really PRINT, such as letters, numbers and punctuation marks. Change Line 20 so the program reads:

```
20 PRINT STRING$(70,"-")
```

...and Run.

Works nice, doesn't it, and we didn't have to look up the ASCII code.

We can bring in a single string character via a string variable. This simple New program shows a variation on the theme and may trigger some ideas:

```
10 INPUT "ANY LETTER, NUMBER OR SYMBOL";A$
20 PRINT STRING$(70,A$)
30 PRINT : GOTO 10
```

Play around with STRING$ a while. It's really very helpful when needed, particularly for giving display PRINTouts some class. An obvious advantage is its ability to do a lot of PRINTing with very little programming.

EXERCISE 25-4: Print a string of 30 asterisks centered at the top of the screen.

SPACE$ And SPC

The SPACE$ allows us to print from 0 to 255 proportionally-spaced blank spaces. For example:

```
PRINT "A";SPACE$(20);"B"
```

will print A and B with 20 spaces between them.

SPC is almost the same function as SPACE$, but it doesn't use proportional spacing. Example:

```
PRINT "A";SPC(20);"B"
```

prints 20 non-proportional blank spaces between A and B.

On The Lighter Side

The specialized string functions enable us to do all sorts of exotic things. Here is the beginning of a simple but fun New program which uses LEN and MID$. You can easily figure it out, especially after you've seen it Run.

Enter:

```
10 REM  * TIMES SQUARE BILLBOARD *
20 CLS : N=0 : PRINT : READ A$
30 L=LEN(A$) : F=1
40  IF L>N THEN L=N+2
50 B$ = MID$(A$,F,L)
60 PRINT TAB(64-N);B$
70 FOR T=1 TO 200 : NEXT T
80  IF N=63 GOTO 100
90 N=N+1 : IF N<63 GOTO 120
100 L=L-1 : F=F+1 : IF L<0 THEN L=0
110  IF L=30 GOTO 20
120 CLS : GOTO 40
```

```
500 DATA "LUCKY LINDY HAS LANDED IN PARIS ..."
510 DATA "... MET BY CROWD AT LEBOURGET AIRPORT"
```

...and Run.

Your assignment, if you choose to accept it, is to complete the program so it repeats, ends, or otherwise does not crash.
Good luck!.

.....Fssssss!

Learned In Chapter 25

Functions

LEFT$
MID$
RIGHT$
INSTR
STRING$
SPACE$
SPC

Miscellaneous

INSTRing routine

Chapter 26

TIME$ And DATE$

How about a short and simple Chapter?

Wouldn't it be nice to be able to use time and date information in a BASIC program? We can, and it's as easy as A$, B$, C$,...

All we have to do is enter the Command window and type:

```
PRINT TIME$          Return
```

and TIME is displayed.

The DATE can also be displayed by typing:

```
PRINT DATE$          Return
```

Setting The Clock And Calendar From BASIC

The DATE is set by typing:

```
DATE$ = "12-12-85"          Return
```

or

```
DATE$ = "12/12/85"          Return
```

The Computer places the date into the Operating System. Verify it by typing:

```
PRINT DATE$          Return
12-12-1985
```


AT THE TONE...

To set the time from BASIC, type:

```
TIME$ = "19:06:30"          Return
```

Type:

```
PRINT TIME$          Return
```

to verify. Depending on how fast a typist you are, several seconds will have elapsed.

All of the string operators we learned about in the previous Chapters can be used to manipulate these two strings. For example, to PRINT only the day and month from DATE$, return to the List window and type:

```
10 DAY$ = MID$(DATE$,4,2)
20 MONTH$ = LEFT$(DATE$,2)
30 PRINT "THIS IS DAY #";DAY$;
40 PRINT " IN MONTH #";MONTH$
```

Save As DATE ... and Run.

Note carefully that DATE$ and TIME$ are built into the Macintosh, but DAY$ and MONTH$ are simply string variables we created.

Type in this New program:

```
10 PRINT DATE$, TIME$
20 GOTO 10
```

...and Run.

How's that for cheap and dirty? There are an endless number of much more sophisticated ways to display time and date. Any ideas?

EXERCISE 26-1: Write a program which continuously displays the time and date neatly on the screen.

Keyboard Buffer

You may have noticed that the Computer seems to remember what we have typed on the keyboard even when it is busy performing some other task.

An area in memory is set aside to be a Keyboard Buffer. That buffer stores our keystrokes until the Computer is ready to accept them. We can easily "type ahead" of the Computer while it is busy performing such tasks as reading the FILES, printing information on the screen or printer, performing large calculations, executing FOR-NEXT loops, etc.

The Keyboard Buffer can store up to 31 key strokes.

> When the buffer is overloaded, it will signal you. If the sound has not been turned off, Macintosh will BEEP. If it has been turned off, the Menu bar will flash.

Enter this delay loop program.

```
10 PRINT "TYPE CHARACTERS UNTIL I BEEP."
20 FOR N = 1 TO 30000 : NEXT
30 INPUT "PRESS [Return]";A$
40 PRINT A$
```

As soon as you Run the program, type any group of letters or numbers until the Computer beeps and wait. When program execution is finished, the keystrokes are displayed. Press Return to satisfy the INPUT statement.

For fast typists, this is a real time-saver.

Learned In Chapter 26

Statements	Miscellaneous
TIME$	Keyboard buffer
DATE$	

PART 4

VARIABLE PRECISION AND MATH

Chapter 27

What Price Precision?

The two versions of Microsoft BASIC, BASIC(b) and BASIC(d), store and display numbers with different accuracy. When BASIC(b) is selected, 7 digits are displayed, though only 6 will be accurate. This is called "single precision" accuracy and is more than adequate for most applications.

The old slide rule was accurate to only 3 digits.

For large business or special scientific applications, however, greater accuracy is needed. With Microsoft BASIC, we have a capability called "double precision." When BASIC(d) is selected, the Computer stores numbers accurate to 14 digits and PRINTs them out accurate to 13. However, we pay a price for this precision both in the additional memory it takes to store and process long numbers and in the extra time required to process them.

We could use either version of BASIC to learn about single and double precision numbers since both versions can convert numbers to either precision. However, since up to this point, we have been using BASIC(b) to write our programs, it's best to continue using it. Programs written in one version of BASIC cannot be loaded into memory when using the other version.

If you're not in BASIC(b) or are not sure which version of BASIC is loaded in the Computer, select Quit from the File menu and return to the Finder. Now, double-click the BASIC(b) icon. When BASIC(b) is loaded, enter this program:

```
10 X = 12345678909876543 21          (Check 'em.)
20 Y = .000000000123456789           (Check 'em.)
30 Z = X * Y
```

```
40 PRINT X;"TIMES";Y
50 PRINT "EQUALS";Z
```

Note that the number values in Lines 10 and 20 have been converted to Exponential Double Precision. That's what the "D's" in those Lines stand for.

Now Run.

```
 1.234568E+18 TIMES 1.234568E-10
EQUALS 1.524158E+08
```

Ummm-hmmm. A very large number times a very small number and the answer -- all expressed in Exponential notation. That's what the "E's" stand for, and each number has been clipped to 7 significant digits. (The 'E' designates *Exponential notation.* E+18 means the number before it times 10 to the +18th power. E−10 means the number ahead of it times 10 to the −10th power.)

Double Precision

We can easily convert storage, processing and printing of X, Y and Z to double precision. The BASIC Statement is an easy one:

```
5 DEFDBL A-Z
```

DEFDBL stands for "DEFine as DouBLe precision," and A-Z means "every variable from A through Z."

Insert the Line and RUN.

```
1.23456789098765D+18 TIMES 1.23456789D-10
EQUALS 152415787.6238378
```

Quite a difference, eh? Those lost significant digits in the answer came back from the hinterland and expanded our printout from 7 places to 16.

Such precision is usually wasteful of memory space and time except in short programs; but fortunately only a few variables ever need to be so precise.

Since we are only using 3 variables, X, Y, and Z, there is really no point in DEFining more than them to double precision. We can tell the Computer to handle only those as double precision and leave any other variables (of which there are none, right now) alone. Change Line 5 to:

```
5 DEFDBL X-Z
```

...and Run.

Same results.

Overruled!

There are times when we will want to *temporarily* override the DEFDBL declaration, converting a number or answer back to single precision. Suppose we want Z to be printed as single precision. We can override the Line 5 declaration by changing only those Lines which contain Z. Do it:

```
30 Z! = X * Y
50 PRINT "EQUALS";Z!
```

...and Run.

```
 1.23456789098765D+18 TIMES 1.23456789D-10
EQUALS 1.524158E+08
```

Our "raw" data and the calculating was done in double precision, but our final answer is printed out with only single-precision accuracy -- just what we asked for. A *specific* declaration (like the ! which stands for "single precision") always takes precedence over a *global* declaration like Line 5. (Global means "valid for the entire program," not just one character or one Line.)

DouBLe Precision -- Simplified

There's another way to calculate with high (double) precision but print the answer in single precision. Since single precision is the "default" mode, we

can simply *not* include Z in Line 5.

Change Lines 5, 30 and 50 and Run.

```
5 DEFDBL X,Y          (or DEFDBL X-Y)
30 Z = X * Y
50 PRINT "EQUALS";Z
```

Same results.

Global Override

It is possible to override the "global" DEFDBL declaration with a global single precision declaration. DEFSNG will change everything back to single precision. Let's try it by adding these Lines:

```
60 DEFSNG X-Y
70 PRINT X;"TIMES";Y;"EQUALS";Z
```

...and Run.

Good Grief -- our "single-precision" numbers turned to zeros, but the Z answer is correct!

Well, it turns out that X *DouBLe precision* is a completely separate variable from X *SiNGle precision*. It's as different from X as is Y, or any other variable. If we want to use X and Y again as single-precision numbers, we have to go back and assign them values *after* declaring them to be single precision. Hmmmm. This is getting complicated.

A cheap and dirty way to show the point is to change Line 70 to:

```
70 GOTO 10
```

...and Run -- choosing Stop or hitting the ⌘ . keys after both double and single precision versions are printed in Lines 40 and 50.

Line 60 reDEFines X and Y as single precision, then control returns to Line

10, and the calculations are performed again. (Fortunately, there is rarely reason to *reDEFine* a variable within a program. If necessary, we can do it with conventional string techniques.)

DouBLe Precision, Another Way

Instead of a "global" declaration of accuracy, we can do it one variable at a time. Change the resident program to read:

```
10 X# = 12345678909876543 21
20 Y# = .000000000123456789
30 Z# = X# * Y#
40 PRINT X#;"TIMES";Y#
50 PRINT "EQUALS";Z#
```

...and Run.

Same results as before. The # sign declares that the variable letter preceding it is to be handled as DouBLe precision, overriding the normal presumption that it is SiNGle precision.

Remember, X# is not the same as X. It is an entirely different variable. Same with Y# and Z#. To nail this point down, add:

```
5 X = 4.321
60 PRINT "X =";X
```

...and Run.

The values of X and X# had no effect on each other, did they?

INTeger Precision

In those frequent cases where the numbers used are integers (and in the range between -32768 and +32767), execution can be speeded up by declaring them to be INTegers with the % sign or the DEFINT statement. Type this NEW program:

```
20 PRINT "START"
```

```
30 FOR N = 1 TO 22000
40 NEXT N
50 PRINT "STOP"
```

Using a stopwatch or clock with a second hand, measure the time it takes for the 22000 passes thru the FOR-NEXT Loop ... and Run.

Should be around 10 seconds. By default, the Macintosh processed the values of N in single precision.

Now, let's declare N to be an INTeger (which is all the accuracy we need), and time it again. Insert:

```
10 DEFINT N
```

...and Run.

Aha! It took only about 5 seconds. Cut the processing time in half.

We can accomplish the same thing using specific declarations instead of the global DEFINT. Delete Line 10, and change the program to read:

```
30 FOR N% = 1 TO 10000
40 NEXT N%
```

...and Run.

Same fast results.

One More Way

The conversion functions CSNG(#), CDBL(#) and CINT(#) provide 3 additional ways to declare numbers as SiNGle, DouBLe or INTeger precision. Enter this NEW test program:

```
10 X = 12345.6789
20 PRINT X
30 PRINT CSNG(X)
```

```
40 PRINT CDBL(X)
50 PRINT CINT(X)
```

...and Run.

It tells the whole sordid story:

```
12345.68
12345.68
12345.6787109375
12346
```

Line 10 changes to `10 X = 12345.6789#` indicating the number was so long that it could not be held in single precision.

Line 20 PRINTed the value of X accurate to 7 digits.

Line 30 PRINTed the SiNGle precision value of X -- the same value as PRINTed by Line 20.

Line 40 PRINTed the DouBLe precision value of X, but it sure isn't a duplicate of what we specified as X in Line 10! The problem is, we only input the number in single precision (by default). PRINTing it out in double precision requires the Computer to just "make up" numbers to fill out the places.

Don't try to be more accurate than what you begin with. It's the programmer who's supposed to be creative, not the Computer!

Line 50 PRINTed the INTeger value of X. This works slightly different than INT(X). CINT(X) "rounds off" the fractional part.

Let's make the value of X negative and see what happens. Change Line 10 to:

```
10 X = -12345.6789#
```

...and Run.

No surprises. CINT acted just like INT does, rounding downward to arrive at -12346.

DouBLe The Trouble -- DouBLe The Fun

Now let's go back and declare the value of X to be DouBLe precision, change it to a positive number and do all our PRINTing in DouBLe precision. The edited program will read:

```
10 X# = 12345.6789
20 PRINT X#
30 PRINT CSNG(X#)
40 PRINT CDBL(X#)
50 PRINT CINT(X#)
```

...then Run,

and the display reads:

```
12345.6789
12345.68
12345.6789
12346
```

All makes sense, and all quite predictable, isn't it?

Caveat

Degrees of precision may not be the most inspiring subject, nor always seem to be the most consistent. But, if we're at least aware of the differences in precision, we'll not be caught off guard and be deceived by numbers that never were.

Learned In Chapter 27

Statements	Functions	Miscellaneous
DEFDBL	CDBL	Double precision (#)
DEFSNG	CSNG	Single precision (!)
DEFINT	CINT	Integer precision (%)

Chapter 28

Intrinsic Math Functions

The BASIC language includes a number of **mathematical** functions. These math functions are all very straightforward and easy to use, but if your math skills are a bit rusty, you will want to refresh them to fully understand what we're doing. We'll keep everything here at the 9th-grade Algebra level so there's no need to panic (unless maybe you're in the 6th grade ... but even so, just hang on and you'll be OK).

INT(N)

We have studied the INTeger function in some detail in earlier Chapters so we won't cover that ground again. INT stores and executes numbers in single precision.

FIX(N)

FIX is just like INT, but instead of rounding negative numbers downward, it simply chops off everything to the right of the decimal point.

Try this simple test in the Command window:

```
PRINT INT(-12345.67)
```

produces -12346.

```
PRINT FIX(-12345.67)
```

produces -12345.

The one we use depends on what we want.

SQR(N)

The SQuare Root function is simple to use.

Type this New program in the List window:

```
10 INPUT "THE SQUARE ROOT OF";N
20 PRINT "IS";SQR(N)
30 PRINT : GOTO 10
```

...and Run some familiar numbers.

Another way to find the square (or any) root of a number is by using the ^ (caret). The caret is produced by pressing the Shift and 6 keys at the same time. It means "raised to the power." Finding the square root of a number is the same as raising it to the 1/2 power. Change Line 20 to:

```
20 PRINT "IS ";N^(1/2)
```

...and Run some familiar numbers.

The same logic which allows us to find the *square* root with the ^ will let us find any *other* root. (Even the thought of doing that in pre-computer days drove men mad.) Out of the sheer arrogance of power, let's find the 21st root of any number. Change the first two Lines:

```
10 INPUT "THE TWENTY-FIRST ROOT OF";N
20 PRINT "IS ";N^(1/21)
```

...and Run.

Now there is real horsepower! Problem is, how can we be sure that the answers are right? Well, it's easy enough to add a few Lines that will take the root and raise it back to the 21st power to find out. Let's change the program to make it read:

```
10 INPUT "THE TWENTY-FIRST ROOT OF";N
20 R = N^(1/21)
```

```
30 PRINT "IS";R
40 PRINT
50 PRINT R;"TO THE 21ST POWER =";R^21
60 PRINT : GOTO 10
```

...and Run.

The INPUT and output numbers check out pretty close, don't they? This "proof" process might not stand up under rigorous scrutiny, but the answers are correct.

EXERCISE 28-1: Pythagoras discovered that the sides of a right triangle always obey the rule:

$$C^2 = A^2 + B^2$$

where C is the longest side (hypotenuse). Stated another way: "The length of side C equals the square root of the sum of the squares of sides A and B ($C = \sqrt{A^2 + B^2}$)."

If side A = 5 and side B = 12, write a program to calculate the length of side C.

ABS(N)

ABSolute value has a lot to do with signs, or without them. The absolute value of any number is the number *without* a sign. If you've forgotten, this program will quickly refresh your memory:

```
10 INPUT "ENTER ANY NUMBER";N
20 A = ABS(N)
30 PRINT A
40 PRINT : GOTO 10
```

...and Run.

Respond with various large and small, positive and negative numbers, and zero.

They all come out as they went in, didn't they, except the sign is missing?

MOD

No, not the Music. MOD isn't really a math Function; it's more of a Math Operator. MOD returns the remainder when one number is divided into another number. For example:

```
PRINT 17 MOD 4
```

returns a 1 since 17/4 is 4 with a remainder of 1.

Other examples to try:

8 or 16 or 24 MOD 8 each equals 0. (There's 0 remainder when any of them are divided by 8.)

9 or 17 or 25 MOD 8 each equals 1. (There's 1 remainder after any of them are divided by 8.)

10 or 18 or 26 MOD 8 each equals 2. (There's 2 remainder after any of them are divided by 8.)

15 or 23 or 31 MOD 8 each equals 7. (There's 7 remainder after any of them are divided by 8.)

LOG(N)

No, a LOG isn't what they build cabins with, but even the swiftest among us have to refresh our memory from time to time to keep the details straight.

A LOG (logarithm) is an *exponent*. Exponent of what? The exponent of a *base*. What's a *base?* A *base* is the number that a given number *system* is built on. Aren't all number systems built on 10? 'Fraid not.

$10^3 = 1000$

10 is the BASE.

3 is the LOG(exponent), and

1000 is the answer.

Think it has something to do with "new math," but I was too old to take it, too young to teach it, and grateful for not learning it from those who didn't understand it.

As if life weren't complicated enough, the LOGarithm system is centered around what are called *natural* logs. Exactly what that means is the subject of another discussion, but we're stuck with it anyway. Natural logs use the number 2.718282 as their base. (Really makes your day, doesn't it!) Some BASIC interpreters provide a second LOG option using 10 as the base, as in our decimal system, but making the conversion isn't too bad -- and we do have to live with it.

Type this New program:

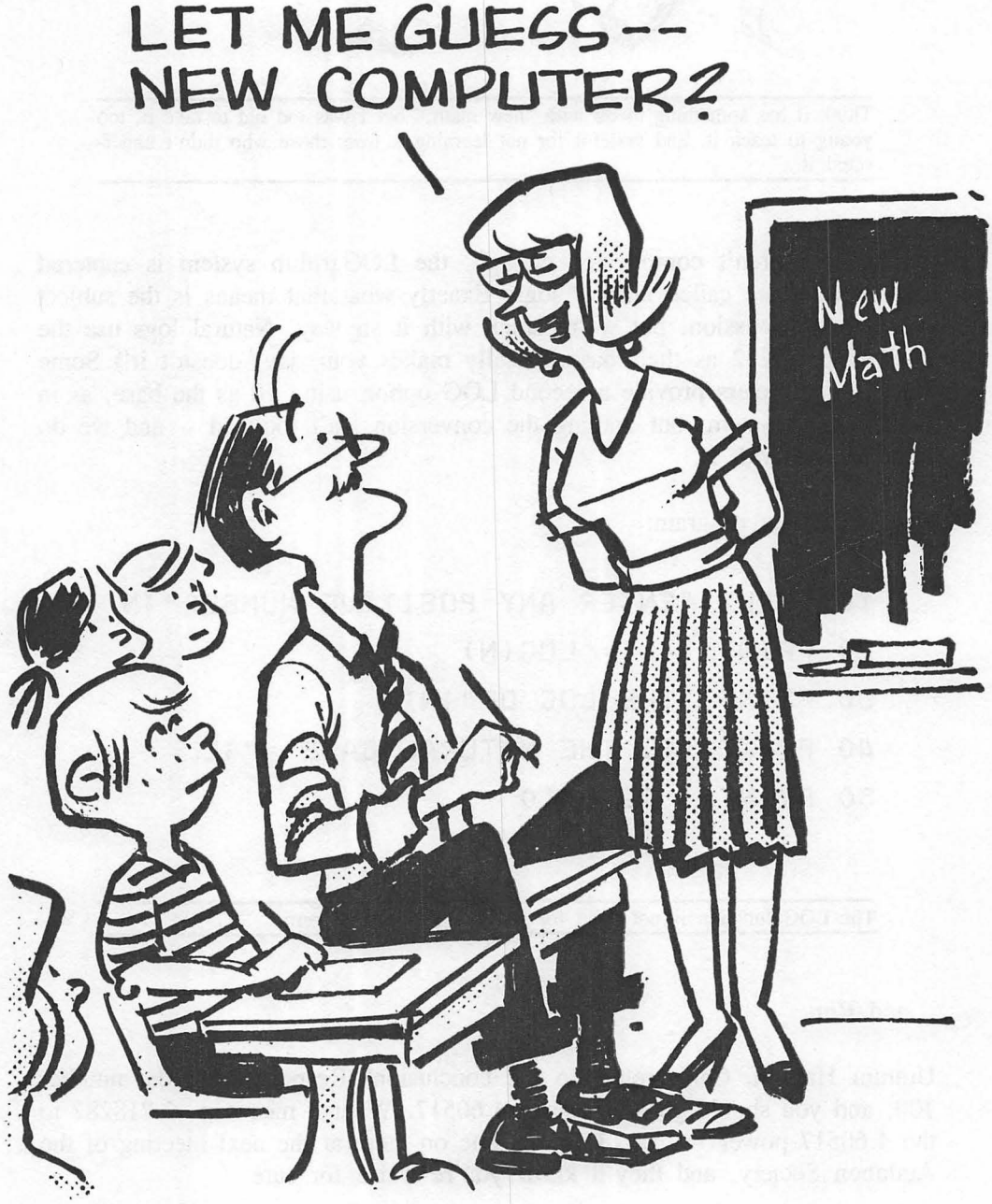
```
10 INPUT "ENTER ANY POSITIVE NUMBER";N
20 PRINT : L = LOG(N)
30 PRINT "THE LOG OF";N;
40 PRINT "TO THE NATURAL BASE =";L
50 PRINT : GOTO 10
```

The LOG function is not valid for negative numbers or zero.

...and Run.

Ummm Hmmm. Can't relate to the conclusion? Respond with the number 100, and you should get the answer 4.60517. What it means is, 2.718282 to the 4.60517 power = 100. Lay that one on them at the next meeting of the Audubon Society, and they'll know you're weird for sure.

Let's jack this thing around to where the vast majority of us who have to work with LOGs can use it ... into the decimal system.


LET ME GUESS--
NEW COMPUTER?
New
Math

Decimal-based LOGs are called "common," or "base 10," Logs. Insert these Lines:

```
45 PRINT "THE LOG OF";N;
47 PRINT "TO THE BASE 10 =";L*.4342945
```

...and Run, using 100 as the number.

Ahhh! That's more like it. We can all see that 10 to the 2nd power equals 100. It's good to be back on *relatively* solid ground.

The magic conversion rules are:

> *To convert a natural log to a common log, multiply the natural log by .4342945.*
>
> *To convert a common log to a natural log, multiply the common log by 2.3026.*

And that's the name of that tune.

This final New program scoops it up and spreads it out:

```
10 REM  * LOGARITHM DEMO *
20 INPUT "ENTER A POSITIVE NUMBER";N
30 PRINT
40 PRINT "THE NUMBER","NATURAL LOG",
50 PRINT "COMMON LOG"
60 PRINT N,LOG(N),LOG(N)*.4342945
70 PRINT : GOTO 20
```

Wring it out until you're comfortable with the concept.

EXP(N)

EXP is sort of the opposite of LOG. EXP computes the value of the answer, given the EXPonent of a *natural* log. (Another winner.)

2.718282 raised to the EXP power = the answer.

Type in this New program:

```
10 INPUT "ENTER A NUMBER";N
20 A = EXP(N)
30 PRINT "2.718282 RAISED TO THE";N;
40 PRINT "POWER =";A
50 PRINT : GOTO 10
```

...and Run.

We're entering the EXPonent now, so it's easy to INPUT a number that is too big for the Computer and will cause it to *overflow*.

As a benchmark against which to test the program, enter this number:

```
4.6051702
```

The BASE of the natural log system raised to this power should equal 100 (or something very close).

Being this far into logs, you can create your own advanced test programs, and check the results against a LOG table. *And if you're not too comfortable with all this ... try making a log cabin with the remainders!*

EXERCISE 28-2: (For math fans only) Convince yourself that LOG and EXP functions are inverses of each other (hint: LOG(EXP(N)) = N). Try putting the two functions together in the opposite order using both positive and negative values for N. Why do the negative values create havoc?

Learned In Chapter 28

Functions	Math Operators	Miscellaneous
INT	^ (caret)	Natural Logs
FIX		Common Logs
SQR		
ABS		
MOD		
LOG		
EXP		

Chapter 29

The Trigonometric Functions

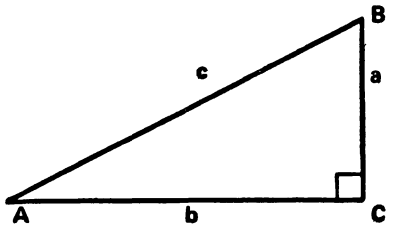
Since this is about as deep as we'll get into mathematics, I have to assume you know something about elementary trig.

Trigonometry, of course, deals with triangles, their angles, and the ratios between the lengths of their sides. In the triangle below, the Sine (abbreviated SIN) of angle A is defined as the *ratio* (what we get after dividing) of the *length* of side a to the *length* of side c. COSine and TANgent are defined similarly:

SIN A=a/c

COS A=b/c

TAN A=a/b



From these relationships, we can find any ratio if we know the corresponding angle. Let's try this simple New program:

```
10 INPUT "ENTER AN ANGLE (0-90 DEGREES)";A
20 S = SIN(A*.0174533)
30 PRINT "THE SIN OF A";A;"DEGREE ANGLE IS";S
40 PRINT : GOTO 10
```

...and Run.

It really works! Try the old "standard" angles like 45°, 30°, 60°, 90°, 0°, etc.

Unless you're right up to snuff on trig, Line 20 undoubtedly looks strange. Well, it turns out that most computers think in radians, not degrees (always has to be some nasty twist, doesn't there...!). A radian is a unit of measurement equal to approximately 57 degrees. In order to convert from degrees

(which most of us use) to radians, we changed the INPUT from degrees to radians. The SIN function will not work correctly without this conversion.

To convert angles from degrees to radians, multiply the degrees by 0.0174533.

To convert angles from radians to degrees, multiply the radians by 57.29578.

Failure to make these conversions correctly is by far the biggest source of computer users' problems with the trig functions.

COSine and TANgent work the same way. Change the resident program to:

```
10 INPUT "ENTER AN ANGLE (0-90 DEGREES)";A
20 C = COS(A*.0174533)
30 PRINT "THE COS OF A";A;"DEGREE ANGLE IS";C
40 PRINT : GOTO 10
```

...and Run.

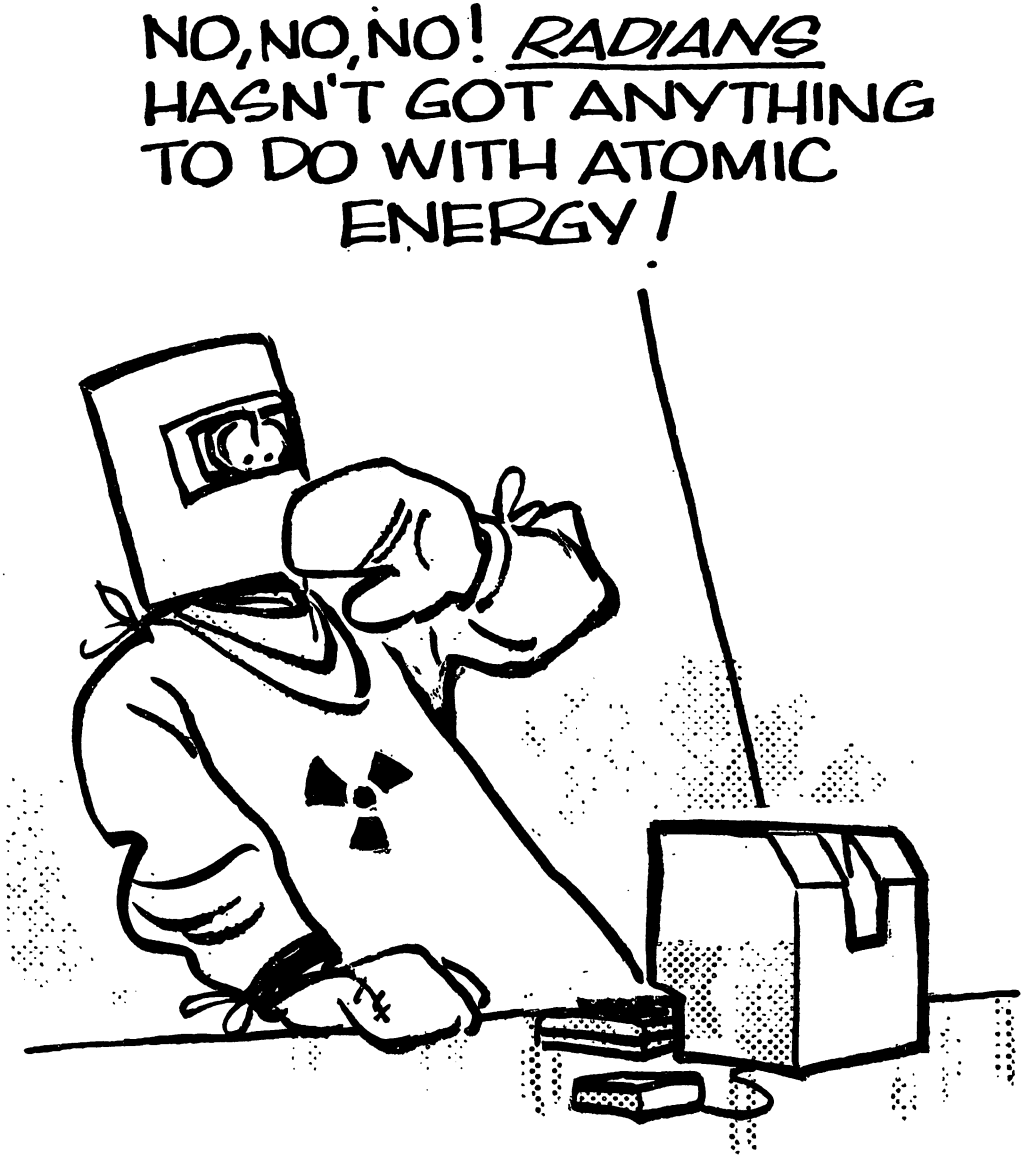
We know that COS(90°) should be 0. Unfortunately, the Computer is slightly off because it calculates these functions by approximation. It's doing the best that it can ... honest!

For TANgent, Run this program:

```
10 INPUT "ENTER AN ANGLE (0-90 DEGREES)";A
20 T = TAN(A*.0174533)
30 PRINT "TAN OF A";A;"DEGREE ANGLE IS";T
40 PRINT : GOTO 10
```

The TAN function is not even defined for 90°, though Microsoft BASIC will *try* to calculate it.

This next New program displays all 3 major trig functions at the same time.


NO, NO, NO! RADIANS HASN'T GOT ANYTHING TO DO WITH ATOMIC ENERGY!

Note that in Line 20 we *divide* our incoming angle by 57.29578 instead of multiplying it by 0.0174533. The results are the same.

```
10 INPUT "ENTER AN ANGLE (0-90 DEGREES)";A
20 A = A/57.29578 : PRINT
30 PRINT "SIN =";SIN(A)
40 PRINT "COS =";COS(A)
50 PRINT "TAN =";TAN(A)
```

Inverse Trig Functions

The opposite of finding a *ratio* between two sides of a triangle when an *angle* is known, is finding an *angle* when the *ratio* of two sides is known. There are 3 trig functions available to do it, but most computers only make provisions for one, called ATN (Arc of the TaNgent).

The following simple program takes the angle we INPUT, converts it to radians, computes and PRINTs its TANgent. Then, as a "proof check," it takes that TANgent value and reverses the process by computing its arc (angle). The letter "I" is used in the program since the arctangent is also known as the "Inverse" (sort of the "opposite") of the TANgent.

```
10 REM  * ATN DEMO *
20 INPUT "ENTER AN ANGLE (0-90 DEGREES)";A
30 T = TAN(A/57.29578) : PRINT
40 PRINT "TANGENT =";T
50 I = ATN(T) * 57.29578
60 PRINT "ARC OF THE TANGENT =";I
```

If you're one of those rare types who is very familiar with trig, you can probably throw numbers around in such a fashion that the other 2 "inverse" trig functions, ARCSIN and ARCCOS, are not needed. But for those of us who get confused when we run out of fingers, the last 2 functions are built into this simple New program by way of special routines. The accuracy is close enough for "government" work. Give it a try:

```
10 REM  * INVERSE FUNCTION ROUTINES DEMO *
```

```
20 INPUT "ENTER THE RATIO OF 2 SIDES";R
30 CLS : PRINT
40 ARS=2*ATN(R/(1+SQR(ABS(1-R*R)))) * 57.2958
50 AC=90 - ARS : PRINT
60 PRINT "RATIO","ARCSIN","ARCCOS","ARCTAN"
70 PRINT "(NUMBER)","(DEGREES)","(DEGREES)",
80 PRINT "(DEGREES)" : IF ABS(R)>1 THEN 110
90 PRINT R,ARS,AC,ATN(R)*57.2958
100 PRINT : GOTO 20
110 PRINT R,"U","U",ATN(R)*57.2958
120 PRINT : GOTO 20
```

Remember, while the TANgent can be any number, when our ratio moves outside the range of −1 to 1, SIN and COS are both mathematically "Undefined." Also, ARCTAN and ARCSIN produce angle measures between −90 and 90 degrees, but ARCCOS has a range between 0 and 180 degrees.

Learned In Chapter 29

Functions	Miscellaneous
SIN	Degrees
COS	Radians
TAN	
ATN	

Chapter 30

DEFined FuNctions

This Chapter is for advanced math types. If that isn't your bag, skim it lightly, and move on down the road.

In addition to the *intrinsic* (built-in) Functions, Microsoft BASIC allows us to define *our own* Functions.

In what kind of situation would we want to do that? Repetition of formulas and simple operations that are used repeatedly can be greatly shortened by building a custom Function. They won't operate as fast as the other, factory built-in Functions, but like subroutines, they greatly simplify BASIC programming.

The Format for defining a Function is:

```
DEF FN name(v1,v2,...) = formula
```

where:

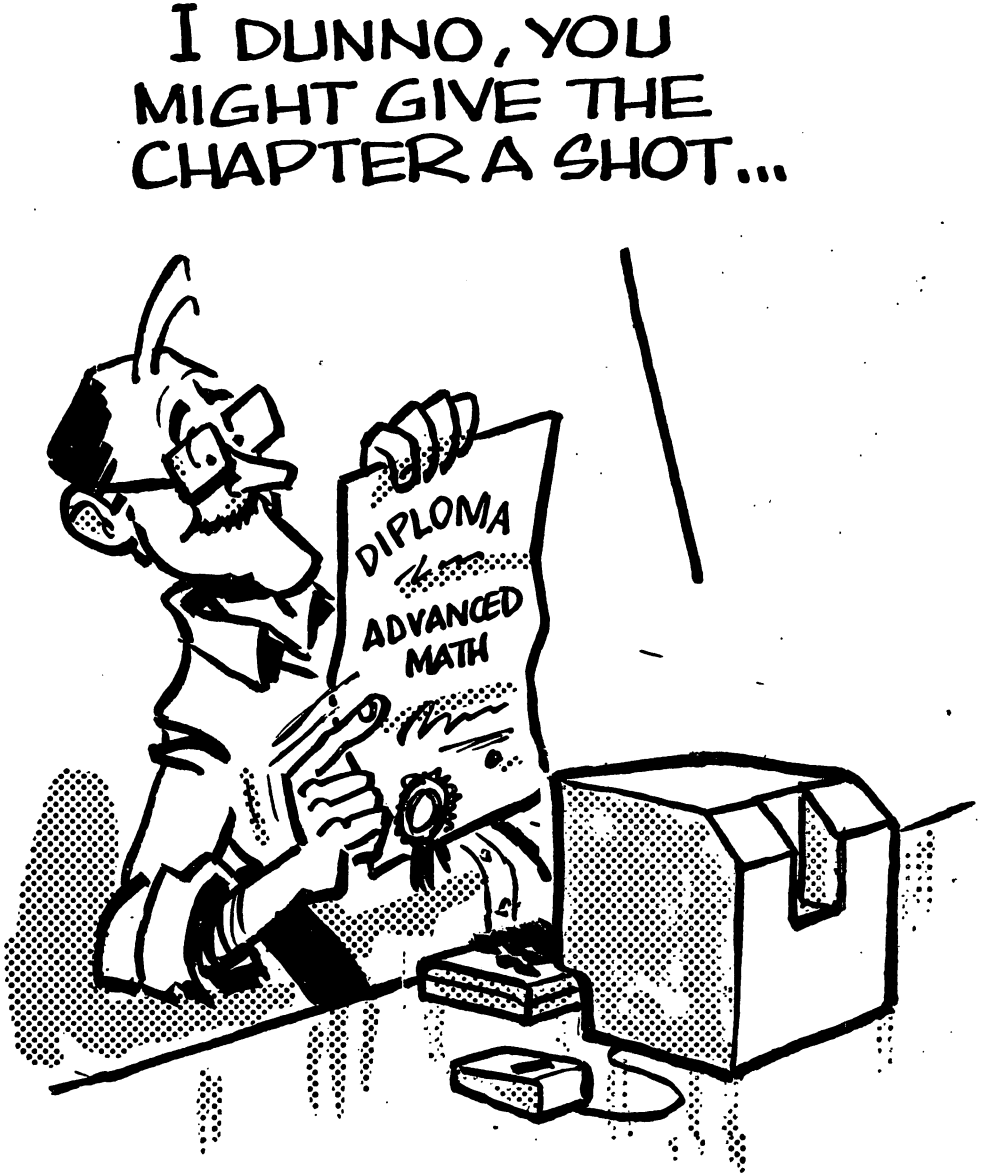
name is the Function name, and

v1, v2, ... are dummy variables that represent the values the Function will act on. *Name* and *v1, v2 ...* can be any valid variable names.

formula is the expression where the calculations are carried out.

Let's create a Function to do MODular arithmetic. MOD is one of our math operators, but we'll use it to demonstrate the technique DEFined FuNction. Try this on for size:

```
10 INPUT "ENTER X";X
20 INPUT "ENTER Y";Y
```


I DUNNO, YOU MIGHT GIVE THE CHAPTER A SHOT...
DIPLOMA
ADVANCED MATH

```
30 DEF FNC(X,Y) = INT(X-Y * INT(X/Y))
40 REM C = FUNCTION NAME/X,Y = THE NUMBERS
50 PRINT X;"MOD";Y;"=";FNC(X,Y)
```

...and Run.

The variables X and Y used in defining the Function in Line 30 are really "dummy arguments." They only show the Computer *how* to perform the calculations. Change Line 30 to:

```
30 DEF FNC(A,B) = INT(A-B * INT(A/B))
```

...and Run.

Same results? You betcha. In fact, we could even use A and B elsewhere in the program; Line 30 won't effect their values at all.

The FuNction variable can be INTeger, Single, or Double precision or even a string variable. The value returned to the program is determined by the type of variable. Try this NEW sample with a string:

```
10 DEF FNZ$(A$) = "-" + A$ + "-"
20 PRINT FNZ$("FUNCTION")
```

...and Run.

Functions are very powerful when used for repetitive calculations. How about the distance between two coordinate points in a plane, (X1,Y1) and (X2,Y2). Use:

```
10 DEF FND#(X1,Y1,X2,Y2) = SQR((X1-X2)*(X1-X2)
   + (Y1-Y2)*(Y1-Y2))
20 PRINT "DISTANCE IS"; FND#(-1,3,2,7)
```

Note: Line 10 is shown on two lines to fit the book. The Computer displays it on one line.

...and Run.

NOTE that D# is a double precision variable.

And it is even possible to come up with a Function that uses no variables at all:

```
10 DEF FNA = 1 + INT(RND*5)
20 PRINT FNA
```

Learned In Chapter 30

Statement

DEF FN

PART 5

DISPLAY FORMATTING

Chapter 31

Video Display Graphics

And It Draws Pictures Too!

Our Macintosh can draw an endless variety of pictures on the Video Display. We will learn some of the basic procedures and capabilities in this Chapter. After that, what you create is limited only by your own imagination. Who knows...you may write a graphic program artistically equivalent to the Mona Lisa.

Now, the 2 most basic of the 4 graphic commands:

PSET turns on (darkens) a particular section or point on the screen.

PRESET turns off (lightens) a particular point.

For graphic use, the screen is divided into a large number of sections or "pixels." Each pixel is a rectangular block, and each pixel has its own "address."

The letter "O" occupies a space that is 9 pixels high by 6 pixels wide.

For example:

```
PSET (55,32)
```

means -- "turn on the light" at the junction of 55th "H" Street and 32nd "V" Avenue.

H is the horizontal address, counting across from the left-hand side of the screen. V is the vertical address, counting down from the top of the screen. All "street addresses" start counting from the upper left-hand corner. H and V as used here are the same as X and Y used in the first quadrant of

mathematical coordinate grid systems. H and V are more descriptive and easier to work with while learning.

Type in:

```
10 PSET(55,32)
```

...and RUN.

Look carefully for the dot because it is very tiny (there are approximately 74 dots per inch).

Careful now, don't mess up the screen. Open the Command window and type:

```
PRESET(55,32)
```

How about that? We found the ON-OFF switch!

Want to really press you luck? Try darkening two pixels. That's right; add this Line:

```
20 PSET(55,33)
```

...and RUN.

We now have 2 pixels light, one on top of the other. Let's turn the upper pixel off by RUNning our program with this additional Line:

```
30 PRESET(55,32)
```

The point of all this obviously is that we can control whether each pixel on the screen is dark or light (on or off) by "talking" to it at its individual address with PSET and PRESET statements.

Flying Saucers Or Lightning Bugs?

If one has an ON-OFF switch, what does one do with it? With a little imagination, we could create pixels that go ON and OFF, to attract attention...

by blinking. This simple program shows how to set up a "blinker." Run it:

```
10 H = 100
20 V = 100
30 PSET(H,V)
40 PRESET(H,V)
50 GOTO 30
```

Simple FOR-NEXT loops at 35 and 45 could be used to control the blinking rate.

Once Again, More Heavily

In the Horizontal direction, there are 32768 addresses, numbered 0 to 32767. 0 is at the far left, 245 is near the middle, 490 is at the far right of the visible screen, and 32767 is at the extreme right, off the screen.

In the Vertical direction, there are also 32768 pixel addresses; 0 is at the top, 300 is at the bottom of the screen (with the Command window closed) and 32767 is at the very bottom, off the screen.

The statement "PSET(H,V)" darkens the pixel which is the Hth pixel from the left in the horizontal direction and the Vth one down from the top in the vertical direction. And, you've figured out that PRESET works in the same way except that it lightens the pixel.

Let's exercise PSET more aggressively. This NEW program will darken any one pixel of your choosing. Type:

```
10 INPUT "HORIZONTAL (0 TO 490)";H
20 INPUT "VERTICAL (0 TO 300)";V
30 CLS
40 PSET(H,V)
```

...and RUN a number of times using various values of H and V.

You may have noticed that if a pixel is lit in the area of the List window, it is covered over when the program ends. Try H = 330 and V = 150. We can avoid this problem by either closing the List window before Running or by not returning control to the prompt -- by adding:

```
99 GOTO 99
```

This Line locks the Computer in an endless loop. Add Line 99, and Run the program trying values of H = 300 and V = 100. To break the loop, press ⌘ . or select Stop from the Run menu.

CLS is a single statement which PRESETs every pixel on the screen to "OFF" in one operation; we don't have a similar statement to turn them all "ON."

However, we can easily write a program that "darkens," or "paints," the entire screen. It uses one CLS (not really a *must*, but always a good habit to use in graphics programs), two FOR-NEXT loops and one endless "locking loop." Type this:

```
10 FOR H = 0 TO 490
20  FOR V = 0 TO 300
30   PSET(H,V)
40  NEXT V
50 NEXT H
99 GOTO 99
```

...and RUN.

The program fills the display from left to right. Redesign it so it starts at the top and fills to the bottom.

Answer

```
10 FOR V = 0 TO 300
20  FOR H = 0 TO 490
```

```
30    PSET(H,V)
40   NEXT H
50 NEXT V
99 GOTO 99
```

Next, rewrite it so it starts painting at the bottom and fills to the top.

Answer

```
10 FOR V = 300 TO 0 STEP -1
20   FOR H = 0 TO 490
30    PSET(H,V)
40   NEXT H
50 NEXT V
99 GOTO 99
```

Did you forget FOR-NEXT could STEP backwards?

Rewrite it so it starts painting at the upper right-hand side and fills to the lower left-hand side.

Answer

```
10 FOR H = 490 TO 0 STEP -1
20   FOR V = 0 TO 300
30    PSET(H,V)
40   NEXT V
50 NEXT H
99 GOTO 99
```

Just for practice, Run the program using other positive and negative STEP increments...

Fantastic -- now we can paint the old barn at least four ways!

EXERCISE 31-1: Write a program which will allow the painting of only a small part of the display (you determine which part). Allow keyboard INPUT to determine the starting and ending pixel numbers in both the horizontal and vertical directions.

Getting the hang of it? Great! Enough playing with blocks ... let's draw some lines. Erase the resident program.

> You haven't forgotten how to do that, have you! Select Erase...no, no! Select New from the File menu.

We'll start our artistry with a straight line. This program PSETs a straight horizontal line across the entire display. Type:

```
10 INPUT "VERTICAL ADDRESS (0 TO 300)";V
20 CLS
30 FOR H = 0 TO 490
40  PSET(H,V)
50 NEXT H
99 GOTO 99
```

...and RUN several times.

We can just as easily create a straight vertical line. Try this.

```
10 INPUT "HORIZONTAL ADDRESS (0 TO 490)";H
20 CLS
30 FOR V = 0 TO 300
40  PSET(H,V)
50 NEXT V
99 GOTO 99
```

...and RUN a number of times.

Now, let's see if we can modify this last program to allow us to INPUT both

the starting vertical address and the length (in pixels):

```
12 INPUT "VERTICAL STARTING ADDRESS # (0 TO
   300)";V
14 INPUT "NUMBER OF VERTICAL PIXELS";N
16 IF V + N < 301 GOTO 20
18 PRINT "TOO MANY VERTICAL PIXELS!"
19 END
30 FOR V = V TO V + N
```

Now that we can draw straight lines, we can form figures -- like squares and rectangles. This program forms a rectangle. After selecting New, type:

```
10 INPUT "HORIZONTAL STARTING ADDRESS (0
   TO 490)";H
20 INPUT "VERTICAL STARTING ADDRESS (0
   TO 300)";V
30 INPUT "LENGTH OF EACH SIDE (IN PIXELS) --
   (0 TO 300)";S
40 CLS
50 FOR L = H TO H + S
60  PSET(L,V)
70  PSET(L,V+S)
80 NEXT L
90 FOR M = V TO V + S
100  PSET(H,M)
110  PSET(H+S,M)
120 NEXT M
999 GOTO 999
```

...and RUN.

You may want to come back later for some heavier study.

A Little Diversion

All our graphics work so far has been drawing dark lines on the light display. We can do just the reverse by painting the display dark first, then lightening the desired areas with PRESET. This New program draws a white horizontal line on a black background. To save time, we will only darken part of the screen. Type:

```
10 INPUT "VERTICAL POSITION (0 TO 150)";V
20 CLS
30 FOR H = 0 TO 120
40  FOR J = 0 TO 150
50   PSET(H,J)
60  NEXT J
70 NEXT H
80 FOR H = 0 TO 120
90  PRESET(H,V)
100 NEXT H
999 GOTO 999
```

...and RUN.

If you're interested, go back and try similar easy modifications to other demonstration programs and have fun with these reverse (or "negative") displays.

Learned In Chapter 31

Statements	Miscellaneous
PSET	Pixel
PRESET	

Chapter 32

Intermediate Graphics

We can draw other straight (more or less) lines by just changing H and V addresses of PSET in the FOR-NEXT loop. Try this New program to draw a diagonal line:

```
10 INPUT "HORIZONTAL STARTING ADDRESS
   (0 TO 490)";H
20 INPUT "VERTICAL STARTING ADDRESS
   (0 TO 300)";V
30 INPUT "DIAGONAL LENGTH";D
40 CLS
50 FOR L = 0 TO D - 1
60  PSET(H+L,V+L)
70 NEXT L
99 GOTO 99
```

Once we have the diagonal line, we can form a right triangle by making these changes and additions:

```
70  PSET(H,V+L)
80  PSET(H+L,V+D)
90 NEXT L
```

or

```
70  PSET(H+D,V+L)
```

```
80  PSET(H+L,V)
90 NEXT L
```

Try them both.

Question: What is the difference in the displays?

Answer: They are inverted, mirror images of each other.

Broken Lines

In every prior graphics program, we could have made the lines "broken" by introducing a STEP other than "1" in the FOR-NEXT loops. For example, try drawing a broken horizontal line with this New program:

```
10 INPUT "VERTICAL ADDRESS (1 TO 300)";V
20 INPUT "STEP SIZE";S
30 CLS
40 FOR H = 0 TO 490 STEP S
50  PSET(H,V)
60 NEXT H
99 GOTO 99
```

Run this program with various values of S. Note that as you increase S, the line is drawn much faster (since the Computer has less work to do). In fact, for S = 10 or more, we can hardly see the line being drawn. This is how a TV picture is created -- since it too is drawn one unit at a time (but so fast we don't notice the "drawing time").

Insert the following Lines into the resident program:

```
5 REM * V MUST BE LARGER THAN 0 *
55 PRESET(H,V-1)
70 V = V + 1
80 IF V < 301 GOTO 40
```

If S is small, we can see each line being drawn and cleared. But if S is fairly large (try 20), the line seems to move in somewhat "old-time-movie" fashion. This is the way the illusion of motion is created on a TV set and in some of the popular video games.

Try this New program. It paints a dot on the display and moves it down.

```
10 INPUT "HORIZONTAL STARTING ADDRESS
   (0 TO 490)";H
20 INPUT "VERTICAL STARTING ADDRESS
   (1 TO 300)";V
30 CLS
40 PRESET(H,V-1)
50 PSET(H,V)
60 V = V + 1
70 IF V < 301 GOTO 40
99 GOTO 99
```

Having problems spotting the dot? Don't worry, it isn't your eyes. The action is so fast and the pixel is so small that it's difficult to spot it. The PRESET statement simply followed along behind and erased the dot from the last PSET.

What happens if you omit PRESET? When you try it, remember to change Line 70 to GOTO 50.

One Minor Detail

If a negative coordinate is used with PRESET and PSET, the line will begin off the screen. Take a look at Line 40:

```
40 PRESET(H,V-1)
```

If you INPUT V equal to −100, then the V address really becomes −101. The line won't appear until V is increased in value to 0.

More Of The Good Stuff

We can just as easily move a point to the right by substituting these Lines:

```
10 INPUT "HORIZONTAL STARTING ADDRESS
   (1 TO 490)";H
20 INPUT "VERTICAL STARTING ADDRESS
   (0 TO 300)";V
30 CLS
40 PRESET(H-1,V)
50 PSET(H,V)
60 H = H + 1
70 IF H < 491 GOTO 40
99 GOTO 99
```

EXERCISE 32-1: Change the last two programs so that they move the dot up and to the left respectively.

Let's have the dot move down until it strikes a barrier. The New program will read:

```
10 INPUT "HORIZONTAL STARTING ADDRESS
   (1 TO 490)";H
20 INPUT "VERTICAL STARTING ADDRESS
   (1 TO 100)";V
30 INPUT "LOWER BARRIER (200 TO 300)";B
40 CLS
50 FOR M = 0 TO 490
60  PSET(M,B)
70 NEXT M
80 PRESET(H,V-1)
```

```
90 PSET(H,V)
100 V = V + 1
110 IF V < B GOTO 80
999 GOTO 999
```

The dot appears to strike the barrier and stick to it.

Now let's have the dot start in the middle and ricochet off the top and bottom. Select New, then enter this program:

```
10 FOR H = 0 TO 490
20  PSET(H,50)
30  PSET(H,250)
40 NEXT H
50 V = 150
60 D = 1
70 PRESET(245,V-D)
80 PSET(245,V)
90 V = V + D
100 IF V = 251 GOTO 120
110 IF V <> 49 GOTO 70
120 V = V - 2 * D
130 D = -D
140 GOTO 80
```

The change in direction of the moving dot is caused by:

```
130 D = -D
```

Note that we must be careful not to accidentally erase part of the boundary. To do this, we move the dot back 2 steps with Line 120 (after moving it forward 1 in Line 90), but we also return to the PSET in 80 rather than to

PRESET in 70. Tricky, tricky. You can kill the whole day messing around with this silly bouncing ball. Rather good resilience, eh?

Save this program As BOUNCE for use in the next Chapter.

Real Moving Pictures

We can draw whatever figures we like. Let's try a stick man. First, his legs:

Select New, then:

```
10 H = 64
20 FOR K = 0 TO 30
30  PSET(H+K,200+K)
40  PSET(H-K,200+K)
50 NEXT K
999 GOTO 999
```

...and RUN.

Then add his body and arms:

```
60 FOR K = 0 TO 20
70  PSET(H+K,179+K)
80  PSET(H,179+K)
90  PSET(H-K,179+K)
100 NEXT K
```

...and RUN.

And finally his head:

```
110 FOR K = 0 TO 4
120  PSET(H+K,169+K)
130  PSET(H-K,169+K)
```

```
140  PSET(H+K,178-K)
150  PSET(H-K,178-K)
160 NEXT K
```

...and RUN.

Now let's try and move him to the right. Add:

```
35  PRESET(H+K-2,200+K)
45  PRESET(H-K-2,200+K)
75  PRESET(H+K-2,179+K)
85  PRESET(H-2,179+K)
95  PRESET(H-K-2,179+K)
125  PRESET(H+K-2,169+K)
135  PRESET(H-K-2,169+K)
145  PRESET(H+K-2,178-K)
155  PRESET(H-K-2,178-K)
170 H = H + 2
180 GOTO 20
```

...and RUN.

Sure moves funny, doesn't he? Well, I'm no animator either, but you're beginning to get the idea.

Line Drawing With LINE

We have been drawing lines (horizontal, vertical and diagonal) by using PSET. There is an easier and shorter method for drawing straight lines. Type in this New program:

```
10 LINE (100,40)-(50,90)
20 LINE - (150,90)
```

```
30 LINE - (100,40)
```

...and RUN.

WOW! Now that's fast. It only took 3 program Lines to draw 3 display lines. A similar program using PSET would require about five loops. By analyzing each Line, we'll discover that LINE is actually similar to PSET. LINE and PSET use the same Horizontal and Vertical address numbers to spot the starting point on the display.

Nice. But what is the program doing?

Line 10 *draws* a diagonal LINE by following the dots from coordinates 100 (horizontal) and 40 (vertical) to (-) 50 (horizontal) and 90 (vertical).

Line 20 draws the horizontal base line. The to (-) and the destination coordinates are all that are included in this LINE statement. When the starting coordinates are omitted, the Macintosh uses the coordinates last used by PSET or LINE. In this example, the last pixel turned on in Line 10 was at (50,90). So, Line 20 really says, "From the last coordinate (50,90), draw a LINE to (-) horizontal position 150 on vertical line 90."

Line 30 begins at the last pixel turned on by Line 20 and draws the third LINE up to the top of the triangle.

Zeroing Out The LINE

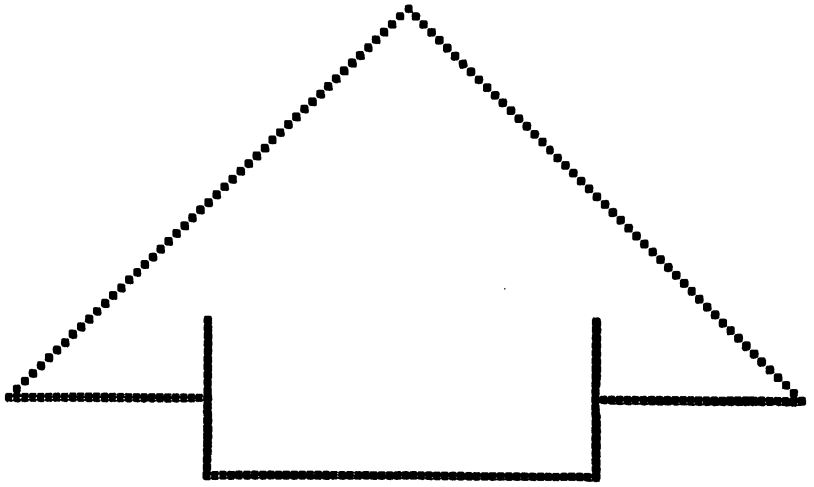
Add Line 40 to the resident program:

```
40 LINE (75,90)-(125,90),0
```

...RUN

By selecting a portion of the horizontal LINE and adding ,0 to the end of the LINE statement, we made LINE act like PRESET. Do some experimenting on your own with LINEs before going on.

EXERCISE 32-2: Add three Lines to the LINE program that will draw this completed diagram:



Drawing Boxes

We can also draw boxes without topses. Try this NEW program:

```
10 LINE (5,8)-(234,118),1,B
99 GOTO 99
```

Line 20 did all that? The first coordinate (5,8) established the top left-hand corner of the box, and (234,118) set the bottom right hand corner. Then B told LINE to connect these two points in the shape of a Box.

How about a box within a box? Add:

```
20 LINE (50,18)-(190,108),1,B
```

Now wouldn't it look neat to fill in the inside box? Suppose that requires several FOR-NEXT loops and a PSET? Wrong, just adding the letter F to our LINE statement Fills in the Box. Change Line 20 to:

```
20 LINE (50,18)-(190,108),1,BF
```

...and RUN.

Just like that, the Box is Filled.

Do you suppose there are any more tricks left in the LINE statement? Well, there just happens to be one more. We can remove an entire box the same way we removed a LINE. Let's add the finishing touches to this program with these Lines:

```
30 LINE (55,23)-(185,103),0,B
40 LOCATE 4,11 : PRINT "BURMA"
50 LOCATE 5,11 : PRINT "SHAVE"
```

Study these three new Lines closely. By tracing the coordinates used in Line 30's LINE statement, we can see how the pixels were turned off inside the black box boundary. The Computer automatically clears an area around the words PRINTed by Lines 40 and 50.

These have been two long and active Chapters ... and to think, all this with only the PSET, PRESET, and LINE statements. In many cases, by simply exchanging PRESET for PSET or a 0 for a 1, we could have drawn the same pictures with light on a dark background instead of dark on light. You might want to give it a try.

Because the ideas come so fast in the area of graphics, we have deliberately chosen to show a number of straightforward examples rather than get bogged down in elaborate programs. There is no substitute for lots of experimenting with graphics, and you now know the basics. Put in your time, study the examples, and soon you can apply for membership in the artists' guild.

Learned In Chapter 32

Statements	Miscellaneous
LINE	Animated graphics
	Diagonal lines
	Boxes
	Filling Boxes

Chapter 33

Formatting With LOCATE

We used this in the last chapter; now let's see what it's all about. The LOCATE statement allows PRINTing to *begin* anywhere on the screen. Type:

```
10 CLS : LOCATE 8,10
20 PRINT "HELLO THERE, 8,10!"
```

...and RUN.

The LOCATE locations start at 1,1 in the upper-left hand corner and, as long as a WIDTH hasn't been specified, go through to 1,32767 in the first line. They pick up on the second line with 2,1 and continue through to 32767,32767 (the lower-right hand corner). This is well beyond the limits of what we can see on the display, so we will normally restrict the LOCATE numbers to within 16,60.

The actual number of spaces available in the line depends on the value selected in the WIDTH statement. If the WIDTH statement is not used, the default line width is 32767 spaces.

If we want to PRINT on line 19, we must first close the Command window and expand the Output window to the bottom of the display or else our PRINTed message will be lost beneath the Command window.

It's That Time Already?

Let's create a 24-hour clock. Sounds like more fun than digging through this obscure LOCATE statement mapping. Type:

```
10 LOCATE 7,28
```

```
20 PRINT "H M S"
30 FOR H = 0 TO 23
40   FOR M = 0 TO 59
50     FOR S = 0 TO 59
60       LOCATE 8,27
70       PRINT H;":";M;":";S
80       FOR N = 1 TO 2000 : NEXT N
90     NEXT S
100  NEXT M
110 NEXT H
120 GOTO 10
```

Save As TIMER ... and RUN.

Nothing to it. Ahem!

"Hello? Bureau Of Standards?"

Of course, the accuracy of this timer depends on how closely we calibrate it. We earlier discussed that the Mac will execute somewhere around 2200 simple FOR-NEXT loops per second. When used in a program like this, the FOR-NEXT value in Line 80 must be adjusted to allow for the activity going on with the various other FOR-NEXT loops and LOCATE statements. If we really get carried away with this program, it can be calibrated against a precision timepiece, increasing or decreasing the "2000" as needed, or better yet, using our TIME$ statement along with LOCATE. Over the short run, it is quite a good timer. Note that we are not triggering this with the power line frequency or a crystal oscillator, but relying solely on the amount of time required to execute FOR-NEXT loops. (It's not nearly as accurate as the clock built into the Computer.)

Oh, Yes ... The LOCATE

Anyway -- let's not lose sight of the forest for the trees (or is it trees for the forest). The purpose of this program is to demonstrate the LOCATE statement. We used it twice. With blazing speed, the HMS (no, no, not Her Majesty's Service -- it stands for Hours, Minutes and Seconds) are PRINTed -- and the HM&S are updated each second.

For a better clock program, the real clock nut only needs to calibrate this program a little closer to be an acceptable sundial. Then the Computer becomes the most expensive clock in the house!

POS(N) And CSRLIN

An additional and sometimes useful function allows the Computer to report back the horizontal POSition of the cursor. This simple New program exercises the POS function.

```
10 INPUT "A NUMBER BETWEEN -9 AND 50";A
20 CLS
30 PRINT TAB(10+A)
40 PRINT POS(0);
50 PRINT " WAS NUMBER OF NEXT PRINT POSITION"
```

...and RUN.

Line 40, containing POS, is the key.

The 0 inside the brackets is just a "dummy." Most any other number or variable would work as well -- but something has to be placed there. POS reports back any cursor POSition on the screen up thru 4096. Numbers above 4096 are reported as 1.

To help locate the cursor, we can add these Lines to the resident program:

```
45 L = CSRLIN
55 PRINT "AND WAS LOCATED IN LINE";L
```

CSRLIN tells us the CuRSor LINe (Row) number from 1-2048 that the cursor was on at the time CSRLIN was encountered.

Remember, CSRLIN returns the CuRSor LINe (Row), and POS(0) returns the column.

The CSRLIN and POS limits will vary somewhat when different fonts are selected.

That's How The Ball Bounces

Meanwhile, back with the bouncing ball. Select Open from the File menu, and double click the program named "BOUNCE" Saved in the last Chapter. It reads:

```
10 FOR H = 0 TO 490
20  PSET(H,50)
30  PSET(H,250)
40 NEXT H
50 V = 150
60 D = 1
70 PRESET(245,V-D)
80 PSET(245,V)
90 V = V + D
100 IF V = 251 GOTO 120
110 IF V <> 49 GOTO 70
120 V = V - 2 * D
130 D = -D
140 GOTO 80
```

Since we did not explain in detail how that fairly simple program worked, take time now to see if you can follow it through. Concentrate your thinking on the PSET and PRESET Lines and the logic that gives them their numerical values. When you have it figured out, tackle this exercise:

> **EXERCISE 33-1:** Using LOCATE statement(s), cause the word "PING" to appear near the ball each time it bounces off either the top or bottom boundary. A sample answer is in Section B.

Isn't it amazing how close we are building towards some of the video games that are all the rage -- and yet it's really so simple and logical.

WRITEing To The Screen

The WRITE statement allows us to PRINT on the screen. It is similar to

PRINT, except that the WRITE statement automatically *inserts a comma* between each item the Computer WRITEs on the screen. It also places quotes around all strings. Try this New program:

```
10 READ A,B,C,D$
20  WRITE A;B;C;D$
30 DATA 100,200,300, ...ETC
```

Variables in Line 20 can be separated by semicolons or commas. The Computer will treat them the same. Note the 4th piece of DATA in Line 30 is a string but has no quotes.

Run, and see:

```
100,200,300," ...ETC"
```

We already know that BASIC is unable to READ a string from the DATA Line if it contains quotes within quotes. By using WRITE, we can READ a string and let the Computer insert the quotes.

Learned In Chapter 33

Statements	Functions
LOCATE	POS(N)
WRITE	CSRLIN

Chapter 34

Graphing Trig Functions

It is often helpful to graph mathematical functions to better understand what's happening. Macintosh graphics can be used for a non-precision examination of many mathematical functions, and the following short demo programs illustrate that capability.

Just imagine there is an X-Y coordinate system drawn on the display (or draw *your* own, either with the Computer or a china marker). The numbers in these demo programs are not magic; they just allow the graphs to be drawn large, but not so large they run off the display.

These programs are included to show how LOCATE can be used in a supporting role to the Macintosh graphics. Experiment to get what you want for your own particular application.

A Single Sine Wave

```
10 LOCATE 1,1 : PRINT "SINE"
20 FOR X = 0 TO 240
30  Y = SIN(X/38)
40  PSET(X,150-Y*100)
50  LOCATE 1,40 : PRINT "X =";X
60  LOCATE 1,50 : PRINT "Y ="; INT(150-Y*100)
70 NEXT X
80 GOTO 80
```

Graph Of 3 Cosine Waves

```
10 LOCATE 1,1 : PRINT "COSINE"
20 FOR X = 0 TO 720
30  Y = COS(X/38)
40  PSET(X/3,150-Y*100)
50  LOCATE 1,40 : PRINT "X =";INT(X/3)
60  LOCATE 1,50 : PRINT "Y =";INT(150-Y*100)
70 NEXT X
80 GOTO 80
```

Graph Of The Tangent

```
10 LOCATE 1,1 : PRINT "TANGENT"
20 FOR X = 0 TO 240
30  Y = TAN(X/160)
40  PSET(X,230-Y*16)
50  LOCATE 1,40 : PRINT "X =";INT(X)
60  LOCATE 1,50 : PRINT "Y =";INT(230-Y*16)
70 NEXT X
80 GOTO 80
```

There is obviously quite an education to be had by a careful study of the graphs. Look for such things as relative density of the line at different points, the rate at which blocks are lit relative to the other variable, etc. Sure beats the "early days" when we had to try and imagine these things on a blackboard.

Merely For Display Purposes

A good way to get a feel for LOCATE (or any other feature) is to look at

a fairly simple program which illustrates its use. This New program lays out a graph format on the display. Type:

```
10 LOCATE 1,20 : PRINT "G R A P H
   H E A D I N G"
20 REM * HORIZONTAL MARKERS *
30 FOR X = 1 TO 50
40  LOCATE 17,10+X : PRINT "."
50 NEXT X
60 REM * HORIZONTAL NUMBERS *
70 FOR X = 0 TO 45 STEP 5
80  LOCATE 18,10+X : PRINT X
90 NEXT X
100 REM * VERTICAL MARKERS *
110 FOR Y = 1 TO 14
120  LOCATE Y+2,9 : PRINT "-"
130 NEXT Y
140 REM * VERTICAL NUMBERS *
150 FOR Y = 1 TO 14
160  LOCATE Y+2,1 : PRINT 14-Y
170 NEXT Y
999 GOTO 999
```

Remember to close the Command window and enlarge the Output window to make use of the entire screen.

What you do with these programs beyond this point depends on your own needs and interest, but they are worth entering, studying and becoming comfortable with.

Learned In Chapter 34

Miscellaneous

Graphing with LOCATE

Chapter 35

INKEY$ And INPUT$

The INKEY$ (pronounced Inkey-string) function is a powerful one which enables us to INPUT information from the keyboard without having to use the Return key.

Enter this New program:

```
10 IF INKEY$ = "T" THEN 30
20 GOTO 10
30 PRINT "YOU HIT THE LETTER 'T'"
40 GOTO 10
```

...and Run.

Press a variety of individual alpha and numeric keys.

The keyboard seems to be dead until we hit the T key. Why?

Aha! The test in Line 10 is passed, execution moves to Line 30 and a message is PRINTed. Then the process starts over. Hit T again. Hold it down.

The way INKEY$ works is clever if somewhat subtle, so pay close attention.

The Macintosh keyboard is constantly scanned by the Computer, checking to see if any key is pressed. If a key was pressed before the Computer encounters INKEY$, the character that key represents was stored in the INKEY$ storage, or buffer, area. The buffer can hold only one character at a time so when a new key is pressed, that new character replaces whatever preceeded it in the buffer, if anything. INKEY$ automatically assumes the String Value of whatever character is in its buffer.

Since INKEY$ can only "photograph" one letter or number at a time, if we want to test for more than one character, we have to write the program to test for each one in sequence. In so doing, however, we must be careful or INKEY$ will trip us up.

Press ⌘. to stop the program and ⌘L to List it. Then add these Lines to the program:

```
15 IF INKEY$ = "P" THEN 50
50 PRINT "YOU HIT THE LETTER 'P'"
60 GOTO 10
```

...and Run again.

As you can see, we no longer get an "instant" response each time T or P are pressed. This distressing condition exists because the INKEY$ buffer is cleared and reset to a null string *each* time INKEY$ is hit. Aaawk! Just when it was starting to make sense. Select Stop from the Control menu, or press ⌘., and List (⌘L) so we can take a good look at the program to see how this clearing of the buffer results in the "loss" of a keystroke.

Suppose the operator presses the T key just as Line 15 begins execution. Where does the T go? Into the INKEY$ buffer, of course. There it sits until another key is pressed, or INKEY$ is "called."

When Line 15 is executed, INKEY$ "reads" the buffer. The buffer's current value (T) is compared to Line 15's "P". Since the two strings are not equal, control passes to the next Line, then back to Line 10. In Line 10, INKEY$ is called again, but when it checks the buffer this time, "T" is gone. What happened to the T?

Let's replay that last sequence and zoom in for a closeup on the INKEY$ buffer. As the operator hits the T key, we see the T stored in the buffer. As the INKEY$ Function in Line 15 is executed, the buffer suddenly goes blank. Ahhhhh! Thank heavens for instant replay. It's now obvious that each time INKEY$ is called, its buffer is cleared, whether or not it meets the string test in the Line calling it. If we want to preserve the value of T, we'll need to store it elsewhere, maybe in a temporary string variable.


WAKE UP HARRY!
JUST WHO IS THIS "INKEY STRING" YOU KEEP MUMBLING ABOUT?

Change Lines 10 and 15 to:

```
10 A$ = INKEY$ : IF A$ = "T" THEN 30
15 IF A$ = "P" THEN 50
```

...and Run.

Aha! Now we're getting somewhere. Give it the ultimate test -- alternate pressing T and P as quickly as possible.

By setting a "regular" string variable equal to INKEY$ and having T and P checked against the variable instead of against the INKEY$ buffer, we store its value for as long as needed and process it much more efficiently and predictably.

Rapid Scanner

If INKEY$ scans the buffer and does not find a key pressed (the usual case), it is said to read a "null string." INKEY$ is a string Function, and null means *nothing*. A null string is represented by two quote marks with nothing between them, thus:

```
""
```

The ASCII code for null is 0.

To see how fast we can scan for INPUT with INKEY$, try this New program:

```
10 K$ = INKEY$
20 IF K$ = "" THEN PRINT "NO KEYBOARD INPUT"
30 PRINT ,,K$ : GOTO 10
```

...and Run.

Type in random characters and words, and see them break the scan.

Get the general idea how to use INKEY$? So simple, yet the possibilities are enormous. Only a lot of experimenting will make you comfortable with it, but INKEY$ will keep you awake nights staring at the ceiling thinking of ways to put it to work.

Out Of The Blue Of The Western Sky...

While chasing the solitude needed to write computer books, your author piloted a heavily loaded private plane, packed with computers, ham radio and other goodies, into a medium sized city airport. Transferring this freight to a rental car turned out to be a big deal since security wouldn't let a car on the apron to unload the plane. (You're supposed to drop it by parachute?)

After some cajoling (and a gratuity), it was agreed that my car could be driven up *near* the apron, and an "officially approved" car would haul the goodies from the plane to my car. It seemed a bit officious, but elections were far away...

Anyway, to get my car thru the security fence it was necessary to open an electrically-operated gate. A secret code was punched into a numeric keypad for some sort of computer to analyze, and it controlled the motorized gate. *The secret code number was 1930.*

Needless to say, as soon as the computer was set up, I wrote a BASIC program to do everything but actually open the gate. It provides a good example of a real-life application of INKEY$ and is offered here for your amusement, amazement and careful study.

```
10 LOCATE 7,21 : PRINT "TYPE THE COMBINATION"
20 LOCATE 8,20 : PRINT "FOLLOWED BY AN
   ASTERISK"
30 LOCATE 2,18 : PRINT "THE ELECTRIC GATE IS
   CLOSED"
40 K$ = INKEY$ : IF K$ = "" THEN 40
50 READ D$ : IF D$ = "*" THEN 80
60 IF D$ = K$ THEN 40
```

```
70 RESTORE : GOTO 40
80 CLS : LOCATE 2,22 : PRINT "YOU MAY ENTER
   NOW"
90 LOCATE 4,19 : PRINT "WAIT FOR THE GATE
   TO OPEN"
100 FOR T = 1 TO 3000 : NEXT T
110 CLS : RESTORE : GOTO 10
DATA 1,9,3,0,*
```

Save As GATE and Run. Try the combination.

The password (1930 followed by an asterisk) is imbedded, a character at a time, in the DATA Line. The commas only separate the characters and should not be typed in as part of the password.

Line 40 holds the magic. It stores the buffer contents in K$ and checks K$ for something besides a null string. If it finds a key was pressed, execution drops to Line 50.

Line 50 READs a piece of DATA. If it happens to be an asterisk (which can only be READ from DATA after all of the other code characters have been READ), execution moves to Line 80 where the gate is OPENed.

If, however, the test in Line 50 does not find an asterisk, execution defaults to the next test, in Line 60.

Line 60 checks to see if the keyboard character matches up with the character READ from DATA. If so, the first hurdle has been passed and execution returns back to Line 40 for INKEY$ to await another keyboard character. If the keyboard and DATA characters don't match, the test fails and execution drops to Line 70.

Line 70 RESTOREs the DATA pointer back to its beginning and returns execution to Line 40 to start scanning all over again. The keypad puncher sees none of this and has no idea if he is making progress towards cracking the code.

Line 100 merely allows the gate a brief time to open and close (and us to read the screen), then

Line 110 CLears the Screen, RESTOREs the DATA and starts the program over from the beginning.

The password can be changed to any combination of characters by changing the DATA Line. If we wanted it to be 'MACINTOSH' for example:

```
DATA M,A,C,I,N,T,O,S,H,*
```

Or 'OPENSESAME'

```
DATA O,P,E,N,S,E,S,A,M,E,*
```

Don't forget that last piece of DATA, the asterisk.

Happy gate crashing!

INPUT$

INPUT$ can be thought of as a *multi-character* INKEY$. It allows us to INPUT a certain number of characters from the keyboard without printing them on the screen. It's great for entering passwords.

Make the following changes:

```
40 READ PASSWORD$
50 L = LEN(PASSWORD$)
60 K$ = INPUT$(L)
70 IF K$ = PASSWORD$ THEN 80 ELSE 60
DATA MACINTOSH
```

...and Run.

Very carefully, type MACINTOSH (no Return).

This change has a disadvantage in that once you start typing, there's no way to start over if you make a mistake. With more elaborate programming, a "reset" could automatically take place after a period of time.

I hope you enjoyed this Chapter as much as I did in creating it.

Learned In Chapter 35

Functions	Miscellaneous
INKEY$	INKEY$ Buffer
INPUT$	Null String

Chapter 36

PRINT USING

Of all the ways we have to PRINT, the most powerful (and most complex) is one called PRINT USING. The name PRINT USING implies that we PRINT by USING something else. That implication is correct.

As originally developed for use on large computers, PRINT USING consists of two parts -- PRINT and USING. PRINT prints, USING the format (called the "image") found in *another* Line. The Macintosh PRINT USING feature is similar, but does not always require a second Line for the "image" ... as we will see.

PRINT USING With Numbers

Type in this New program:

```
10 A = 123.456789
40 U$ = "###.##"
50 PRINT USING U$;A
```

...and Run.

The answer is:

```
123.46
```

A was rounded UP and PRINTed to an accuracy of 2 *decimal* places, following the same format as Line 40, the *image Line.*

Add:

```
20 B = 1.6
60 PRINT USING U$;B
```

...and Run.

The Display shows:

```
123.46
  1.60
```

Notice that we called upon Line 40, the *image Line*, twice -- once in Line 50 and again in Line 60. Also, note that the answers appear with their decimal points lined up. Last, see that a 0 has been added to the 1.6 to make it read 1.60. These latter 2 points are important when PRINTing out financial reports.

One more addition:

```
30 C = 9876.54321
70 PRINT USING U$;C
```

...and Run, produces:

```
123.46
  1.60
%9876.54
```

Oh-oh! Vas ist los?

Well, the % sign means we have overrun our *image Line's* capacity to PRINT digits *left* of the decimal point, but it PRINTs them anyway. Better to lose the decimal point lineup than important numbers, but it does call our attention to a programming problem.

Let's add another # sign to make room for that extra digit left of the decimal point. (We are adding another *element* to the *field* in the *image Line*. Got that?)

```
40 U$ = "####.##"
```

Line 40 now has 4 elements in "left field" and 2 in "right field." The decimal point is the dividing point.

...and Run.

That's better -- but the overRUN message would appear again if we tried to PRINT a number with more than 4 digits on the left.

This PRINT USING business looks like it might have some potential, lining up decimal points like it does. We don't have any other reasonable, straightforward way to accomplish that, and it's essential for PRINTing dollars and cents. Wonder how we can PRINT a dollar sign?

Change the *image Line* to:

```
40 U$ = "$####.##"          (Check 'em carefully.)
```

...and Run.

Nice, eh? The dollar signs all line up in a row:

```
$ 123.46
$   1.60
$9876.54
```

But suppose we want the dollar signs to snug right up against each dollar amount? Make 40 read:

```
40 U$ = "$$###.##"
```

...and Run.

and shure enuf:

```
  $123.46
    $1.60
$9876.54
```

Not an especially attractive format, but taken singly, as when writing checks, it's almost essential.

The lessons so far are:

1. PRINT USING with # lines up the decimal points.

2. It rounds off the cents (the numbers to the right of the decimal point) to the number of elements specified. It does not round off dollars (left of the decimal point), but sends up an error flag (%), PRINTs all dollars and slips the printout to the right if the field isn't large enough.

3. If a single $ is added to left field, dollar signs are PRINTed and lined up in a column like decimal points. This single $ does not expand the field.

4. If two $ are placed on the left, one $ will be PRINTed on each Line immediately in front of the first dollar digit. One of these $'s can be used to replace one # in the field, thereby not expanding it.

We've covered a lot in a very small program, but have a long ways to go.

Printing Checks

When using a printer for writing checks, it's usually wise to take extra precautions against "alterations." This is easily accomplished by changing Line 40 to read:

```
40 U$ = "*****.##"          (Count 'em.)
```

...and Run.

The display reads:

```
**123.46
****1.60
*9876.54
```

That's swell. It fills up the unused space alright, but there's no dollar sign. Okay, replace the first # sign with a dollar sign, like so:

```
40 U$ = "**$##.##"
```

Aren't you glad we have an Editor for all these changes?

See it Now:

```
*$123.46
***$1.60
$9876.54
```

just like they do it uptown! Only 1 $ was needed when using leading *'s, compared to $$ without them.

If we really want to impress others with the size numbers we usually deal in at our local lemonade stand, add lots more # signs to the *image Line*, thus:

```
40 U$ = "**$##################.##"
```

and our checks read:

```
**************$123.46
****************$1.60
************$9876.54
```

...very impressive.

An Illegal funcion call error will occur if more than 24 characters are assigned to a PRINT USING variable.

Since we're obviously big time operators, having now franchised the lemonade stands, it's getting hard to keep track of the big numbers. How about some commas to break them apart? (Knock out those extra #'s first. Too hard to count them.)

```
40 U$ = "**$,##.##"          (Look closely.)
```

...and Run.

```
**$123.46
****$1.60
$9,876.54
```

Only one of our numbers has more than 3 digits in left field, but a comma separated its 9 and 8 for easier readability. In the image field, the comma can be placed *anywhere* between the $ and the decimal point, and only *one* comma is required to automatically insert commas to the left of every 3rd digit left of the decimal point. (You really big-time operators who deal in the millions will have to wait 'til the next chapter to see how to go "double precision" to avoid losing the loose change.)

NOTE: The comma does *not* serve as a field element.

Stringing It Out

Let's rework the resident program to show some other PRINT USING capabilities:

```
10 A = 123.456789
20 B = 1.6
30 C = 9876.54321
40 U$ = "####.##   5   ####.##   5   ####.##"
```

```
50 PRINT USING U$;A,B,C
```

...and Run.

The PRINT USING statement will reuse its image Line until all the fields are PRINTed.

Shorten Line 40 to:

```
40 U$ = "####.##      "
```

...and Run again, with the same effect.

See how numbers can be displayed horizontally instead of vertically? Line 50 determines *where* the fields are PRINTed.

```
123.46    9    1.60   6   9876.54
```

EXERCISE 36-1: Write the various forms Line 40 must take to PRINT these formats:

```
123.46          1.60          9876.54
$ 123.46        $   1.60      $9876.54
$123.46         $1.60         $9876.54
$123.46         $1.60         $9,876.54
```

and finally

```
***$123.46      *****$1.60      *$9,876.54
```

PRINT USING With Strings

Select New, and enter this program:

```
10 A$ = "IT'S"
20 B$ = "HOWDY"
```

```
30 C$ = "DOODY"
40 D$ = "TIME"
50 U$ = "\\"
60 PRINT USING U$;A$
```

...and Run.

The only thing unique about this program are the back slashes in Line 50. \ is a symbol in Macintosh PRINT USING which is to strings something like what the # is to numbers.

The \\ reserved 2 spaces for strings. Only IT was PRINTed. Unlike #, however, to reserve more spaces in a *string* field, we must add spaces between the \ signs. Change Line 50 to:

```
50 U$ ="\ 2 \"
```

(The small 2 is just for us.)

...and Run.

4 spaces are set aside, and IT'S is PRINTed without clipping.

Let's make room for PRINTing another string on the same line.

```
50 U$ = "\ 2 \\ 3 \"
60 PRINT USING U$;A$,B$
```

...and Run.

Oops! We ran:

```
IT'SHOWDY
```

together.

To space them apart we have to put an actual space in the image field just as we did earlier when PRINTing numbers.

```
50 U$ = "\ 2 \1\ 3 \"
```

...and Run.

That's more like it.

Now it's your turn. Complete Lines 50 and 60 to print `IT'S HOWDY DOODY TIME` all on one line.

Answer:

```
50 U$ = "\ 2 \1\ 3 \1\ 3 \1\ 2 \"
60 PRINT USING U$;A$,B$,C$,D$
```

...and Run.

It's time to quit doodling around and get down to business! Change our HOWDY DOODY to some typical report headings.

```
10 A$ = "PART NUMBER"
20 B$ = "DATE PURCHASED"
30 C$ = "DESCRIPTION"
40 D$ = "COST"
50    (Figure out this one yourself)
60 PRINT USING U$;A$,B$,C$,D$
```

Assignment: Design the *image* needed in Line 50.

Answer:

```
50 U$ = "\    9 spaces    \ 4 \        12        \ 4 \ 9
          \ 4 \ 2 \"
```

...and Run.

EXERCISE 36-2: Duplicate the following statement. Use PRINT USING for all but the column headings.

```
                          CREDITS       TAX       TOTAL
ASTRAL COMPUTER             18.30       .70       19.00
BIOFEEDBACK ADAPTER          1.80       .00        1.80
PERSONALITY MODULE           7.20       .30        7.50
                                       DUE:       28.30
```

Learned In Chapter 36

Statements

PRINT USING

Miscellaneous

Image Line
PRINT USING
Symbols
. $ * , \

Chapter 37

PRINT USING -- Round 2

In the previous Chapter we learned almost everything really needed to put PRINT USING to work. Here are a few other "tricks" that some will find helpful.

When PRINTing big bucks (over 9,999,999 dollars), it is necessary to use double precision or we lose the loose change. Type in this New program:

```
10 A$ = "$$#######,###.##"          (Count 'em.)
20 D = 123456789.01
30 PRINT USING A$;D
```

...and Run.

Sure enough, it rounds to $123,456,792.00. Granted, it's only a few seconds interest on the national debt, but for businesses doing the *taxpaying*, the accuracy can be easily improved by simply switching to double-precision.

Change Lines 20 and 30 to:

```
20 D# = 123456789.01
30 PRINT USING A$;D#
```

There it is -- $123,456,789.01 -- even the change to tip the porter who hides the public baggage carts. Notice that the *image Line* didn't have to change? All we did was use the double-precision techniques we learned earlier.

If the 17-place accuracy of double precision isn't adequate to keep track of the Krugerrands in your mattress, you and Scrooge McDuck can probably afford to spring for a bigger computer.

Profit Or Loss?

Was that last number this quarter's PROFIT from the lemonade stand, or was it a LOSS? We can make the *image Line* PRINT either one. Change it to read:

```
10 A$ = "+$$#######,###.##"
```

...and Run.

Very nice. Wonder what would happen if D# was a negative number?

```
20 D# = -123456789.01
```

...and Run.

So far, so good. Suppose we take the + out of the image Line. Wonder if it will PRINT the minus sign anyway? Use the Editor, and remove it from Line 10.

Then Run.

Oh, Pshaw! It goofed it up. Negative numbers require one more field element than positive numbers, and the extra $ doesn't do the job. The + did count as an element, so let's put the + sign back in, this time at the *end* of the image.

```
10 A$ = "$$########,###.##+"
```

...and Run.

Mmmmm. That's nice. The sign is PRINTed at the end. Let's change D# back to a *positive* number and see what happens.

```
20 D# = 123456789.01
```

...and Run.

Very nice. Looks better to have the signs at the end, not interfering with the dollar sign, don't you think?

Most printers don't PRINT deficits in red. How can we tag them so it's harder for the project manager to slip them by us? (We'll just take all + numbers for granted.) Let's try changing the image + to a minus and see what happens.

```
10 A$ = "$$########,###,##-"
```

...and Run.

Seems normal. How about when it's hit with a negative number.

```
20 D# = -123456789.01
```

...and Run.

AHA! Sticks out like a sore thumb. Now about this little deficit here, Smythe...

EXERCISE 37-1: Duplicate this simplified ledger by use of PRINT USING:

```
REVENUES          EXPENSES  ASSETS
1,203,104.22          0.00  1,203,104.22
        0.00    560,143.80    560,143.80-
```

More On Strings

There are two more PRINT USING characters that have real value. Like so many exotic "upgrades" of BASIC, it does nothing that can't be achieved using other BASIC words, but does it easier. Enter this New program:

```
10 X$ = "ALEXANDER"
20 Y$ = "GRAHAM"
30 Z$ = "BELL"
40 A$ = "!1!1\ 2 \"
50 PRINT USING A$;X$,Y$,Z$
```

...and Run.

Who should appear before our very eyes but:

```
A G BELL
```

Each ! reserves an element in the field for the *first letter* of the string assigned to it. Very handy when we want to PRINT the initials and last names of a list of people.

Another Short Cut

An area of PRINT USING worthy of examination is incorporation of the *image Line* into the PRINT USING Line. It requires some care and has value primarily when only a few variables are to be PRINTed or are only PRINTed once. In most practical applications, the *image Line* is referenced many times during a Run, frequently by different PRINT USING Lines.

Change Line 50 and delete Line 40 in the resident program so it looks like this:

```
10 X$ = "ALEXANDER"
20 Y$ = "GRAHAM"
30 Z$ = "BELL"
50 PRINT USING "!1!1\ 2 \";X$,Y$,Z$
```

...and Run.

We simply did away with A$ and incorporated its elements into a combination PRINT and *image Line*, separated by a semicolon. It does save space, and for short and uncomplicated PRINT USING applications, has great value. For the long and complicated ones, it's better to keep the *image* and PRINT USING Lines separate.

INPUTting The Image

We move deeper into the woods as we make BASIC's PRINT formatting capabilities resemble the superior (and far more complicated) ones of the FORTRAN language from which it is derived. We can even INPUT the *image*

Line, since it is a string. An easy way to see this is by using our resident program, adding Line 40 and changing Line 50:

```
40 INPUT A$
50 PRINT USING A$;X$,Y$,Z$
```

...and Run.

We now have to respond by typing in the *image Line*. (Seems like they're hard *enough* to create without INPUTting.) The safest one to use is old Line 40, so respond to the question mark with:

```
? !1!1\ 2 \                    Return
```

and see:

```
A G BELL
```

appear again.

Run again, this time responding with something like:

```
? \    7    \1\   4  \1\ 2 \
```

and we should see something like:

```
ALEXANDER GRAHAM BELL
```

Try some other INPUTs and see how fast we get into trouble with "Type Mismatch" errors. The down-to-earth value of this particular capability is a little elusive.

Let's experiment with a new PRINT USING character. Run the program again, and when it asks for an INPUT type:

This new character allows the use of variable length strings. The three names are concatenated and PRINTed. This is a little strange because it defeats the purpose of PRINT USING since there's no way to control column placement. However, when the right application pops up, it's there to use.

Scientific Forms Of PRINT USING

Would you believe a double-precision number, clipped and expressed via PRINT USING in double-precision Exponential notation? The technical types among us with mismatched socks and rope for a belt will salivate at that one. We aren't going to bore the business types with the gory details except for a quick intro.

Type in this New program:

```
10 A$ = "####################^^^^"          (18+4)
20 D# = 12345678909876543 21
30 D = 12345678909876543 21
40 PRINT USING A$;D!
50 PRINT USING A$;D
```

...and Run.

What we see is what we get, both in double and single precision. Using the Editor, move the block of 4 carets (up-arrows) to the left, one position at a time, by deleting #'s and then adding #'s to the right of the carets. Have fun!

Bring On The Money Changers

Here is a straightforward user program which uses PRINT USING in a practical way. One would be hard pressed to get the same results in so short a program without USING it.

If you're not in the international currency biz, just type in the first half-dozen or so DATA Lines, plus Line 1500 to get a feel for what PRINT USING can do. See how \ and # can be mixed with blank spaces on the same image Line?

Remember, DATA items are easy to change. If you want to use the current rate of exchange, just enter them in place of these.

Count spaces in Line 100 *very carefully!* Add a "measuring Line" 99 if necessary.

```
10 REM  * INTERNATIONAL MONEY CHANGER *
20 REM  * USING SAMPLE RATES OF EXCHANGE *
30 RESTORE
40 PRINT "HOW MANY U.S. DOLLARS";
50 INPUT "DO YOU WISH TO EXCHANGE";D
60 CLS
70 PRINT "AT TODAY'S RATE YOU WILL GET"
80 PRINT
90 READ A$,A : IF A$ = "END" THEN 30
100 P$ = "\      (16 spaces)   \    ########.##"
110 PRINT USING P$;A$;D/A
120  GOTO 90
1000 DATA ARGENTINE PESO, .02576
1010 DATA AUSTRALIAN DOLLAR, .9025
1020 DATA AUSTRIAN SCHILLING, .05179
1030 DATA BELGIAN FRANC, .01786
1040 DATA BRAZILIAN CRUZEIRO, .0006809
1050 DATA BRITISH POUND, 1.3830
1060 DATA CANADIAN DOLLAR, .7710
1070 DATA CHINESE YUAN, .4523
1080 DATA COLOMBIAN PESO, .01034
1090 DATA DUTCH GUILDER, .3210
1100 DATA DANISH KRONE, .09930
1110 DATA ECUADORIAN SUCRE, .01075
1120 DATA FINNISH MARKKA, .1715
1130 DATA FRENCH FRANC, .1185
1140 DATA GREEK DRACHMA, .009217
1150 DATA HONG KONG DOLLAR, .1279
1160 DATA INDIAN RUPEE, .0904
1170 DATA INDONESIAN RUPIAH, .000991
1180 DATA IRISH PUNT, 1.1150
1190 DATA ISRAELI SHEKEL, .005528
1200 DATA ITALIAN LIRA, .0005910
```

```
1210 DATA JAPANESE YEN, .004305
1220 DATA LEBANESE POUND, .1751
1230 DATA MALAYSIAN RINGGIT, .4322
1240 DATA MEXICAN PESO, .004902
1250 DATA NEW ZEALAND DOLLAR, .6470
1260 DATA NORWEGIAN KRONE, .1277
1270 DATA PAKISTANI RUPEE, .07246
1280 DATA PERUVIAN SOL, .0003378
1290 DATA PHILLIPPINE PESO, .07133
1300 DATA PORTUGUESE ESCUDO, .007143
1310 DATA SAUDI ARABIAN RIYAL, .2841
1320 DATA SINGAPORE DOLLAR, .4739
1330 DATA SOUTH AFRICAN RAND, .7800
1340 DATA SPANISH PESETA, .006494
1350 DATA SWEDISH KRONA, .1236
1360 DATA SWISS FRANC, .4415
1370 DATA TAIWANESE DOLLAR, .02526
1380 DATA THAI BAHT, .04351
1390 DATA URUGUAY NEW PESO, .01965
1400 DATA VENEZUELAN BOLIVAR, 1.333
1410 DATA WEST GERMAN MARK, .3648
1500 DATA END, 0
```

As we've seen, PRINT USING is the most complex of our PRINT statements but by far the most powerful. If you're a serious programmer, you should master PRINT USING completely. Then you can take our many simple learning examples and expand them into large, useful routines.

Learned In Chapter 37

Miscellaneous

PRINT USING symbols
+ – ^ !

Chapter 38

Using A Printer

Ready for a break to learn something that's very simple?

LPRINT And LLIST

These BASIC Commands/Statements are almost too easy.

Our Chapter is written for the Imagewriter because it was designed specifically for the Mac, but Mac will support the new Laserwriter and other serial printers if a special driver is purchased from your Apple dealer.

We have learned a lot of ways to PRINT, but they have all been on the video screen. Now we'll learn how to PRINT-out to the Imagewriter. If you don't have one, at least skim this Chapter before proceeding.

Hook up and turn on the printer, then type this new one-Line program:

```
LPRINT "THE PRINTER WORKS!!!"
```

Notice that the first word is LPRINT, not PRINT. Run the program.

Did it print? If your printer did nothing, check the connections again. Make sure the printer is *on*, the SELECT light is lit and the covers are in place. Try Running the one-Line program again.

NOTE: There is much widespread misuse of the language when it comes to naming printers. Here are some definitions:

PRINTer = a device which converts computer talk to "hard copy."

Dot Matrix Printer = A printer such as the Apple Imagewriter which creates characters and graphics by printing clusters of dots.

Character Printer = A printer which, like most typewriters, prints complete pre-formed characters.

Line Printer = A very large "hi-speed" printer which literally "sets" and then prints an entire LINE of print at one time.

There is much misnaming of printers. Very few are true "Line Printers," though many are sold under that name. True Line Printers are very expensive and can print over 1000 lines of type per minute.

It is from the Line Printer name that the "L" in LPRINT was derived.

LLISTing The Program

LLIST is typed in the Command window when we want a LISTing of a program sent to the PRINTer.

Both LPRINT and LLIST can be used either as statements or commands. If you want to PRINT both on the screen and on paper, use duplicate program Lines, with PRINT for the screen and LPRINT for the PRINTer.

To print a listing as it appears on the screen with the same spacing and font styles, select Print... from the File menu. A paper option window appears that lets us choose various paper sizes and styles. You can change the settings or leave them as is and continue by clicking the **OK** box. The next window lets us select the quality of printing and the Page Range. The Page Range will come in handy with very long programs that fill more than one printed page. Clicking inside the **OK** box in this window starts the printing.

Enter any program of your choice, and convert it to LPRINT the results on your PRINTer. Make a "hard copy" LLISTing of it.

If you accidentally precede either PRINT or LIST with the letter L and don't have a PRINTer connected, there may be trouble. It's very easy to accidentally turn a simple LIST into LLIST. If there is no PRINTer hooked up or it's turned OFF, LPRINT and LLIST have no effect. However, if the printer is turned on and the SELECT light is off, the Computer freezes until the SELECT switch is pressed.

LPRINT TAB

The TAB function can handle numbers up through 32767. This has little value

in displays PRINTed on the Computer, but on big PRINTers, it is common to PRINT Lines up to 132 characters long.

We recall that PRINT STRING$ is used to repeat a number of characters or actions. We can use it to sneak around the above rule by having it repeat a number of spaces. For example:

```
LPRINT STRING$(95,32);X
```

will "PRINT" 95 blank spaces before PRINTing the value of X. "32" is the ASCII code for a space.

Formatting

Now, let's see how to PRINT with a nice format. Select New, and type:

```
10 FOR X=1 TO 100
20 LPRINT X,
30 NEXT X
40 LPRINT
```

...Run it.

See how the printer will format the PRINTing into neat little columns? The comma with LPRINT works the same as it does with PRINT.

Try using a semi-colon in Line 20 rather than a comma. Type:

```
20 LPRINT X;
```

...and Run. The semi-colon works the same on the PRINTer as it does on the video screen. Let's see how TAB works. Type this New program:

```
10 LPRINT TAB(25); "TELEPHONE LIST"
20 LPRINT
30 LPRINT TAB(15); "NAME";
```

```
40 LPRINT TAB(45)¦ "TELEPHONE NUMBER"
50 LPRINT
60 INPUT "TYPE A FRIEND'S NAME"¦A$
70 INPUT "PHONE NUMBER"¦B$
80 PRINT "THANK YOU"
90 LPRINT TAB(15)¦ A$¦TAB(45)¦ B$
100 INPUT "IS THERE ANOTHER FRIEND (Y/N)"¦Q$
110 IF Q$="Y" THEN 60
```

...and Run.

If the paper width is smaller than 8 1/2 inches, you'll want to use different TAB settings.

LPRINT USING

In the last Chapter we saw how PRINT USING can format our PRINT outputs on the screen. Those same features can be applied to the printer by using LPRINT USING. Incorporate LPRINT USING in one of the simple programs from the last Chapter.

```
10 X$ = "ALEXANDER"
20 Y$ = "GRAHAM"
30 Z$ = "BELL"
50 LPRINT USING "!1!1\ 2 \"¦X$¦Y$¦Z$
```

...and Run.

Advanced LPRINT Capabilities

The printer is capable of producing all different kinds of printouts. By sending the printer certain codes, it can be made to display graphics, underline text, make boldfaced letters, etc. We have listed just a few of the popular ASCII codes. See the Imagewriter Printer Manual for more.

8	backspace
9	horizontal tab
10	line feed and carriage return
12	roll paper to top of next sheet
13	carriage return
14	begin headline mode
15	end headline mode

To see what this all means, enter this New program:

```
10 INPUT "ENTER A CODE NUMBER";N
20 LPRINT N;"IS ";CHR$(N);" TO A PRINTER"
30 GOTO 10
```

...and Run.

Try each of the codes, and see what happens. Some codes may do nothing. If something goes wrong, shut the printer off and turn it back on to "clear" it out.

The "top of form," or "top of next sheet," feature is necessary when using the printer with preprinted forms or when printing must start at the top of the page.

When your Computer is turned on, if it is going to do any PRINTing, it automatically assumes it will be PRINTing 6 lines per inch on sheets of paper 11 inches long, 66 lines per page.

Screen Dumps

By holding ⌘ Shift 4 with the Caps Lock down, the Macintosh will dump the entire screen to the printer. If the Caps Lock key is not down, only the active window will be printed.

LCOPY

LCOPY does what pressing ⌘ Shift 4 does, but from within a program.

With a little experimenting, your PRINTer will be doing what you paid to have it do.

Learned In Chapter 38

Statements	Miscellaneous	Menu
LPRINT	Trailing semi-colon	File
LLIST	Screen dump (⌘ Shift 4)	Print...
LCOPY		

PART 6

ARRAYS

Chapter 39

Arrays

We know we can use combinations of the 26 letters of the alphabet, digits 0-9, and the decimal points to create variable names of up to 40 characters in length. We've also discovered that very few of our programs have required anywhere near that many variables. There are times, however, when we need more variables -- sometimes *hundreds* or even *thousands* of them.

The way we control and keep track of that many variables is by holding them in an ARRAY. Array is just another word for "lineup," "arrangement" or "series of things."

Let's organize a collection, arrangement or lineup (array) of autos, each of which has a different I.D. (address) number.

We line up 10 cars, as in an *array*. They are all the same except for their engine size -- and each has a different I.D. or license number. Let's say the I.D. numbers range from 1 to 10, and we want to use the Computer to quickly spit out the engine size when we identify a car by its I.D. number. This might not seem like a real heavyweight problem -- but, as before, we discover the full potential of these things by learning little steps at a time.

The I.D. numbers and engine sizes are as follows:

CAR #	ENGINE
1	300
2	200
3	500
4	300
5	200
6	300
7	400
8	400
9	300
10	500

Now, we could give each of these cars a different letter name, using the variables A through J, but what a waste -- and what will we do when there are a thousand cars, not just ten?

Setting Up Arrays

Microsoft BASIC allows any valid variable name to be used as an array name. An *Array* named "A" is not the same as the *Numeric* variable "A," and neither is it the same as *string* variable A$. It is a totally separate "A" used to identify a *Numeric array*. We call it A-sub(something), and it can only hold numbers. We will name the cars A(1) through A(10), pronounced "A sub 1" through "A sub 10." Get the idea?

What's that -- you don't believe there can be 3 separate variables all named "A"? Ok, in the Command window type:

```
A = 12          Return
A$ = "(YOUR NAME)"          Return
A(1) = 999          Return
```

then: `PRINT A,A$,A(1)` Return

Does that make you a believer?

Let's store the car engine sizes in DATA statements. Return to the List window, and type:

```
500 DATA 300,200,500,300,200
510 DATA 300,400,400,300,500
```

Notice how carefully we kept the DATA elements in order from 1 to 10 so the first car's engine size is found in the first DATA Location, and the 10th one's in the 10th location?

We now have to "spin up" an array inside the Computer's memory to make these data elements *immediately addressable.*

Big words meaning "so we can find a car fast."

Think how difficult it would be to try to address the 7th engine (or the 7 thousandth!), for example, using only what we've learned so far. It *can* be done using only DATA, READ and RESTORE statements, but that would be very messy and slow.

The easy way to create the array is to insert:

```
20 FOR L = 1 TO 10
30  READ A(L)
40 NEXT L
```

...and Run.

Nothing happen? Yes, it did. We simply didn't display what happened.

The FOR-NEXT loop READ 10 pieces of DATA and named the elements (or "cells") in which they're stored A(1) through A(10). To PRINT out the values in those array elements, add:

```
95 FOR N = 1 TO 10
100  PRINT A(N)
110 NEXT N
```

...and Run.

Aha! It works, but how? We READ the DATA elements into an array called A(L), but PRINTed them out of an array called A(N). Why the difference? Nothing significant.

The array's NAME is "A." The *location* of each data element within that array is identified by the number we place inside the parentheses. That number can be brought inside the parentheses by using any numeric variable and even some simple arithmetic can be done inside the parentheses, if necessary. We arbitrarily used N to READ them in and L to PRINT them out.

Remember, the array we are using is named "A." Its elements are numbered and called A-sub(number).

Some pure mathematicians might insist on calling A(X) A "OF" X. We don't need that added confusion. Best you know, just in case.

Let's work some more on the program.

Insert, and change:

```
80 PRINT
90 PRINT "CAR#","ENGINE SIZE"
100 PRINT N,A(N)
```

...and Run.

Now that's more like it. We have every I.D. number, every engine size and are not "using up" any of the "regular" alphabetic variables to store them. Having demonstrated that point, remove Lines 95 and 110, and insert:

```
10 INPUT "WHICH CAR'S ENGINE SIZE";W
100 PRINT W,A(W)
```

...and Run, answering with a car #.

Get the idea? Can you see the crude beginning of a simple inventory system for a small business?

Let's go one small step (for mankind) further. Suppose we know the color of each of the 10 cars, and for simplicity, suppose the colors are coded 1, 2, 3 and 4. We might then have a master chart that looks like this:

CAR#	ENG. SIZE	COLOR
1	300	3
2	200	1
3	500	4
4	300	3
5	200	2
6	300	4
7	400	3
8	400	2
9	300	1
10	500	3


HEH, HEH. ALL I ASKED FOR WAS ARRAYS.

In the language of professional computer types, this is called a *matrix*. A *matrix* is just an array that has more than one dimension. (In our arrays, we can have at most 255 dimensions with a maximum of 32,768 elements. Our first array had the dimension of 1 by 10 -- 1 *column* by 10 *rows*.) This new array has a horizontal dimension of 2 and a vertical dimension of 10.

If we wanted to be terribly *in*efficient about the matter, we *could* say that this is a 3 by 10 array, counting the I.D. number. If so, our first example would be called a 2 by 10 array -- but who needs it? As long as we keep the I.D. numbers in a simple 1 to 10 FOR-NEXT loop and the DATA in proper sequence, the arrays will be simple and easier to handle.

Since we do not store the car number in the Computer it is a "pointer" or an "index." That's why we don't consider it as another "DIMension" to the matrix.

How then can we label this 2 by 10 *matrix*? We have already used up our A array elements numbered 1 through 10. Oh, you want to know how many elements we have to work with? Very good!

Let's arbitrarily assign array locations 101 through 110 to hold the color code. We also have to put the color code info in the program using a DATA statement. From the table, type:

```
520 DATA 3,1,4,3,2,4,3,2,1,3
```

and insert:

```
50 FOR S = 101 TO 110
60  READ A(S)
70 NEXT S
```

These last Lines load the color code DATA into the array. Array element numbers 11 through 100 are not used, nor are those from 111 to the end of memory since they have not been formally assigned any values.

...Run, and select any car number.

Awwk!! What is this "Subscript out of range" business? Well, since arrays take up a lot of memory space, the Computer automatically allows us to use up to only 11 array elements without question. (They can be numbered from 0 to 10.) Then our credit runs out. We earlier used elements numbered from 1 to 10 without any problem.

If we'd wanted to, we could have put at the beginning of our program:

```
5 OPTION BASE 1
```

This OPTION changes the lowest (or BASE) array element number to 1, instead of 0. 1 or 0 are the only numbers that can be used with the OPTION BASE statement.

To use array elements numbered beyond 10 in the array called "A," we have to "reDIMension" the available array space. Our highest number in Array "A" needs to be 110, so we'll add a program DIMension statement:

```
5 DIM A(110)
```

...and Run again. That's better, but it's not PRINTing the color code.

To display all the information, change these Lines:

```
10 INPUT "WHICH CAR TO EXAMINE";W
90 PRINT "CAR#","ENG. SIZE","COLOR"
100 PRINT W,A(W),A(W+100)
```

...then Run.

Check your answers against the earlier master matrix chart. Save the program As CARARRAY.

Let your imagination go. Can you envision entire charts and "look-up" tables stored in this way? Entire inventory lists? How about trying to *find* the car which has a certain size engine *and* a certain color? Hmmm. We will come back to the Logic needed for that last one.

EXERCISE 39-1: Assume that your inventory of 10 cars includes 3 different body styles, coded 10, 20 and 30, as follows:

CAR#	BODY
1	20
2	20
3	10
4	20
5	30
6	20
7	30
8	10
9	20
10	20

Modify the resident program to PRINT the body style information along with the rest when the car is identified by I.D. number.

A Smith & Wesson Beats 4 Aces

If we want to create a computerized card game (they make good examples to show so many things), how can we program it so it draws the 52 or so (watch the dealer at all times) cards in a totally random way? **ANSWER:** Spin up the deck into a single-dimension array, pick array elements using a random number generator, as each card is "drawn" set its array element value equal to zero, then test each card drawn to be sure it isn't zero. Now that is *really* simple! (Might want to read it once again, more slowly.)

We will now, a step at a time, write a program which will draw, at random, all 52 cards numbered from 1 through 52 and PRINT the card numbers on the screen as they are drawn. No card will be drawn more than once. When all cards have been drawn, it will PRINT "END OF DECK!"

You do a step first, then check against my example. Then change yours to match mine -- otherwise we might not end up at the same place at the same time.

STEP 1: Spin up all 52 cards into an array.

```
10 WIDTH 60
20 DIM A(52)
30 FOR C=1 TO 52 : READ A(C) : NEXT C
500 DATA 1,2,3,4,5,6,7,8,9,10,11,12,13
510 DATA 14,15,16,17,18,19,20,21,22,23
520 DATA 24,25,26,27,28,29,30,31,32,33
530 DATA 34,35,36,37,38,39,40,41,42,43
540 DATA 44,45,46,47,48,49,50,51,52
```

At this point, all we can tell when Running is that processing time is required since the cursor (flashing bar) doesn't come back right away.

Shhhh! I know there's a shorter way to program this special case, but it doesn't teach what's needed.

STEP 2: Draw 52 cards at random, PRINTing their values.

```
40 FOR N = 1 TO 52
50 V = INT(RND * 52 + 1)
60 PRINT A(V);
70 NEXT N
```

...and Run.

True, 52 card values are PRINTed on the screen, but if we look carefully, the same number appears more than once. This means that some "cells" are not being READ and some READ more than once.

STEP 3: When a card is drawn, set its array value equal to 0. Test each card drawn to be sure it is not 0. When 52 cards have been drawn and PRINTed, PRINT "END OF DECK!"

```
40 P = 52
55 IF A(V) = 0 GOTO 50
70 A(V) = 0 : P = P - 1
80 IF P<>0 GOTO 50
90 PRINT : PRINT "END OF DECK!"
```

...and Run.

Line 70 sets the value in cell A(V) equal to 0 only if Line 55 finds it *not* equal to 0 already, letting the program pointer fall through.

When a "fall through" occurs:

1. the card's value is PRINTed (Line 60).

2. the number stored in that cell is set to 0 (Line 70).

3. the second statement in Line 70 counts down the number of cards PRINTed. Line 40 initialized the number of PRINTs at 52.

4. the number of PRINTs is tested (Line 80). When there are no more PRINTs to go, "END OF DECK!" is PRINTed (Line 90).

Pretty slick -- and we don't have to watch the dealer (just the programmer).

But how do we really know that every card has been dealt? Write a quick addition to the program to "interrogate" each array cell and PRINT its contents.

```
100 FOR T = 1 TO 52
110   PRINT A(T);
120 NEXT T
```

Run ... and every cell comes up zero. If you don't really trust all this, change Line 40 to read:

```
40 P = 50
```

...Run and see what happens.

AHA! It flushed out those 2 cards up the sleeve, didn't it?

To add a final touch of "randomness" to the deal, add:

```
5 RANDOMIZE TIMER
```

Change P back to 52, Delete test program Lines 100, 110, and 120, and we end up with a good card-drawing routine. You might want to clean it up to your satisfaction and Save it As CARDDRAW for future projects.

Question: Why does the PRINTing of card numbers slow down to a near halt as those last few cards are being drawn? Is the dealer reluctant?

Answer: The random number generator has to keep drawing numbers until it hits one that is the array address of an element which has *not* been set to zero. Near the end of the deck, almost all elements have been set to zero. The random number generator has to draw numbers as fast as it can to find a "live" one.

Look again at the card numbers PRINTed. There will not be any duplication. No stray aces.

> **EXERCISE 39-2:** Change the program so the original array can be loaded with the card numbers without having to READ them in from DATA Lines.

New Dimensions

We have already done some DIMensioning with single dimension *numeric* arrays. *String* arrays must also be DIMensioned.

Suppose we have a program like this: (Type it in.)

```
10 FOR N = 1 TO 15
20  READ A$(N)
30  PRINT A$(N),
40 NEXT N
```

```
100 DATA ALPHA,BRAVO,CHARLIE,DELTA
110 DATA ECHO,FOXTROT,GOLF,HOTEL
120 DATA INDIA,JULIETTE,KILO,LIMA
130 DATA MIKE,NOVEMBER,OSCAR
```

...and Run.

Oops. There's that same problem. "Subscript out of range" means "not enough space set aside for an array." Recall that only 11 elements *per array* (from 0-10) are set aside on power-up. We are trying to read in 15 of them, starting with 1. The solution:

```
5 DIM A$(15)
```

...and Run.

DIMensioning a string array is just like dimensioning a numeric one -- simply call it by its name. In this case, its name is A$. You "high speed" types will want to know that to do "dynamic redimensioning" (that's doing it while a program is running), the program must first encounter a CLEAR. Oh.

All CLEAR

The CLEAR statement simply CLEARs the memory of all meaningful information except the actual program. It makes *all* string variables and arrays contain nothing and sets *all* numeric variables to 0. And anything we DEFined with a DEF FN statement will be forgotten.

For example, enter the Command window, and type:

```
CLEAR          Return
```

and then:

```
PRINT A$(3)          Return
```

Nothing. Typing RUN Return to start the program again and to reload the Array, then PRINT A$(3), and we get CHARLIE.

ERASE will null out the contents of a *specific* array variable.

For example, type:

```
ERASE B$          Return
```

By telling the Computer to ERASE all data in the B$ array, we have not removed the data in A$ array. (Since we don't have a B$ array in our program, we'll get an "Illegal function call.") To prove this point type:

```
PRINT A$(3)       Return
```

Now type:

```
ERASE A$          Return
```

and

```
PRINT A$(3)       Return
```

Try PRINTing other elements in A$ array. They have *all* been ERASEd.

Array Names

```
A(N)
BC(N)
D3(N)
E4$(N)
XY$(N)
A.SAMPLE.ARRAY$(N)
```

are examples of legal array names. The last 3 are for "string arrays."

Learned In Chapter 39

Statements

DIM
CLEAR
ERASE
OPTION BASE

Miscellaneous

Arrays
Array names

Chapter 40

Search And Sort

One of the Computer's most powerful features is its ability to *search* through a pile of DATA and *sort* the findings into some order. Alphabetical, reverse alphabetical, numerical from smallest to largest, or the reverse are all common sorts. The *search/sort* feature is so important we will spend this entire Chapter learning how to use it.

Typical applications of *search* and *sort* include:

1. Arranging a list of customers' or prospects' names in *alphabetical* order.

2. Sorting names in *ZIP-Code* order for lower-cost mailing.

3. Sorting the names of clients in telephone *area code* order.

While not really all that complicated, the sorting process is sufficiently rigorous that we are going to take it *very slowly* and examine each step. Once we get the hang of it, the Computer can blaze away without our considering the staggering number of steps it's going through.

A Problem Of Sorts

Let's start with a problem. We have the names of 10 customers. (If that doesn't grab you, make it 10 million -- the process is identical.) We wish to arrange them in alphabetical order.

Start by storing their names in a DATA Line. Select the List window and type in:

```
DATA BRAVO,XRAY,ALPHA,ZULU,FOXTROT,TANGO,
     HOTEL,SIERRA,MIKE,JULIETTE
```

Since we are sorting by *name* rather than by number, we have to use *string* variables, *string* arrays, etc. They work equally well with numbers such as zip codes, while numeric variables and arrays work *only* with numbers.

The backbone of a *sort* routine is the array. Each name is to be READ from DATA into an array. So add:

```
10 REM  * ALPHA SORT OF STRINGS FROM DATA *
20 FOR D = 1 TO 10 : READ A$(D) : N=N+1 :
   NEXT D
```

Line 10 is, of course, just the title.

Line 20 "loads the array" by READing the 10 names into storage slots A$(1) to A$(10). N is simply a counter which will follow through the rest of the program. In this simple program, we could have made N=10 since we know how many names we have. In the next sample program, we won't know how many names there are, so let's leave N the way it's usually used.

Important to the *sort* routine are 2 nested FOR-NEXT loops.

1. The first one, F, controls the First name.

2. S, the second one, controls the name to be compared against the first.

Names and words are compared as we learned in the Chapter on ASCII set, remember?

Let's establish the loops first, and fill in the guts later:

```
30 FOR F = 1 TO N-1     (F=First word to be compared)
40   FOR S = F+1 TO N   (S=Second word to be compared)
90   NEXT S                          (Makes 9 passes)
100 NEXT F                           (Makes 9 passes)
```

It may seem puzzling that F and S only have to make 9 passes when there are 10 names. Think of it this way. Whatever word *isn't* smaller (ASCII #) than the rest, just ends up last. No need to test again to prove that.

The F loop READs array elements 1 through 9 (N-1 = 9). The S loop READs array elements 2 through 10. This always provides *different* array elements to compare.

Now we'll jump to the end of the program and prepare it to PRINT out what will happen. Type:

```
110 WIDTH 60,12
120 FOR D = 1 TO N : PRINT A$(D), : NEXT D
```

When the *sort*ing is done, the contents of A$(1) to A$(10) will be the same names READ from DATA, but they will be in alphabetical order. We'll PRINT the array contents on the screen.

```
50    IF A$(F) <= A$(S) THEN 90   (tests for smaller ASCII#)
60    T$ = A$(F)          (first word to Temp storage)
70    A$(F) = A$(S)         (copy Second word to First place)
80    A$(S) = T$            (copy Temp word to Second place)
```

And there is the biggie! If you can understand the last 4 Lines, the rest is duck soup.

Line 50 says, "If the First word is smaller than (or equal to) the Second word, leave well enough alone and bail out of this routine by going to Line 90, which will end this pass and READ another word to compare against F. If it is larger, drop to the next Line."

Line 60 says, "Oh, they weren't in the right order, eh? We'll just copy the First word in a Temporary storage location called T$ and store it there for future use. I'm sure we'll need it again."

Line 70 copies the name held in the Second cell into the First array cell. If the Second one had an earlier starting letter than the First one, we do want to do this, don't we?

Line 80 completes the switch by copying the name Temporarily stored in T$ into the Second array cell. A$(1) and A$(2) contents have now been exchanged with the aid of the Temporary holding pen, T$.

Us simple country boys find this one easy: *There are two brahma bulls in separate pens, A$(1) & A$(2), and we want to switch them around. Ain't no way we're going to put them in the same pen at the same time. (Not with me in there anyway. Already broken too many 2 by 4's between their horns and have some scars on the wrong end from escapes that were a hair too slow.) That's why we built a temporary holding pen called T$.* Got it?

If we did everything right, the program should:

```
Run
```

and in a flash the names appear on the screen in alphabetical order:

```
ALPHA     BRAVO     FOXTROT     HOTEL     JULIETTE
MIKE      SIERRA    TANGO       XRAY      ZULU
```

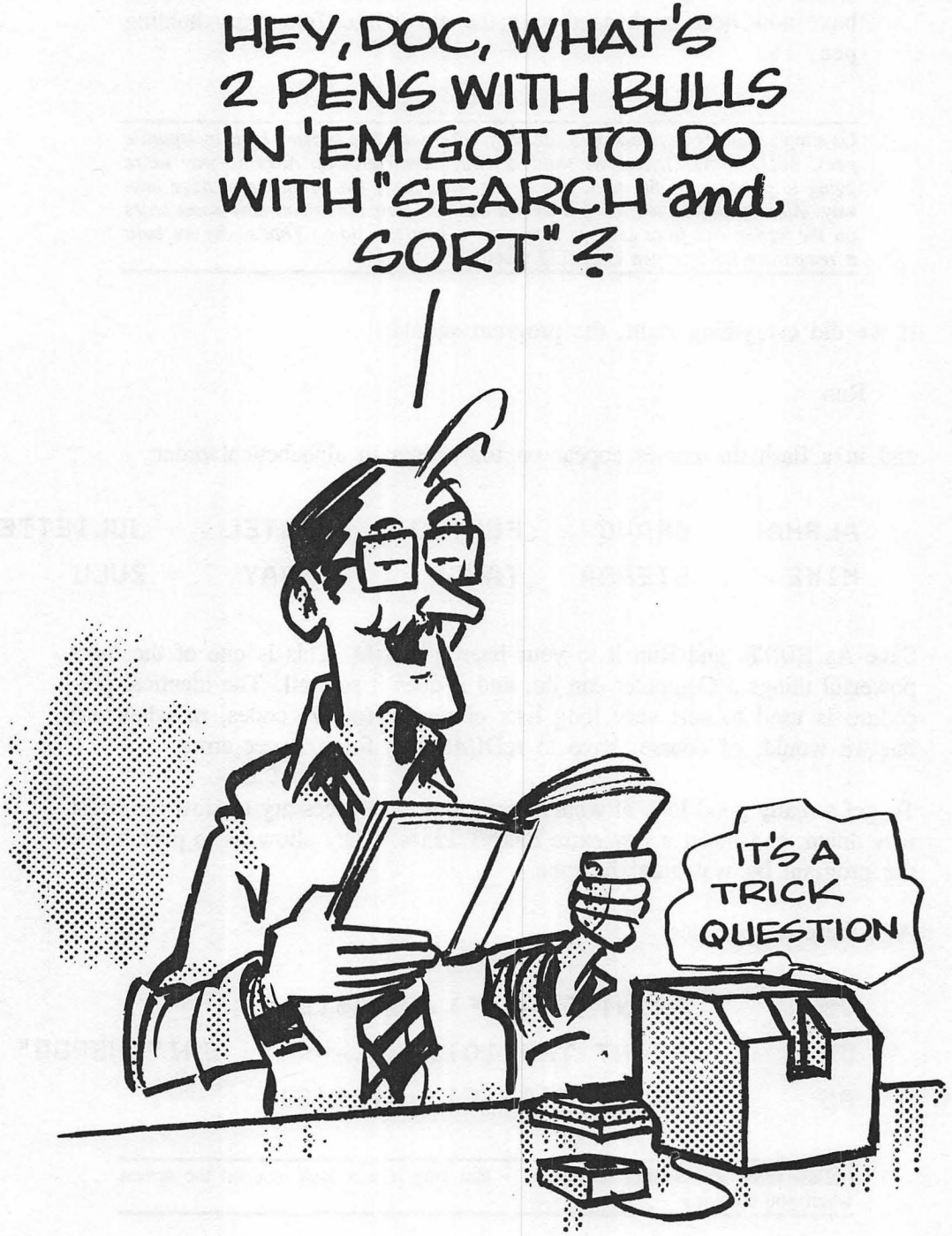
Save As SORT, and Run it to your heart's delight. This is one of the most powerful things a Computer can do, and it does it so well. The identical procedure is used to sort very long lists of names (or zip codes, or whatever) but we would, of course, have to reDIMension for a larger array.

To get a really good look at what's happening, it's necessary to slow the beast *way* down, and insert a few extra PRINT Lines. They allow us to peer inside the program by watching the tube.

Add these temporaries:

```
45        PRINT F;A$(F),,S;A$(S)
55        PRINT TAB(10); "<<--<<    SWITCHEROO"
85        PRINT F;A$(F),,S;A$(S)
```

(Allow three spaces after the arrow -- that way it will look nice on the screen when you Run it.)


HEY, DOC, WHAT'S 2 PENS WITH BULLS IN 'EM GOT TO DO WITH "SEARCH and SORT"?
IT'S A TRICK QUESTION

...and Run.

Aw c'mon horse -- Whoa!

If that wasn't slow enough, add Line 47 and make the delay long enough so there is time to completely think through each step. Pretend you're the Computer, and make the decision that Line 50 has to make. Take it from the top -- very slowly!

```
47        FOR Z = 1 TO 1000 : NEXT Z
```

The Diagnosis

```
1 BRAVO                                    2 XRAY
```

means "in cell #1, is the word BRAVO, and in cell #2, is the word XRAY" just like they came from the DATA Line. Of those two words, BRAVO is the "smallest" (ASCII#), so it stays in number 1 place. On to the next pass of S...

```
1 BRAVO                                    3 ALPHA
```

Oops. BRAVO is in #1 and ALPHA is in #3, but ALPHA is smaller than BRAVO. We better switch them around. So

```
                    <<--<<      SWITCHEROO
```

```
1 ALPHA                                    3 BRAVO
```

Don't worry too much about what is happening in the second column. S is scanning through the array, and its contents are always changing, testing against what's in the first. It's what *ends up* in the *first* column that counts -- and that list must be in increasing alphabetical order.

As the program Runs, watch new words appear in S, loop and column, and compare them against what's in F. Try to guess what the Computer's going to do. Also keep an eye on the increasing numbers on the left. The *final word* assigned to a given number in the first column is what will appear in the final PRINTout.

Run the program as many times as it takes (and at as many sessions as it takes) to completely understand what's happening. It's awfully clever, very important and absolutely fundamental. We carry this technique over to many useful programs in the future, but only if we *really* understand it.

When you feel it's under control, add one more little item to the screen. What T$ is holding while all this *sort*ing is going on is interesting. Add and change these Lines so they read:

```
45    PRINT F;A$(F),,S;A$(S),"T$ = ";T$
85    PRINT F;A$(F),,S;A$(S),"T$ = ";T$
```

...and Run.

"T$ = " starts off empty since there is nothing in the holding pen. BRAVO is replaced by ALPHA in the switching process; however, T$ holds it. When BRAVO replaces XRAY in the #2 position, T$ holds XRAY, etc.

On a clear head it's not hard to follow what's happening. If you're tired, it's hopeless. Save this program and review it as often as necessary for a deep understanding of the process.

Sorting From The Outside

We don't really have to keep all our names, numbers or other information in DATA Lines. It can be INPUT from the keyboard or from disk. The following program is quite similar to the resident one, and the logic is identical. Change and add these Lines:

```
10 REM  * ALPHA SORT OF NAMES VIA INPUT *
20 INPUT "NEXT NAME";N$ : IF N$="END" GOTO 30
25 N=N+1 : A$(N) = N$ : GOTO 20
```

Delete the DATA Line.

...and Run.

INPUT 6 or 8 random names, and when finished, INPUT the word "END." The sort process is identical to what we used before.

Can you see the potential for all this?

> **EXERCISE 40-1:** Change Line 50 of the sort program to list the names in reverse alphabetical order.

Learned In Chapter 40

Miscellaneous

Sorting

Chapter 41

Multi-DIMension Arrays

We have learned that an array is nothing more than a temporary parking area for lots of numbers, or characters, or both. In addition, we learned that it is a straight-forward procedure to compare values of variables outside the matrix (or array) with those inside it.

An array which only has one DIMension, that is, just one long line-up of parking places is sometimes called a *vector*. We can take that one-dimensional array and cut it into perhaps four equal chunks, and position those chunks side by side. We then call it a two-dimensional array -- since the parking places are lined up in *rows* and *columns* (or *streets* and *avenues*). Its DATA holding or processing abilities are not changed. Only the *addresses* of the parking places (or elements or memory cells) have changed.

Type in this New program:

```
10 DIM M(40)
20 FOR V = 1 TO 40
30  PRINT V,M(V)
40 NEXT V
```

Remember, any array with more than 11 elements (counting 0) must be DIMensioned.

...and Run.

The Run simply shows the addresses (numbers) of 40 storage positions and their contents. Since they are all lined up in a single row, it is a vector array.

Why are the cell contents always 0? Because every cell value is initialized at zero upon entering BASIC and whenever we Run, just like all other numeric and string variables. Line 30 shows how easy it is to specify the *address* and read the *contents* of each memory cell.

Side By Side

Let's cut our 40 cell array into 4 equal strips and line them up side by side. That would make 10 *rows* each containing 4 cells ... right? Or 4 *columns* each containing 10 cells. "Multi-dimensional arrays" always have *rows* and *columns*.

Start over with this New program:

```
10 DIM M(10,4)          (10 rows by 4 columns)
20 FOR R = 1 TO 10
30  FOR C = 1 TO 4
40   PRINT R;C,
50  NEXT C : PRINT
60 NEXT R
```

Save As MATADR; then after closing the List window, Run.

The *addresses* of all 40 cells displayed on the screen at the same time, but not their contents. Nothing was changed from the earlier vector array containing the same 40 cells. We just rearranged the furniture and gave it different addresses. They read:

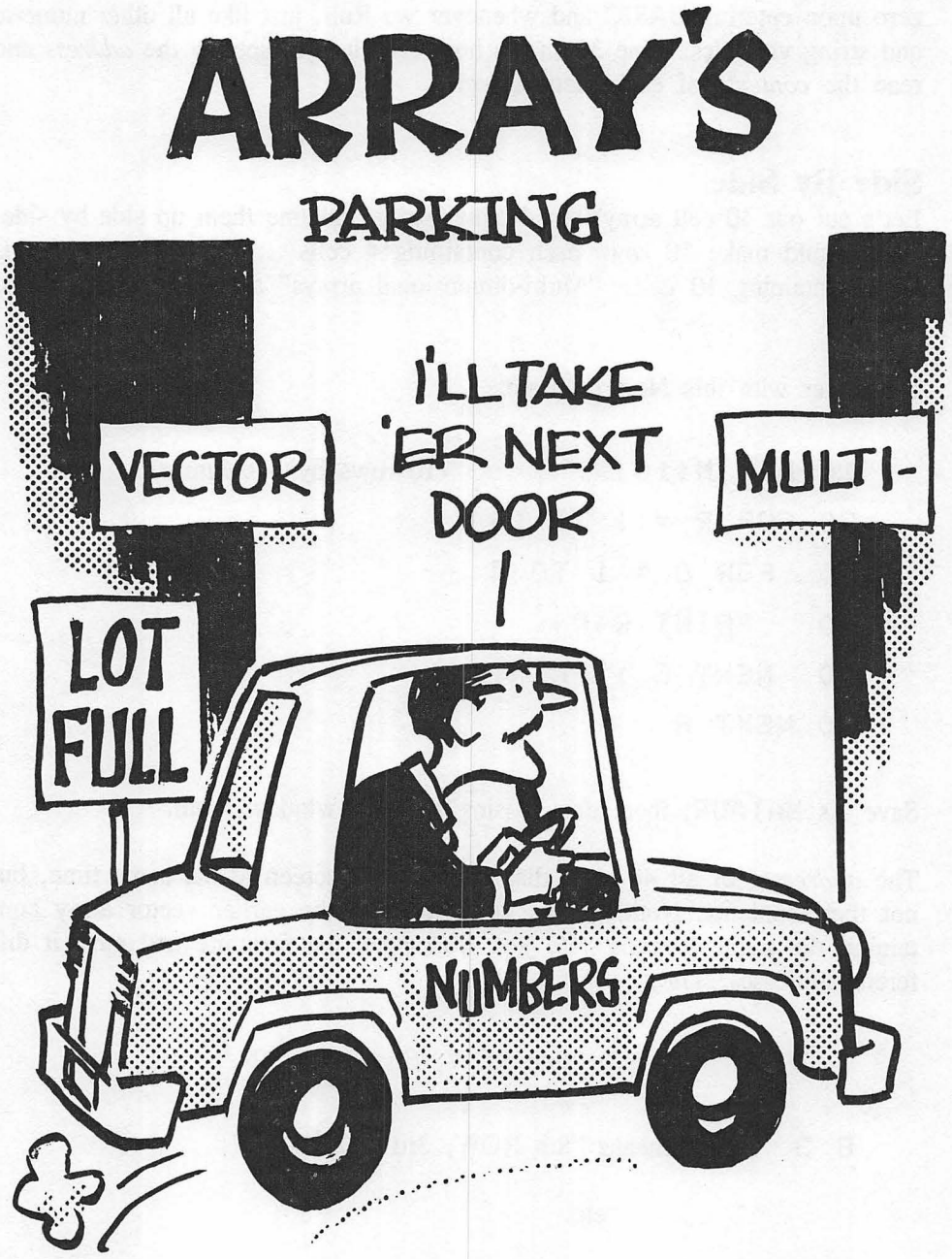
```
1  1        means "first ROW, first COLUMN."

8  3        means "8th ROW, 3rd COLUMN."

                  etc.
```

To view the *contents* of each of these cells, change Line 40:

```
40     PRINT M(R,C),
```


ARRAY'S
PARKING
VECTOR
I'LL TAKE 'ER NEXT DOOR
MULTI-
LOT FULL
NUMBERS

...and Run.

See, the contents remain unchanged. They are still at their initialized value of 0, since we made no arrangement to store information in them. (The *addresses* are no longer displayed.) Isn't this easy (...so far)?

Memory cells, like any other variables, have to be "loaded" with values to be useful. This can be done by READing in DATA from DATA Lines, by INPUTting it via the keyboard or from a previously recorded DATA disk. We will load our Matrix from DATA Lines imbedded in the program.

Add these Lines:

```
100 DATA 1,2,3,4,5,6, etc. to 14
110 DATA 15,16,17,18,19, etc. to 28
120 DATA 29,30,31,32,33, etc. to 40
```

and this Line to READ the DATA into matrix cells:

```
35   READ M(R,C)
```

Save As MATCONT and Run.

The DATA is nicely arranged in the matrix, and each matrix position has its original specific address. Again, that address is not displayed -- just the contents. Let's go to the Command window for a minute and "poll," or "interrogate," several matrix positions to see what they are holding. Ask:

```
PRINT M(2,3)          Return
```

Write down 7, the answer. We'll Run the program again later and check it.

```
PRINT M(9,4)          Return
```

Says *that* cell holds the number 36.

```
PRINT M(3,6)          Return
```

"Subscript out of range"? Why did we get that? Oh, there is no column 6? No wonder.

Run the program again and check the screen, counting down the Rows and over the Columns to see if the answers match up.

Mine did -- how about yours?

```
Row 2 Col. 3 = 7
Row 9 Col. 4 = 36
```

As an aside, in the Command window, type:

```
ERASE M          Return
```

then, check any matrix cell again.

```
PRINT M(2,2)          Return
```

and get 0. ERASE M re-initialized *all* cells of array M to zero. We can, of course, reload them by typing:

```
RUN          Return
```

and verify the results by:

```
PRINT M(2,2)          Return
```

```
Row 2 Col. 2 = 6
```

We must ERASE an array before reDIMensioning it, or will get a "Duplicate Definition" error. It isn't often necessary to reDIMension.

Okay, Now What Do We Do With It?

Good question. Everything we learned in the last Chapter on Arrays applies.

We've only rearranged the deck chairs on this Titanic -- the end result is unaffected.

At this point, what we've learned is best utilized for calling up and loading relatively unchanging DATA. It is placed in a matrix so it can be accessed and compared, processed or otherwise put to work. Typical applications are:

1. Technical Tables: Instead of looking up the same information in tables, store the tables in DATA Lines and let the Computer look them up and do any needed calculations. The time saved may quickly pay for the Computer.

2. Price Quotes: I saw this approach used by a lumber yard to furnish fast quotes on building materials, and by a printing shop for fast quoting of all sorts of printed matter. The programs are written so simply that customers just belly up to the counter, answer the computer's questions, and get their quote right on the screen and printer.

   The latest prices on paper products and printing costs are held in DATA Lines and "spun up" into the Matrix at the beginning of the day. The customer responds to a "Menu" on the screen and answers some questions on quantity and quality. The quote is calculated and PRINTed.

When DATA is loaded in externally, either via the keyboard or disk, we obviously don't want to have to go through that loading process *each time* we want an answer. It's important therefore, to never let execution END. Always have it come back to a screen "Menu" of choices, or at least a simple INPUT statement. If an END is hit, the matrix crashes and the DATA has to be reRun to reload it.

String Matrices

So far we have concentrated on *numeric* arrays. They can also be used to hold letters or words, using the same rules learned in the Chapters on Strings, including CLEARing enough String space.

String matrices need String names. Make these subtle changes in the resident program.

```
10 DIM M$(10,4)
35    READ M$(R,C)
```

```
40    PRINT M$(R,C),
```

...and Run.

Absolutely no difference! We changed to a string matrix but the data is all numeric. Strings handle numbers as well as letters, but not vice-versa.

Let's change the DATA to words and try it again. Change:

```
10 DIM M$(5,4)
20 FOR R=1 TO 5
90 PRINT
100 DATA ALPHA,BRAVO,CHARLIE,DELTA,ECHO
110 DATA FOXTROT,GOLF,HOTEL,INDIA,JULIETTE
120 DATA KILO,LIMA,MIKE,NOVEMBER,OSCAR
130 DATA PAPA,QUEBEC,ROMEO,SIERRA,TANGO
```

Save As STRMAT and Run.

Stop for a moment and contemplate the string-comparing and string-handling techniques we learned a few Chapters ago. Your mind should be running flat out at this point, considering the possibilities.

How About Mixing Strings And Numerics?

Oh! Funny you should ask. That's why we ran all numbers in a string matrix, then all words with that same program. They mix very well, as long as the mixer is a string matrix and not a numeric one.

We have one final program. It is designed for demonstration only but could be expanded to INPUT the DATA from disk and be quite usable. It demonstrates some important possibilities and programming techniques.

The Objective

The objective of this demo program is to allow a church treasurer to keep track of who gave what, when. Could use the same program with a service club, bowling league, or any organization that has a membership and dues.

We want to be able to access every member's record by name and get a readout on his status.

Let's start with the DATA. Type this in the New program:

```
1000 REM  * DATA FILE *
1010 DATA 07.0186,JONES,15
1020 DATA 07.0186,SMITH,87
1030 DATA 07.0186,BROWN,24
1040 DATA 07.0186,JOHNSON,53
1050 DATA 07.0186,ANDERSON,42
```

The first number in each DATA Line employs "data compression," that is, "encoding" several pieces of information into one number. This number contains the Month, Day and Year in one 6 digit number. (Using string techniques, we could easily strip them apart again, if we wished, for special reports.) Single precision will hold the 6 digits accurately.

The second thing we've done with this first number is protect the leading 0. Since months below October are identified by only one digit, the leading 0 would be lost in these months and the number changed to only 5 digits. There are other ways to get around that problem, but we put in a decimal point just to act as an unmovable reference.

The second element in each DATA Line is the *name*. We could put in the full name, and if we used a comma, would of course have to enclose the name in quotes.

The third element in each DATA Line holds the amount of money tendered on that date.

Obviously, a full DATA set would contain many entries for each week, and many weeks in a row. We don't need to enter that much DATA to demonstrate the principles involved and want to keep it short and to the point.

This DATA must now be READ into a string matrix (displaying it as we go).

```
10 FOR E = 1 TO 5 : PRINT E,  'LOAD 5 ENTRIES
```

```
20  FOR D = 1 TO 3                    'LOAD DATE,
                                       NAME, AMT
30   READ R$(E,D)
40   PRINT R$(E,D),                    'TEMP ARRAY
                                       PRINTOUT
50  NEXT D
60  PRINT
70 NEXT E
90 PRINT : PRINT "ENTRY #","DATE","NAME",
   "AMT $"
```

Save As RECORDS1, close the List window and Run.

Very good. The Matrix is loaded, and its accuracy confirmed on the screen. We see the first 5 bookkeeping entries from July 1, 1986.

Now that we know it loads OK, we can remove some of the test software. Change Line 10:

```
10 FOR E = 1 TO 5                    'LOAD 5 ENTRIES
```

Delete Lines 40 and 60

...and Run.

Good. We still get the heading, but the matrix contents display is gone. Now, how can we interrogate the Matrix to pull an individual member's record?

Guess we first have to ask a question. Type:

```
80 INPUT "WHOSE RECORD ARE YOU SEEKING";N$
```

Then we have to write the program to scan the matrix and compare N$, the name we INPUT, with each element, R$(E,D), until we find a match. This means setting up the FOR-NEXT loops again and scanning every element.

Add:

```
100 FOR E = 1 TO 5
110  IF R$(E,2) = N$ THEN 150
120 NEXT E
130 PRINT,N$; " IS NOT IN THE FILE."
140 PRINT : GOTO 80
150 PRINT E, R$(E,1),R$(E,2),R$(E,3)
160 PRINT : GOTO 80
```

Save As RECORDS2 and Run.

Answer with names that are in the DATA Lines, and those that are not. Lines 140 and 160 have built-in defaults back to the question.

The key Line is #150. It PRINTs 4 things:

E	the entry Number on that date
R$(E,1)	the Date in the memory cell just *preceding* the one containing the member's name
R$(E,2)	the Name
R$(E,3)	the Amount

If you have trouble visualizing what Line 150 is doing, insert this temporary Line. It PRINTs the *address* of each DATA element just below it and is very helpful:

```
155 PRINT E, E;1, E;2, E;3
```

...and Run.

Again, the preceding program was not written to be a model of programming style and efficiency -- but to teach the basics of loading and retrieving "record-keeping" type information from a Matrix.

EXERCISE 41-1: Write a program that fills a two dimension string array with:

JONES, C.	10439	100.00
ROTH,J.	10023	87.24
BAKER,H.	12936	398.34
HARMON,D.	10422	23.17

EXERCISE 41-2: Sort the names of the array in Exercise 41-1 alphabetically. Don't forget to keep the rest of the information on each row with the original name. This Exercise will be a challenge. Think it through carefully.

EXERCISE 41-3: If you survived Exercise 41-2, try sorting the array in increasing order by the numbers in Column 3.

Learned In Chapter 41

Statements

ERASE

Miscellaneous

Multi-Dimension Arrays
String Arrays

PART 7

MISCELLANEOUS

Chapter 42

PEEK And POKE

PEEK and POKE are BASIC words that allow us to do "non-BASIC" things. They provide the means whereby we can PEEK into the innards of the Computer's memory and, if we wish, POKE in new information.

It is not our purpose here to become experts in machine language programming nor on how the Computer works. We have to approach this and related topics a little gingerly, lest we fall over the edge into a computer abyss (or is it an abysmal computer?)

We do know, however, that computers do their thing entirely by the manipulation of numbers. Therefore, when we PEEK at the contents of memory, guess what we'll find? Numbers? Very good! (Ummmyaas).

Large chunks of the Computer's memory are reserved, or "mapped," for very specific uses. The entire screen display, for example, uses byte addresses 108,288 through 130,175. All numbers we talk about here are decimals, not hex, octal, binary or Sanskrit.

Turn the Computer off to clear out memory, wait a minute, turn it back on, bring up Microsoft BASIC, and type in this New program:

```
10 N = 0
20 PRINT N, PEEK(N), CHR$(PEEK(N))
30 N = N + 1
40 GOTO 20
```

Let's analyze the program before RUNning it.

Line 10 sets the *beginning* address where we want to start

PEEKing. There are lots of good places to go spelunking, and we can change Line 10 to start wherever we want.

Line 20 PRINTs three things:

1. The address -- that is, the number of the byte at whose contents we are PEEKing.

2. The contents of that byte expressed as a decimal number between 0 and 255.

3. The contents of that address converted to its ASCII character. (Many of the ASCII characters are not PRINTable. Go back to the Chapter on ASCII if *your* memory has grown dim.)

Okay, now Run the program, being ready to freeze it with ⌘S if you see something interesting. It can also be stopped at any time with ⌘. or Stop from the Run menu. To restart without resetting N back to 0, select Continue from the Run menu.

See anything interesting? You have to be able to read vertically as the letters swish by.

Change N to start at any of the 16,777,215 different places in memory and PEEK to your heart's delight. You can't goof up anything by just PEEKing. It's indiscriminate POKEing that gets one into trouble.

If you haven't already done so, stop the program.

The Command window is very handy for resetting the starting address. Change the value of N by just typing:

```
N = 2300          Return
```

for example, then type:

```
CONT      Return    (or select Continue from the Run menu)
```

When done PEEKing, and having seen far more information than can possibly

be absorbed, rework Line 20 to read simply:

```
20 PRINT CHR$(PEEK(N));
```

Insert:

```
5 WIDTH 60
```

...and Run.

It PRINTs only the ASCII characters, horizontally, and is the ideal program to RUN when friends visit. Just act casual about the whole display and avoid direct questions. Makes a great background piece for a science fiction movie.

When you find an interesting spot, hit ⌘., then:

```
PRINT N          Return
```

at the Command window to find out where in memory you are PEEKing. (Don't you wish we could explore the corners of our minds as easily?)

CONTinue on when ready.

Having degenerated from PEEKing to leering, we'd better move on.

Careless POKEing Can Leave Holes...

Before POKEing, we'd better see that we're not POKEing a stick into a hornets' nest. It's with the greatest of ease that we destroy a program in memory by POKEing around where we shouldn't.

Let's PEEK around 68000 and see if anything is going on there.

Change these two program lines to:

```
10 N = 68000
20 PRINT N; PEEK(N),
```

If you have a Mac with 512K of memory, change Line 10 to:

```
10 N = 20000
```

...and Run.

```
68000  0    68001  0    68002  0    68003  255
68004  255  68005  0    68006  0    68007  1
68008  255  68009  255  68010  0    68011  0
```

...etc.

and the results on the 512K Mac are:

```
20000  255  20001  255  20002  255  20003  255
20004  255  20005  255  20006  255  20007  255
20008  255  20009  255  20010  255  20011  255
```

...etc.

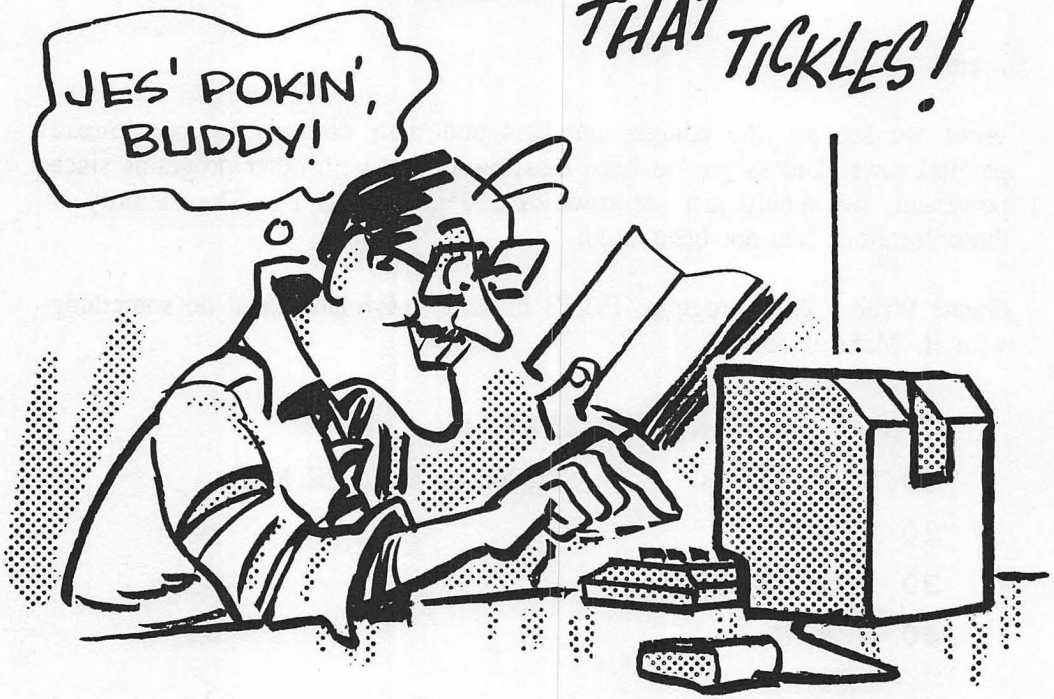
What we see are the address numbers and their contents in easy-to-read parallel rows. Unless you've been messing around with other programs since power-up, we should just see rows of 255's, 0's and 1's. The memory at these locations has not been used.

Great! Write a New program, POKE in some information and do something with it. Make it read:

```
5 REM  * POKE PROGRAM *
10 N = 68000          (20000 on the 512K Mac)
20 READ D
30 POKE N,D
40 N = N + 1
```

HEE HEE HEE HO HO HA

HEE HEE HEE THAT TICKLES!



```
50 IF N = 68011 THEN END          (20011 on the 512K Mac)
60 GOTO 20
100 DATA 80,69,69,75,45,65,45,66,79,79,33
```

Before Running, let's analyze it.

Line 10 initializes the starting address at 68000 (or 20000 on the 512K Mac).

Line 20 READs a number from the DATA Line.

Line 30 POKEs the DATA "D" into address "N."

Line 40 increments the address number by one.

Line 50 ENDs execution when we have POKEd in all 11 pieces of DATA.

Line 60 sends us back for more DATA.

Line 100 stores the DATA we are going to POKE into memory.

...now Run.

Well, that was sure fast. I wonder what it did? How can we find out? Should we PEEK at it? Yes, but let's leave the old program in and just start a new one at 200.

```
200 REM  * PEEK PROGRAM *
210 FOR N=68000 TO 68010
      (20000 TO 20010 on the 512K Mac)
220  PRINT N, PEEK(N)
230 NEXT N
```

...and in the Command window type RUN 200 Return.

		on the 512K Mac:	
68000	80	20000	80
68001	69	20001	69

68002	69	20002	69
68003	75	20003	75
68004	45	20004	45
68005	65	20005	65
68006	45	20006	45
68007	66	20007	66
68008	79	20008	79
68009	79	20009	79
68010	33	20010	33

How about that? We really *did* change the contents of those memory locations. We shot the numbers from our DATA line right into memory. Now if we only knew what those numbers stood for. Wonder ... if we changed them to ASCII characters, would they tell us anything?

Insert:

```
205 CLS
```

and change:

```
220 PRINT CHR$(PEEK(N));
```

...and RUN 200.

That's how PEEK and POKE work.

Learned In Chapter 42

Statements	Functions
POKE	PEEK

Chapter 43

Logical Operators

In classical mathematics (fancy words for simple ideas), there exist what are known as the "logical AND," the "logical OR," and the "logical NOT."

So The One Cow Said To The Other Cow...

In Figure 43-1, if gate A AND gate B AND gate C are open, the cow can move from pasture #1 to pasture #2. If any gate is closed, the cow's path is blocked.

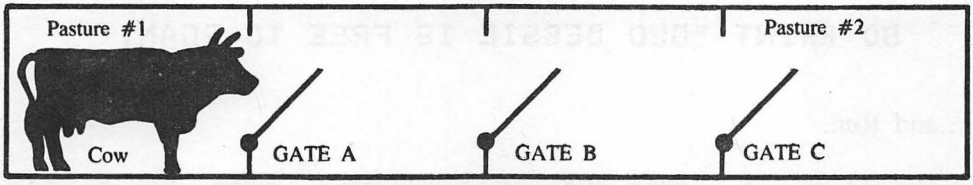


Figure 43-1

The principle is called "logical AND."

In Figure 43-2, if gate X OR gate Y OR gate Z are open, then old Bess can move from pasture #3 to #4. That principle is called "logical OR." These ideas are both pretty logical. If the cow can figure them out surely we can!

Using these ideas is very simple. Type this New program:

```
10 INPUT "IS GATE 'A' OPEN";A$
20 INPUT "IS GATE 'B' OPEN";B$
30 INPUT "IS GATE 'C' OPEN";C$
40 PRINT
50 IF A$="Y" AND B$="Y" AND C$="Y" THEN 80
```

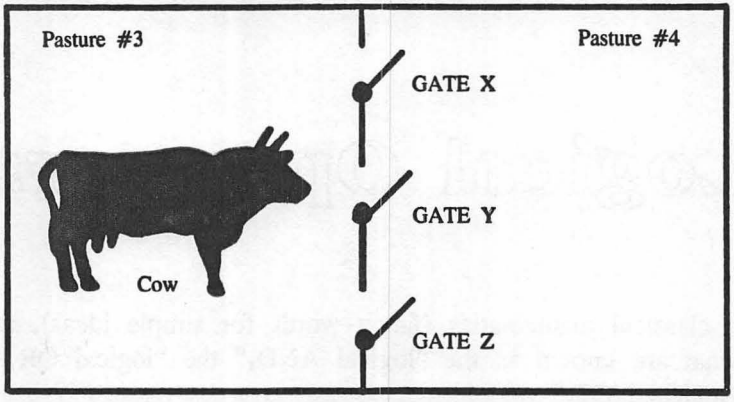


Figure 43-2

```
60 PRINT "OLD BESSIE IS SECURE."
70 END
80 PRINT "ALL GATES ARE OPEN."
90 PRINT "OLD BESSIE IS FREE TO ROAM."
```

...and Run.

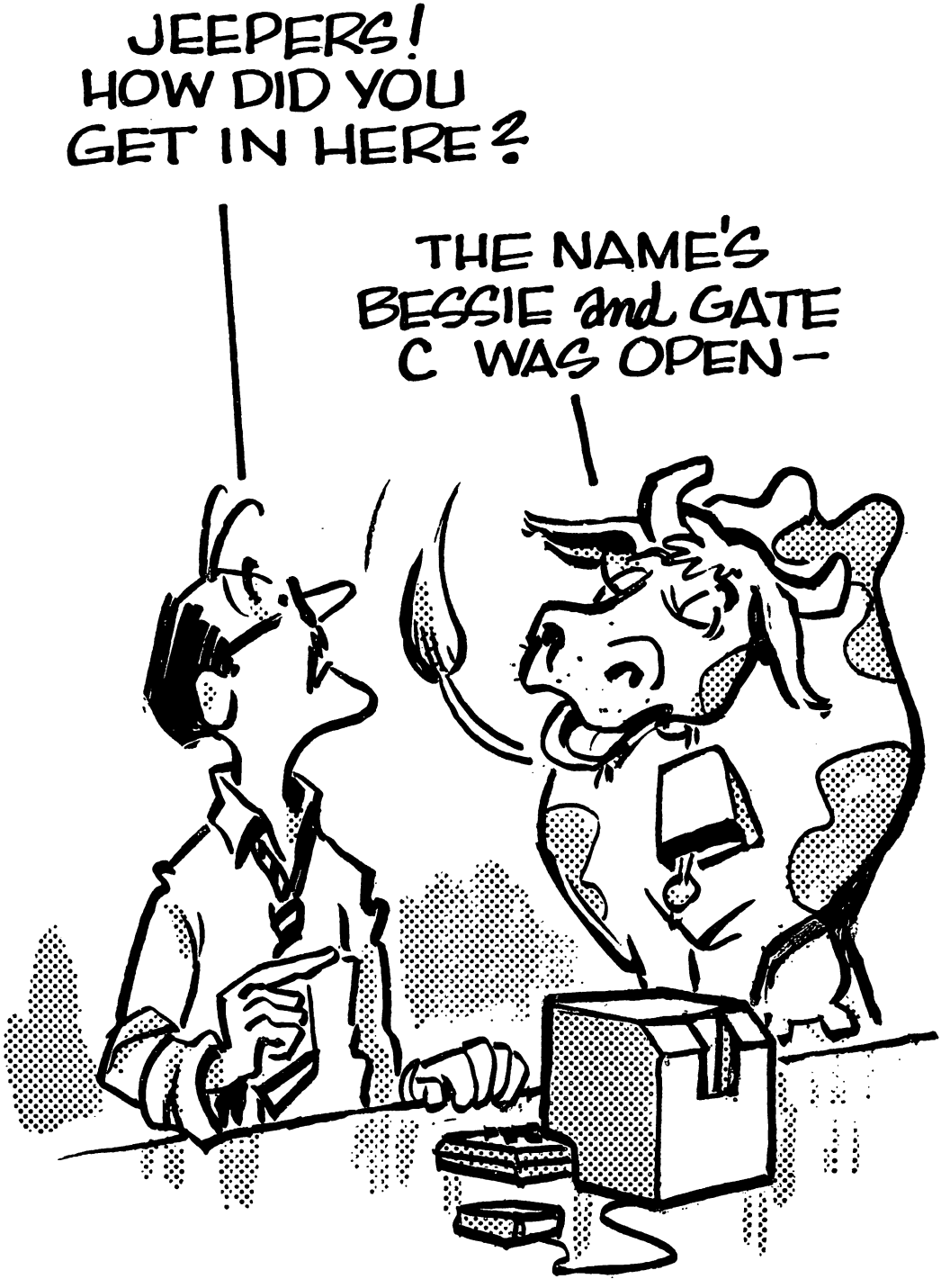
Answer the questions (Y/N) differently during different RUNs to see how the logical AND works in Line 50.

Where Is The Logic In All This?

You should by now understand every part in the program, except perhaps Line 50.

> Lines 10, 20, and 30 INPUT the gate positions as *open* (which we defined as equal to "Y") or *closed* (defined as "N"). We could have defined them the other way around and rewritten Line 50 to match, if we'd wanted to.
>
> Line 50 is the key. It reads, literally, "If gate A is *open*, AND gate B is *open*, AND gate C is *open*, then go to Line 80. If any one gate is closed, report that fact by defaulting to Line 60."

Imagine how this simple logic could be used to create a super-simple "com-


JEEPERS! HOW DID YOU GET IN HERE?
THE NAME'S BESSIE and GATE C WAS OPEN—

puter" consisting of only an electric switch on each gate. Add a battery and put a light bulb in the farmer's house. The bulb could indicate if any of the gates are open. Such a "gate-checking" computer would have only three memory cells -- the switches.

Hmm. It would do the job a lot cheaper than a Macintosh ... but would be awfully hard to play *Invaders* with.

EXERCISE 43-1: Using the above program as a model, and the "OR logic" seen in Figure 43-2, write a program which will report Bessie's status as determined by the position of Gates X, Y and Z.

Teacher's Pet

Here is a simple program which uses > instead of the equals sign in a logical test. The student passes if he has a final grade over 60 OR a midterm grade over 70 AND a homework grade over 75. Enter this New program, Run it a few times, and see how efficiently the logical OR and logical AND tests work in the same program Line (40).

```
10 INPUT "FINAL GRADE";F
20 INPUT "MIDTERM GRADE";M
30 INPUT "HOMEWORK GRADE";H
40 IF (F>60 OR M>70) AND H>75 THEN 70
50 PRINT "FAILED"
60 END
70 PRINT "PASSED"
```

Does this give some idea of the power and convenience of logical math? The actual "cut off" numbers could, of course, be set at any level.

Logical Variations

This next program example mixes equals, greater-than and less-than signs in the same program. It determines and reports whether the two numbers we INPUT are both positive, both negative, or have different signs.

Analyze the program. Note the parentheses. Although they are not necessary, they tell us to shift our thinking to "logical." Type it in and Run.

```
10 INPUT "FIRST NUMBER IS";F
20 INPUT "SECOND NUMBER IS";S
30 IF (F>=0) AND (S>=0) THEN 70
40 IF (F<0) AND (S<0) THEN 90
50 PRINT "OPPOSITE SIGNS"
60 END
70 PRINT "BOTH POSITIVE OR ZERO"
80 END
90 PRINT "BOTH NEGATIVE"
```

With Graphics Too, Yet

Yes, the logical symbols also work with the graphics statements. See if you can figure out the surprise which will be caused by the logical AND in Line 40. Type this New program in and Run.

```
10 FOR X = 50 TO 100
20  FOR Y = 100 TO 150
30   IF (X>=75) AND (Y>=125) THEN 50
40   PSET(X,Y)
50  NEXT Y
60 NEXT X
99 GOTO 99
```

Use [Ctrl] [Break] to exit the program's endless loop.

What happens if we replace the AND in Line 40 with an OR? After you think you have figured out, do it and see the result.

Did you guess right?

There's More?

Oh, yes -- the only limit is your imagination. See how easily the logical notation makes the drawing of lines? Change Line 30 to read:

```
30   IF (X=75) OR (Y=125) THEN 50
```

What happens to the program if we replace OR with AND? Sketch your estimated result, then change Line 30 and try it.

Hope you got it right. If not, it really sneaked up, didn't it?

Using the INT function, we can create a white-on-black grid. The reasoning is:

In the *horizontal* dimension:

The INT(X/10)*10-X will equal 0 when X equals 50, 60, 70, 80, 90 and 100.

In the *vertical* dimension:

The INT(Y/10)*10-Y will equal 0 when Y equals 100, 110, 120, 130, 140 and 150.

Oh come on, it's very simple if you take the time and think it through!

Replace the old Line 30 with:

```
30   IF INT(X/10)*10-X=0 OR INT(Y/10)*10-Y=0
     THEN 50
```

and you will create a five-by-five grid.

And on and on it goes...

NOT

In addition to the logical AND and OR functions, we have what is called logical NOT. Here is how it can be used:

```
10 INPUT "ENTER A NUMBER";N
20 L = NOT(N>5)
30 IF L = 0 GOTO 60
40 PRINT "N WAS NOT GREATER THAN 5"
50 END
60 PRINT "N WAS GREATER THAN 5"
```

...and Run.

Line 20, containing NOT, is obviously the key one. If the statement in Line 20 is *true* (namely, that N is NOT larger than 5), the Computer makes the value of L = -1. The test in Line 30 then fails.

If, on the other hand, N IS larger than 5, the statement is *false* and the Computer makes the value of L = 0.

True = -1 and False = 0. (Time for the primal scream, again. All together, now...)

More Logical Operators

As if these 3 *logical* operators weren't enough, Microsoft BASIC allows use of 3 more "Logical" words. They are (in order of appearance):

EQV, XOR, and IMP.

To help see how these things work, let's write a "testbed" program into which we can install them.

```
10 INPUT "ENTER A VALUE FOR X";X
20 INPUT "ENTER A VALUE FOR Y";Y
```

```
30 IF (X<10) AND (Y>10) THEN 60
40 PRINT : PRINT "CONDITION WAS FALSE"
50 END
60 PRINT : PRINT "CONDITION WAS TRUE"
```

...and Run.

INPUT the number 5 for X and 15 for Y. No big deal. Both comparisons were true, which made the AND condition true.

OR

Replace the AND in Line 30 with OR and Run. Try different numbers to get a feel for the program.

EQV

There are several more "advanced" logical operators. EQV stands for EQuiValence. Replace the OR in Line 30 with the word EQV.

```
30 IF (X<10) EQV (Y>10) THEN 60
```

The condition in Line 30 will be true only if both arithmetical comparisons are the same. Only if X is less than 10 AND Y is greater than 10, OR if X is *not* less than 10 AND Y is *not* greater than 10.

Try the number 5 for X and 15 for Y. Both tests pass so the overall condition is true.

Try 15 for X and 5 for Y. Both conditions are false, but since they are *both* the same (false in this case), the overall condition is true and execution jumps to Line 60.

XOR

XOR stands for eXclusive OR. This means that if one *and only one* test passed, the overall condition will be true.

Replace the EQV in Line 30 with the word XOR. Run with different numbers. Try 5 for X and 15 for Y. Execution falls through to Line 40 because *both* tests pass. Remember if we were using the regular OR, the overall condition would be true.

IMP

Our final operator is IMP which stands for IMPlication. This is probably the hardest to understand. The IMP condition will be *true* for all conditions except when the first test is *true* and the second test is *false*. The *overall condition* is then *false*. Replace the XOR with an IMP:

```
30 IF (X<10) IMP (Y>10) THEN 60
```

...and Run.

Try 5 for both X and Y. These numbers give us a *false* condition. All other conditions are *true*.

Order Of Operations

When trying to figure out which gets calculated first in the thick of a "humongous" equation, remember this pecking order:

Those operations buried deepest inside the parentheses get resolved first. The idea is to clear the parentheses as quickly as possible. When it all becomes a big tie, here's the order:

1. Exponentation -- a number raised to a power

2. Negation, that is, a number having its sign changed -- typically, a number multiplied times -1

3. Multiplication and division -- from left to right

4. Integer Division

5. MODulo Division

6. Addition and subtraction -- from left to right

7. Relational operators — less than, greater than, equals, less than or equal to, greater than or equal to, not equal to -- from left to right

8. The logical NOT

9. The logical AND

10. The logical OR and XOR

11. The logical EQV

12. The logical IMP

And In Conclusion

Logical math is worth the hassle. As one last fun program, enter and Run this "Midnight Inspection." Line 90 checks each response for a NO answer (instead of a YES). Using logical OR, it branches to the "no-go" statement (Line 110) if any one of the tests is negative ("N").

```
10 PRINT "ANSWER WITH 'Y' OR 'N'."
20 PRINT
30 INPUT "HAS THE CAT BEEN PUT OUT";A$
40 INPUT "PORCH LIGHT TURNED OFF";B$
50 INPUT "ALL DOORS/WINDOWS LOCKED";C$
60 INPUT "IS THE T.V. TURNED OFF";D$
70 INPUT "THERMOSTAT TURNED DOWN";E$
80 PRINT:PRINT
90 IF A$="N" OR B$="N" OR C$="N" OR D$=
    "N" OR E$="N" THEN 110
100 PRINT "   (10 spaces)  GOODNIGHT":END
110 PRINT "SOMETHING HAS NOT BEEN DONE. DO
    NOT GO TO BED"
120 PRINT "UNTIL YOU FIND THE PROBLEM!"
130 GOTO 20
```

In most cases, AND and OR statements are interchangeable if other parts of a program are rewritten to accommodate the switch.

Learned In Chapter 43

Miscellaneous

Logical AND
Logical OR
Logical NOT
Logical EQV
Logical XOR
Logical IMP
Order of Operations

Chapter 44

A Study Of Obscurities

Microsoft BASIC has some features that are not used by most beginning programmers. Their use presumes special applications and requires knowledge which is really beyond the scope of this book. In the interest of completeness, however, abbreviated descriptions of what they are and how they are used are included in this Chapter.

CALL

The CALL Function has a variety of uses, most of them having little to do with BASIC. It allows us to "call" or "gosub" a program written in ASSEMBLY language and "return" back to the main BASIC program when it's finished. To make much sense of CALL, you'll need ASSEMBLY language skills -- a whole book in itself.

A typical CALL statement might look like this:

```
CALL ADDRS
```

ADDRS is a numeric variable which contains the address of the machine language program.

CALL also allows us to pass certain values to our machine language routine. We can use some of the Macintosh's machine language programs to do special things for our BASIC programs.

Macintosh ROM Routines

ROM stands for Read Only Memory. The ROM on the Macintosh has many built-in programs that allow the Computer to communicate with us. These programs are machine language routines that take care of all the things that Macintosh does. Screen management, pull down menus, mouse control, and graphics are a few examples.

With the CALL command, we can Run these programs from BASIC. Here's one to try that will really come in handy:

```
CALL MOVETO(X,Y)
```

This moves an imaginary pen to a screen pixel coordinate specified by X and Y. If we printed text or graphics after a CALL MOVETO(X,Y), the output would start at the new coordinates.

Type in this New program:

```
INPUT "ENTER X =";X
INPUT "ENTER Y =";Y
INPUT "ENTER A LETTER: ";S$
CLS
PRINT S$
CALL MOVETO(X,Y)
PRINT S$
```

...and Run.

From Microsoft BASIC, we can make two CALLS to some built-in ROM routines to change the "Geneva" proportionally spaced text font to "Monaco," a mono-spaced font. Insert this Line at the beginning of the program:

```
CALL TEXTFONT(4) : CALL TEXTSIZE(9)
```

...and Run.

The Output window is now displaying the TEXT in the Monaco FONT and the smaller TEXTSIZE allows a full 80 column format.

Use X coordinates between 0 and 480 and Y coordinates from 10 to 220. Numbers outside of this range may cause printing to take place outside the window.

Refer to the Microsoft BASIC manual for a list of available ROM routines.

Machine and Assembly language programming books are readily available for that small percentage of readers who want to pursue the subject. You, at least, have a sufficient introduction to nod your head and smile knowingly when others try to impress you with their knowledge of these things. Consult the Microsoft BASIC reference manual for more details.

VARPTR

While VARPTR (short for VARiable PoinTeR) is found in Microsoft BASIC, it's about as far from main-line BASIC as anything we have.

Take A Deep Breath

If a variable is *numeric,* VARPTR tells us the *location* of the *first byte* of the number stored in that variable.

If it's a *string* variable, VARPTR tells us where in memory the INDEX to the variable is located. Read that last line carefully. We don't want anyone getting lost.

VARPTR doesn't have the common decency to point to the location of the *contents* of a *string* variable. Instead, it points to a five byte "index" to the variable. The five bytes contain:

1. The *length* of the string
2. The address of the contents of the string

To actually find the *contents* of the string variable, we have to calculate the location using the last three bytes of the "index" to that variable. Sound complicated? Well, it's a bit tricky, but this example should clarify matters a bit.

Enter this New program:

```
10 REM * STRING VARIABLE LOCATOR *
20 A$ = "12345"
30 X = VARPTR(A$)
```

```
40 PRINT "THE INDEX TO A$ IS AT";X
```

...and Run.

Line 30 uses VARPTR to store the address of the index to A$ in X. Line 40 PRINTs it.

We haven't found the *contents* of A$ yet, just the *index*. Hang in there. Add:

```
50 L = PEEK(X+2)*65536+PEEK(X+3)*256+PEEK(X+4)
60 PRINT "A$ IS HIDING AT LOCATION";L
```

...and Run.

So that's where the little rascal is. Line 50 uses some fancy footwork to convert bytes three (X+2), four (X+3), and five (X+4) of the index (X) into the actual location L of A$. Line 60 PRINTs the *address* value.

Next we need to find the LENgth of A$ (of course, we could use LEN(A$), but that's not our purpose here). Add:

```
70 S = PEEK(X) * 256 + PEEK(X+1)
80 PRINT "THE LENGTH OF A$ IS";S
```

...and Run.

How do we know that all this information is correct? Sure. PEEK at the contents of A$, and compare it with 12345. Add:

```
90 FOR N = L TO L+S-1
100    PRINT CHR$(PEEK(N));
110 NEXT N : PRINT
```

...and Run.

Satisfied? The 5 digits in A$ are stored in 5 consecutive memory locations.

Now, knowing where a variable is located in memory may not seem too useful at first blush, but it has some surprising consequences. Once we have found the location of the string variable, we can modify its contents. Try this change:

```
110   READ Y : POKE N,Y
120 NEXT N : PRINT
130 PRINT A$
140 DATA 169,217,165,217,169
```

Surprise! We poked graphics codes into an unsuspecting "normal" string variable and transformed it into a pictorial masterpiece.

In the Command window, type:

```
PRINT A$          Return
```

to be sure we aren't just dreaming. Yes, we actually modified the contents of A$ by using VARPTR to *find* the string, and then POKEing in new numbers. These computers can be downright fun once we get to know them.

Press ⌘ L, and look at Line 20. Did we do that? I'm afraid so. A Listing contains the actual graphics.

Leave with this thought. We packed a "dummy" string with only 5 graphic codes. A string variable constant *can* hold up to 255 characters. Just imagine what we could do with strings packed with up to 255 cursor control codes, graphic codes, and spcial character codes! If that doesn't push your imagination into overload, you might as well trade this Computer in for a $4.95 calculator.

SWAP

The SWAP function lets us exchange the position of two variables with ease.

Type:

```
PRINT "ENTER TWO NUMBERS ";
INPUT "SEPARATED BY COMMA";A,B
```

```
PRINT : PRINT "A =";A,"B =";B
SWAP A,B
PRINT "SWAPPED"
PRINT "A =";A,"B =";B
```

...and Run.

It works with string variables, too. Try changing every A and B to A$ and B$. INPUT a pair of names, and watch what happens.

Rodent Control (Or Watching Your Mouse)

Here's what we've all been waiting for. We can use the mouse in programs by using the MOUSE(X) function. The X in parenthesis is a number between 0 and 6 which will report seven different mouse values. They are:

MOUSE(0) = Button Status
MOUSE(1) = Current X (horizontal) coordinate
MOUSE(2) = Current Y (vertical) coordinate
MOUSE(3) = Starting X coordinate
MOUSE(4) = Starting Y coordinate
MOUSE(5) = Ending X coordinate
MOUSE(6) = Ending Y coordinate

Enter this New program:

```
10 REM * MOUSE TRAP PROGRAM *
20 PRINT CHR$(255)
30 X = MOUSE(1) : Y = MOUSE(2)
40 CALL MOVETO(30,13)
50 PRINT "X =";X;" Y =";Y
60 IF MOUSE(0) = 0 THEN 30
70 IF X<3 OR X>11 OR Y<3 OR Y>11 THEN 30
80 PRINT "SNAP!!!" : BEEP
```

...and Run.

Move the mouse, and notice how the X and Y coordinates change whenever the mouse location is updated.

Move the pointer above the display window. The Y value becomes negative. If the pointer is moved past the left edge of the display window, X will become negative.

Okay, ready to catch a mouse? Move the pointer so that it is inside the square in the upper left corner of the display window. Now press the mouse button. SNAP! The little critter is now harnessed by these simple commands.

Here's how the program works:

Line 20 PRINTs the box character.

Line 30 sets X to the current horizontal mouse location and sets Y to the current vertical position.

Line 40 calls a built in Macintosh ROM routine (MOVETO) to enable us to PRINT X and Y at certain screen locations.

Line 50 PRINTs X and Y.

Line 60 performs a check on the status of the mouse button (zero means it is not being pressed -- more about the button later).

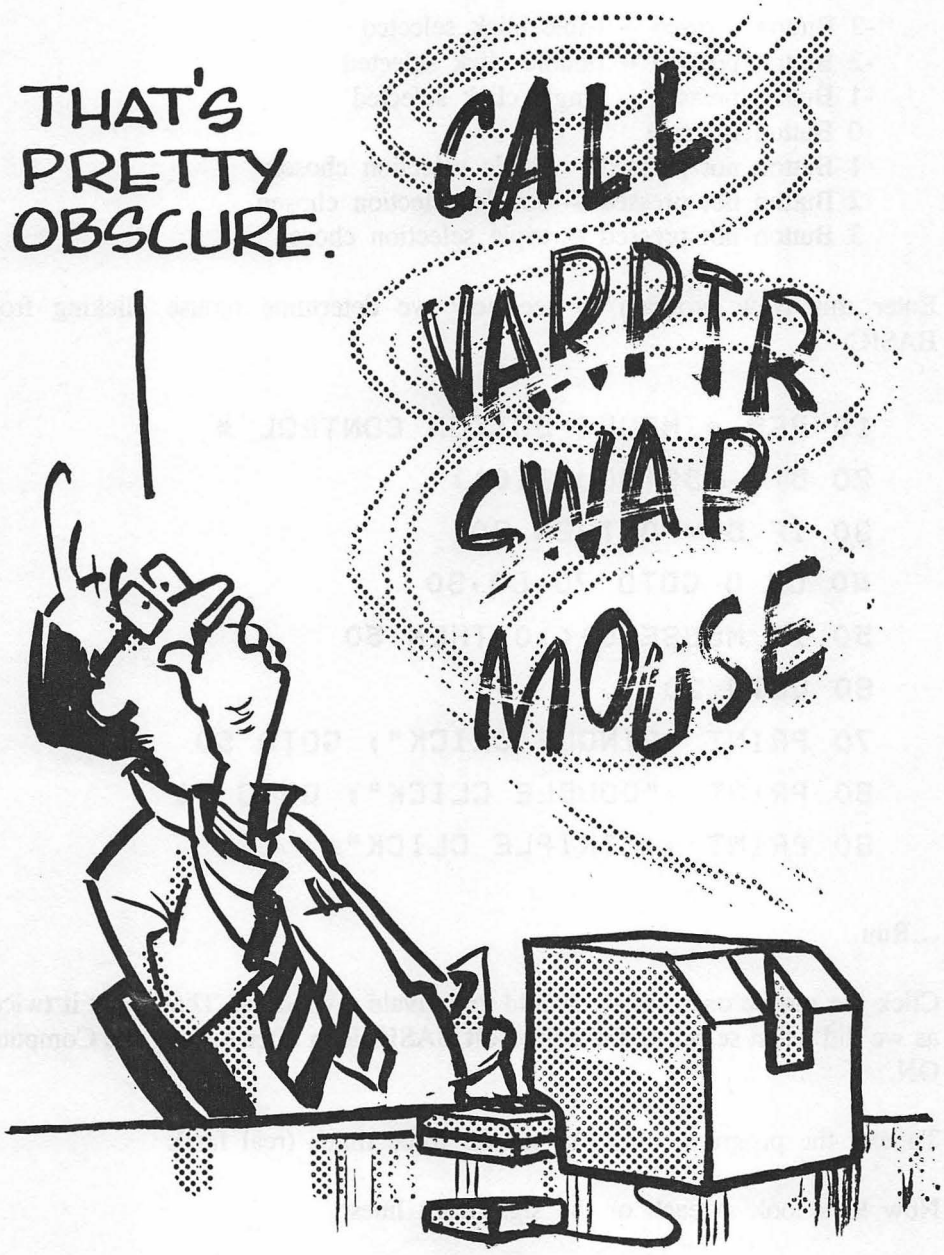
Line 70 tests the location of the pointer. At this point, we know that the button is pressed. But if the pointer is put inside of the box, we go back to Line 30.

If program execution reaches Line 80, then we know all the conditions were met, and we've caught our mouse! Oh, BEEP? Well, what else do you think a command called BEEP can do besides BEEP?

More Than A Mouseful

The MOUSE(0) Function will return a number between -3 and 3. A zero means the mouse button is not currently down (or hasn't been pressed since the last MOUSE(0) statement was executed).

Positive values mean the mouse button is NOT being pressed at the time, but it has been pressed since the *last* reference to MOUSE(0).


THAT'S PRETTY OBSCURE.
CALL VARPTR SWAP MOUSE

Negative values mean that the mouse button is down. These values will also tell us what type of selection is being made.

-3 Button pressed -- triple click selected
-2 Button pressed -- double click selected
-1 Button pressed -- single click selected
 0 Button inactive
 1 Button not pressed -- single selection chosen
 2 Button not pressed -- double selection chosen
 3 Button not pressed -- triple selection chosen

Enter this New program to see how we determine mouse clicking from BASIC:

```
10 REM * MOUSE BUTTON CONTROL *
20 B = ABS(MOUSE(0))
30 IF B = 0 THEN 20
40 ON B GOTO 70,80,90
50 IF MOUSE(0)<>0 THEN 50
60 GOTO 20
70 PRINT "SINGLE CLICK": GOTO 50
80 PRINT ,"DOUBLE CLICK": GOTO 50
90 PRINT ,,"TRIPLE CLICK"
```

...Run.

Click the mouse once, as we would to activate a window. Then click it twice, as we did when selecting the Microsoft BASIC icon after turning the Computer ON.

To exit the program, click the mouse three times (real fast).

Now let's look at each of the significant lines:

Line 20 assigns the variable B to the ABSolute value of the mouse button status. The reason for this technique will become clear in Line 40.

Line 30 tests the button to see if it turned up zero, meaning the button is NOT pressed down. As long as the value is zero, we'll keep going back to Line 20 until it changes.

The button is pressed when we get to Line 40. Remember that a negative mouse button status value means that the button is pressed. ON-GOTO will not work with negative values. That's why we used the ABSolute value of MOUSE(0) in Line 20.

Depending upon the type of click selected, we'll get different responses. If a triple click is selected, the program will stop.

Otherwise execution is passed to Line 50 where we test to see if our finger has "let-up" on the button. IF MOUSE(0) is not zero (we're still holding it down), THEN loop until we let go.

Line 60 sends us back to Line 20 to start over again.

Learned In Chapter 44

Functions	Miscellaneous
CALL	ROM routines
VARPTR	Mouse control
SWAP	
MOUSE(X)	
BEEP	

Chapter 45

Advanced Graphics

PTAB

PTAB is almost identical to TAB in its usage, except PTAB will TAB out to a specific pixel position. Type in this New program:

```
10 LINE (10,10)-(150,100),33,B
20 PRINT : PRINT
30 PRINT PTAB(58);"BURMA" : PRINT
40 PRINT PTAB(60);"SHAVE"
```

CIRCLE

The CIRCLE command lets us draw circles of various sizes on the Macintosh's screen.

Enter this New program and Run:

```
10 INPUT "HORIZONTAL CENTER ";X
20 INPUT "VERTICAL CENTER ";Y
30 INPUT "INPUT RADIUS ";R
40 CLS
50 CIRCLE (X,Y),R
```

X and Y are the coordinates for the center of CIRCLE. R is the number of pixels that make up the radius.

A full CIRCLE syntax (with all the bells and whistles) looks like this:

CIRCLE STEP (x,y), radius, color, start, end, aspect

STEP -- offsets a CIRCLE using a previous graphics coordinate. If, for example, the last coordinates used were (5,10), STEP(10,20) would set X at 15 (10 pixels from 5) and Y at 30 (20 pixels from 10).

radius -- the number of pixels in the radius of the CIRCLE

color -- The default is black. If 0 or 30 is selected as the variable, then CIRCLE is drawn in white.

start -- the circle's starting angle (in radians). When a negative number is used, a line is drawn from the center of the circle (or ellipse) to its starting point.

end -- the circle's end angle (in radians). Acts the same as the starting point in that a negative number causes a line to be drawn from the center of the circle (or ellipse) to its end point.

aspect -- a ratio of X to Y. The default is 1.0. If aspect is not 1, you'll get an oval.

Change Line 50 in our program to see the effect we can get by using some of the CIRCLE options:

```
50 CIRCLE (X,Y),R,,,-.5,-5.8
```

...and Run.

Make your own changes to the Line and try other options before moving on.

POINT

POINT returns the color value of a pixel on the screen. This program will explain it all:

```
10 RANDOMIZE TIMER
20 FOR X = 1 TO 9
30   C = INT(2 * RND(1) + 1)
40   IF C = 1 THEN PSET(X,10)
```

```
50  IF C = 2 THEN PRESET(X,10)
60 NEXT : PRINT
70 FOR X = 1 TO 9
80  P = POINT (X,10)
90  PRINT X;":    POINT =";P
100 NEXT : PRINT
110 PRINT "30 = WHITE, 33 = BLACK"
```

...and Run.

GET And PUT

GET and PUT are two commands that we use to "GET" part of the screen display and "PUT" it somewhere else. In order to do this, we have to DIMension some variable space to hold all this data. Unfortunately, the screen display takes up an extremely large chunk of memory. So we are limited to GETting only small parts of the screen.

The first thing to do in order to demonstrate GET and PUT is to draw (or PRINT) something to the screen. Let's begin by entering these Lines:

```
10 REM * GET AND PUT DEMO *
20 RANDOMIZE TIMER
30 DIM A%(2504)
40 LINE (0,0)-(50,50),,B
50 FOR N = 1 TO 5
60  X = INT(47 * RND(1)+2)
70  Y = INT(47 * RND(1)+2)
80  R = INT(20 * RND(1))
90  CIRCLE (X,Y),R
100 NEXT N
```

...Run.

A box is drawn, and random circles are place in and around it. The box and everything inside of it is what we will be GETting. But first, let's analyze what we have done so far:

Line 20 sets the RaNDom seed.

Line 30 DIMensions an integer variable named A% to 2504 elements. Why did we use this number? Simple. The portion of the screen that we want to GET takes up an area of 50x50 pixels. This means that there are 2500 pixels involved. The other 4 bytes are for the space that A% uses up in memory. Integer variables take up, as one would guess, only 4 bytes. Thus we need a DIMension of 2504.

Line 40 uses our LINE command to draw a box.

Lines 50-100 draw 5 random circles within and around the box.

Now add these last few Lines:

```
110 GET (0,0)-(50,50),A% : CLS
120 FOR I = 2 TO 426 STEP 53
130  FOR J = 2 TO 197 STEP 53
140   PUT (I,J),A%
150 NEXT J : NEXT I
```

...and Run.

How's that for fast graphics?

Line 110 uses the same coordinates as the LINE command in Line 40. The data that we GET is assigned to our dimensioned variable which is listed after the coordinates.

Line 120 initializes the FOR-NEXT loop that is used for PUTting the data in A% along the X axis (horizontal).

Line 130 uses a second FOR-NEXT loop for the Y axis. The loop steps by 53 pixels. In this way, each PUT does not overlap on a

previously PUT set of data. (Wow! There is some possibility for animation here!)

Just to slow things down a bit, let's insert one more Line to the program. Type:

```
105 IF MOUSE(0)=0 THEN 105
```

Line 105 will halt the program until we press the mouse button, allowing us to view the first half of the program before PUTting the boxes all over the screen.

Now let's Run the program, pressing the mouse button when we're ready to go on.

PUT With All The Options

Even more powerful than GET, the PUT statement can also be used with "action verbs" to perform special tasks. Remove Lines 105 and 120 thru 150 from the current program in memory, then add:

```
120 FOR J = 50 TO 200
130   PUT (0,0)-(J,J),A%
140 NEXT
```

...and Run.

Hmmm. Never did care too much for modern art.

In the first program using GET and PUT, the ending X and Y coordinates of PUT were kept equal with the values specified in the GET. Since we changed the values when we PUT the screen data to the display, we change the image's size.

Just for fun, type in this sequence in the Command window:

```
CLS : PUT (20,20)-(300,100),A%          Return
```

It will even change shape!

Now what's all this jazz about "action verbs"?

To clean up the current program, Edit Line 130 to read:

```
130 PUT (0,0)-(J,J),A%,PSET
```

...and Run.

Ah, now that's more like it! But what's PSET doing at the end of the PUT command?

PSET is just one of PUT's action verbs. Here's a list of all the ones we can use:

XOR is the default value. It is used most often for animation.

OR will "superimpose" the image onto whatever exists underneath the PUT.

AND will only PUT pixels to the screen where existing pixels match up with those in the GET buffer.

PSET will PUT exactly what is in the GET buffer to the screen.

PRESET is the same as PSET except that the image is reversed.

Change the PSET in Line 130 to PRESET and Run.

Replace PRESET with one of the other action verbs to see what you can come up with.

Another Chapter Completed!

Take some time out to experiment on your own with all these graphics commands. We have explored enough to keep busy for some time.

Learned In Chapter 45

Commands	Statements
CIRCLE	PTAB
GET	POINT
PUT	

Chapter 46

Introduction To Data Processing

Data Processing is an important computer application. An example of DP would be storing all the names in the telephone book, then recalling any name, address and phone number very quickly. Sorting, alphabetizing, adding and deleting vast quantities of data, inventory (merchandise), general ledger (money), mailing lists (people), recipe files, or other records, replace intricate and complex calculations as the computer's purpose. DP is not the same thing as programming but is simply an application which requires specialized programming with emphasis on disk files used for other than just SAVEing and LOADing the program itself.

At the heart of Data Processing is the accumulation of data in what is known as a DATA FILE. The DATA may be similar to the data we know how to store in DATA LINES, but the quantity is often so large the entire memory of the computer is not enough to hold it. Thus the need for "external storage," as on a disk.

Up to now we've relied on BASIC's numeric variables, string variables and DATA Lines to store the data the programs need. This has 2 severe limitations:

1. The Computer's memory may not be large enough to hold all the data (for example, an inventory list).

2. When the Computer is turned off, the values of all variables are lost.

Diskette data files solve both problems. Virtually endless quantities of information can be stored on an endless stack of disks and retrieved later at will, just as we now SAVE and reLOAD programs. Besides SAVE and LOAD, we need to learn more special statements.

OPEN The Door, Richard

The first is the OPEN statement. OPEN handles all the details of creating a new DATA file. It communicates 3 things to the system:

1. What we plan to *do* with the file, i.e., INPUT data from it or PRINT information into it.

2. What *buffer number* (1 - 255) to assign to the file. (More on that in a second ...)

3. The file's *name.*

Type:

```
30 OPEN "TESTDATA" FOR OUTPUT AS 1
```

but don't RUN yet. OUTPUT means that we intend to OUTPUT information from the Computer to a data file on the disk named "TESTDATA." The Program Line reads, "OPEN the data file named TESTDATA FOR OUTPUTting to disk." If we wanted to INPUT information back from disk to memory, we would use INPUT instead of OUTPUT. We'll learn how to INPUT in a minute.

File Buffers

Line 30 assigns "buffer" number 1 to the file and names it TESTDATA. Any number from 1 to 255 may be used.

A file buffer is a small part of the Computer's memory which is assigned to act as a Traffic Director for information traveling to and from the Computer and a disk file. All 255 buffers are available at any one time on the Mac.

The OPEN statement is our written instruction to the Computer to OUTPUT its data to TESTDATA through buffer #1 until notified otherwise. In addition, the OPEN statement sets the file buffer size to a default value of 128 bytes. The size can be set to a different value by placing

```
LEN = N
```

at the end of the OPEN statement. The LENgth value (N) can be any value from 1 to 32767.

OPEN simply assigns the data file TESTDATA a buffer number (1 in this case) and prepares TESTDATA for either OUTPUT (as in this case) or INPUT from disk to Computer memory. TESTDATA will stay on the disk indefinitely under that name.

CLOSE The Barn Door

The opposite of OPENing a file is CLOSEing it. It's a good habit to CLOSE all files when they aren't being used. And we could all use an extra good habit or two. Better add:

```
50 CLOSE 1
```

Line 50 will CLOSE the file OPENed in Line 30. (Remember, don't RUN yet!)

CLOSE Options

The CLOSE statement severs the association of a file with its assigned buffer.

Without getting too far ahead of ourselves, it's worth noting that if any left-over or stray data is still in the file's buffer, that data is saved to disk when the file is CLOSEd. For example:

`CLOSE 1,3`	CLOSEs only files numbered 1 and 3 and saves to disk any data left in buffers 1 and 3.
`CLOSE N`	CLOSEs only file number N and saves to disk any data left in buffer N.
`CLOSE`	CLOSEs all files currently OPEN and saves any data left in all of the buffers to the correct disk file.

Most of the time, we will simply use:

```
CLOSE
```

since it CLOSEs everything in sight, secures all data from the file buffers and writes it all to the disk. To fully understand the value of the CLOSE statement, we need to take a closer look at the way data is transferred to the disk.

To send data to a disk file, we need 2 additional BASIC statements:

PRINT

Our old friend PRINT directs output to the screen, and LPRINT directs it to the printer. The third member of the family is PRINT # which sends output to a disk file.

Remember the file buffer number that we assigned in the OPEN statement? It's used by the PRINT # statement to direct output to that buffer for transfer to the disk file. We assigned buffer #1 to the file TESTDATA, so we use:

```
PRINT #1
```

to send information to the TESTDATA file.

But what do we want to PRINT, and how do we do it?

Writing DATA into a "sequential file" is very similar to writing data to the screen. We can think of a sequential file as one

V....E....R....Y.....

long stream of data.

Numbers, strings and variables can be separated by commas or semicolons, and these "formatters" have precisely the same effect on the disk file as they do on the screen or printer output. If the formatting is unusually complex and we have enough disk space, we can even use PRINT USING just like we learned for the screen and printer. For example:

```
PRINT #1,USING "##.##";A
```

(See Chapters 36 and 37 for a review if your PRINT USING skills have grown dull.)

Insert the following Lines, and Run:

```
10 REM * SEQUENTIAL FILE PROGRAM *
20 A = 1 : B = 2 : C = 3
40 PRINT #1,A,B,C
```

This is what happened:

Line 20 assigned values to variables A, B and C.

Line 30 OPENed a file on disk named TESTDATA and assigned buffer #1 to OUTPUT data to that file.

Line 40 PRINTed the values in A, B, and C to buffer #1 (at this point it is still not on disk).

Line 50 transferred the data from buffer #1 to the TESTDATA file and CLOSEd it.

We now have a "permanent" record which can easily be read back into the Computer or any other computer which is compatible. Note that the variables A, B and C were *not* written onto the disk -- just the *values* of those variables (in this case, 1, 2 and 3).

INPUT

The next step in this learning process is to INPUT that data from disk back into memory. After all, the only reason to store something on disk is so we can retrieve it later.

Once file buffer number 1 is CLOSEd in Line 50, it is no longer associated with disk file TESTDATA. It's free to be used with any file specified in a new OPEN statement. Add:

```
70 OPEN "TESTDATA" FOR INPUT AS 1
```

We are reOPENing the file TESTDATA using buffer number 1, but this time for INPUTing. (Any other valid buffer number will work as well.) It's important to remember with sequential files that we must first CLOSE the file, then reOPEN it when switching from reading to writing, and vice versa.

To read the contents of the sequential DATA file, insert these Lines in their proper order:

```
60 PRINT "THE NUMBERS";A;B;C;" ARE WRITTEN
   ON DISK."
```

```
80 INPUT #1,A,B,C
90 PRINT "THE DATA HAS BEEN READ FROM DISK."
100 PRINT "A =";A,"B =";B,"C =";C
110 CLOSE
```

...and Run.

The Computer says:

```
THE NUMBERS 1 2 3 ARE WRITTEN ON DISK.
THE DATA HAS BEEN READ FROM DISK.
A = 1          B = 2          C = 3
```

Remember to move the List window or close it to see the entire display.

If yours doesn't look that way, here is a complete program listing:

```
10 REM * SEQUENTIAL FILE PROGRAM *
20 A = 1 : B = 2 : C = 3
30 OPEN "TESTDATA" FOR OUTPUT AS 1
40 PRINT #1,A,B,C
50 CLOSE 1
60 PRINT "THE NUMBERS";A;B;C;"ARE WRITTEN
   ON DISK."
70 OPEN "TESTDATA" FOR INPUT AS 1
80 INPUT #1,A,B,C
90 PRINT "THE DATA HAS BEEN READ FROM DISK."
100 PRINT "A =";A,"B =";B,"C =";C
110 CLOSE
```

Here's what happened:

Line 70 OPENed the TESTDATA file for INPUT via buffer number 1.

Line 80 INPUT # the three numbers from disk into buffer number 1 where the values were assigned to variables A, B and C.

Line 90 PRINTed a reassuring message.

Line 100 PRINTed the data values that were read from disk.

Line 110 CLOSEd file buffer number 1, the only one OPENed.

APPEND

APPEND allows us to OPEN a sequential file and add data to the end of it.

To APPEND more information, add these Lines:

```
200 OPEN "TESTDATA" FOR APPEND AS 1
210 PRINT #1, "MORE INFO"
220 CLOSE 1
230 OPEN "TESTDATA" FOR INPUT AS 1
240 INPUT #1, A,B,C,D$
250 PRINT A,B,C,D$
260 CLOSE
```

...and Run.

Line 200 OPENs TESTDATA so something can be done to or with it and assigns it to buffer 1.

Line 210 PRINTs `MORE INFO` at the end of what's already in file TESTDATA.

Line 220 CLOSEs it again.

Line 230 reOPENs the TESTDATA file to INPUT the 4 pieces of data in it, and Line 240 PRINTs them. Note that since the new data is made up of letters, it is a string variable.

Line 260 CLOSEs the file for the last time.

EXERCISE 46-1: Write a program that stores a shopping list of five items on disk. The program should ask for each item and then write it to the disk. HINT: Use a FOR-NEXT loop. Be sure to CLOSE the file when you are through writing to it.

EXERCISE 46-2: Add a second part to Exercise 46-1 that reads the five items from the disk and displays them to the screen. SAVE the entire program (both output and input) As SHOPLIST.

EOF

Until now, we've been dealing with a precisely known quantity of data, but most of the time, the amount of data in the file is not known. How would we know where the end of the file is? The EOF, or End Of File, function is the answer.

Type in the following NEW program:

```
10 REM  * EOF DEMO *
20 RANDOMIZE
30 OPEN "UNKNOWN" FOR OUTPUT AS 1
40 FOR N = 1 TO INT(10*RND)
50  PRINT #1, "DATA";N
60 NEXT N : CLOSE
```

Note that Line 50 PRINTs 2 pieces of DATA to file, the word "DATA" and the value of N. Due to the deliberate use of the RND function in Line 40 to determine how many FOR-NEXT loops will be executed, we don't know how many data pairs will be written to the file named UNKNOWN.

To read the data back in from UNKNOWN into memory and to display it, add these Lines. We'll use GOTO in Line 110 to keep reading in the DATA

until it runs out. Since the information was written to disk in "data pairs," we INPUT # it back in with a single variable A$ in Line 90. Add:

```
70 OPEN "UNKNOWN" FOR INPUT AS 1
80 REM
90 INPUT #1,A$
100 PRINT A$
110 GOTO 80
120 CLOSE
```

Run the program.

Ack! An error! BASIC won't allow us to just keep reading DATA from disk until it runs out, any more than it will permit us to do it from DATA Lines. The error message "Input past end" tells the story. If we don't know the exact length of the file, we must test for the EOF condition.

EOF works this way. If we are at the end of file, the numeric value of EOF equals −1, paradoxically known as "true." If there's still more data to be read, EOF will equal 0, called "false." From these little truths, EOF can be used in a test, as follows:

Change Line 80 to:

```
80 IF EOF(1) THEN 120
```

The 1 in parentheses is the buffer number assigned to the file when it was OPENed. Line 80 reads, "If we have reached the End Of the File (EOF is true), then branch to Line 120." The EOF function can "look ahead" to signal when INPUT # is at the End Of the File.

Save this program As `EOFTEST` ... and Run.

Whew! Nothing like a smooth running program to make your day!

EXERCISE 46-3: Write a program that asks for the names and ages of several people. Use a GOTO loop to enter the data. After all the names are entered (signified by typing "DONE" or some other key word), CLOSE the file, then reOPEN it for INPUT to read the names and ages back into the Computer. Use EOF to avoid reading past the file's end.

Learned in Chapter 46

Statements

OPEN
CLOSE
PRINT #
PRINT #, USING
INPUT #
EOF (End Of File)

Miscellaneous

Data files (Sequential)
File numbers
File buffers
Buffer size
Output
Input
Append

Chapter 47

Advanced SAVEing, MERGEing, And CHAINing

Everyone type in this New program:

```
10 REM LINE 10
20 REM LINE 20
40 REM LINE 40
```

We know this program is not destined for fame, but SAVE it on disk anyway. Each program SAVEd to disk becomes a FILE. Like any file, it is labeled with a file name. We will call this program FIRST. Save As:

```
FIRST
```

BASIC programs can be Saved on disk in either of 2 "formats." Unless we specify otherwise, the so-called "Compressed (or binary) format" is used.

1. In the *binary* format, everything that can be abbreviated is stored in a shortened form. All numbers except those enclosed in quotes are stored in a minimum number of bytes, with BASIC keywords like PRINT and GOTO stored as special shorthand "codes." This format is the one usually used and is fine for most purposes since it conserves disk space. This is all "invisible" to the user.

2. But there are times when we will sacrifice a little disk space for the luxury of saving a program or data on disk in the "character for character" format. It is called the "Text (or ASCII) format."

Text formatted files have several special purposes.

   1) They can be loaded directly into word processing programs or other "applications" software.

2) Files in Text format can be sent over phone lines to other computers. Electronic mail is here!

3) And, the Text format can be used to MERGE two files -- hooking them end-to-end. Using Text format, we can MERGE a useful routine into several programs without retyping it. (Remember our SGN subroutine?)

Merging Files

Let's try a MERGEr right now. Type this New program:

```
30 REM THIS LINE GOES BEFORE LINE 40
40 REM THIS LINE REPLACES LINE 40
50 REM THIS LINE APPEARS AFTER LINE 40
```

and Save it in Text As

```
SECOND
```

by following this simple procedure. Select Save As from the File menu as usual. Type in the file name, SECOND, but before clicking the **OK** box (or hitting Return), move the pointer to ○ **Text** and click. A black dot will appear within the ○ meaning our file will be saved in Text. Now click the **Save** box.

With that done, we can MERGE the two programs. Load the original program back into memory by selecting Open... from the File menu and either double-clicking on the file name FIRST or by clicking once on the file name and then clicking the **Open** box.

Check the Listing to be sure only the FIRST program is in memory. Now bring in the next program by entering the Command window and typing:

```
MERGE "SECOND"          Return
```

Check the List window to verify that the two programs were MERGEd.

```
10 REM LINE 10
```

```
20 REM LINE 20
40 REM LINE 40

30 REM THIS LINE GOES BEFORE LINE 40
40 REM THIS LINE REPLACES LINE 40
50 REM THIS LINE APPEARS AFTER LINE 40
```

Of course, it worked! We have *new* Lines 30, 40 and 50 and the original Lines 10, 20 and 40.

Observe that the FIRST program did not have to be in Text format, only the second one drawn in for MERGEr. It's now a simple task to cut out the original Line 40 with the Editor.

```
10 REM LINE 10
20 REM LINE 20
30 REM THIS LINE GOES BEFORE LINE 40
40 REM THIS LINE REPLACES LINE 40
50 REM THIS LINE APPEARS AFTER LINE 40
```

The combined program can be Saved as usual under any name. After selecting Save As... from the File menu, be sure to click the Compressed circle before Saving As:

```
MERGER
```

Removing Files From The Diskette

The 2 program files, FIRST and SECOND, are now combined into a MERGEd file, MERGER, and Saved on disk. They are no longer necessary. Right?

What's that about a safety copy of the program?

Yes, we *should* keep an extra copy of any important program and right now, FIRST and SECOND are the only protection we have if MERGER should


W-W-WHAT HAPPENED?!
GASP
I MERGED WITH A MACK TRUCK!
WHEEZE

somehow get zapped. What if we erased them and a nasty electrical spike sizzled the MERGER file?

A safety backup copy is normally made on a different diskette. Since we are only risking 5 Lines of code at this point, we'll gamble with Murphy's law and make our safety copy of the MERGER program on the *same diskette.*

Since we can't Save the same program on the same diskette under the same name, we have to give it another name. Rather than have to remember an excessive number of names, just Save As:

```
MERGER/BAK
```

By appending the "/" and the three letter "extension" "BAK," we create a second file with the same "first name" (MERGER) as our original. The extension "BAK" reminds us that the program is a safety BAcKup, and thus a duplicate, not a different program. /SAF for SAFety, /COP for COPy, /NO1 for Number 1 and other extensions can work as well.

KILL - KILL!

Now we can erase the 2 original files with a clear conscience. In the Command window, type:

```
KILL "FIRST"          Return
KILL "SECOND"         Return
```

Check the files on the disk by typing:

```
FILES          Return
```

to make sure FIRST and SECOND have disappeared.

The KILL instruction doesn't actually "erase" FIRST and SECOND from the diskette. It simply removes their *names* from the list of files. The result is the same, however; if they can't be found, they can't be used, (sort of like having an unlisted telephone number).

(To answer the question in some readers minds, *YES*, with a special UTILITY program we could conceivably patch up the list of files and retrieve our "dead" files. Of course, if another new file is Saved first and it happens to use the same place on the disk, the file(s) is lost for good. For all intents and purposes, consider the files KILLed.)

We can also reNAME programs from BASIC. Suppose we want to change the name of the backup copy to NEWMERGE. No problem. Just type:

```
NAME "MERGER/BAK" AS "NEWMERGE"
```

Check the Files again to be sure that MERGER/BAK is gone and NEWMERGE took its place. (Note that NEWMERGE has no /BAS or /BAK since none was specified.)

CHAINing

The ability to CHAIN programs is very powerful. Not only can we Run one program by calling it from another, but the values of the variables can be transferred from one program to the next without being reset to 0. Try this New program:

```
10 REM  * THIS IS THE FIRST PROGRAM
30 PRINT "PROGRAM ONE"
40 M$ = "MACINTOSH COMPUTER"
50 A = 20
60 PRINT "M$ = ";M$
70 PRINT "A =";A
80 RUN "TWO"
```

Save As ONE, but **do not Run it!** Choose New, and enter these Lines:

```
10 REM  * THIS IS THE SECOND PROGRAM  *
20 PRINT : PRINT "PROGRAM TWO"
30 PRINT "M$ = ";M$
40 PRINT "A =";A : PRINT
```

Save it As TWO, but *do not Run*. Now, from the Command window type:

```
RUN "ONE"
```

Note very carefully that the String and Numeric variables were *not* carried over from the first to the second program. We used a RUN statement to execute TWO (Line 80 in program ONE). Remember RUN clears the screen and initializes all variables back to 0 or null. Now type:

```
LOAD "ONE"
```

(or use Open... from the File menu)

and change Line 80 to:

```
80 CHAIN "TWO",, ALL
```

Select Save from the File menu

...and Run.

Wow! The variables passed from ONE to TWO.

By adding the ALL option, program ONE passed ALL variable data to program TWO. Now let's see what gets placed between the two commas.

Add this Line to program TWO:

```
50 CHAIN "ONE",100,ALL
```

and Save. Line 50 will LOAD program ONE and begin execution at Line 100. If we don't specify a Line number, it would start ONE running at its first Line again, and we would be in an endless loop, or endless CHAIN. If the starting Line number is omitted, as we did in Line 80 of the ONE program, we still have to use the commas as place holders.

Now LOAD ONE back in, and change Line 80 to:

```
80 CHAIN "TWO"
```

and add the following:

```
20 COMMON M$
99 STOP
100 PRINT "WE ARE NOW BACK IN 'ONE'"
110 PRINT "M$ = ";M$
120 PRINT "A =";A
```

Save

...and Run.

Here's what happened. Program ONE ran up thru Line 80, where it CHAINed to program TWO. Only M$ was forwarded from ONE.

Since the ALL option was removed from Line 80, all variables were not carried over to the CHAINed program. But, by adding Line 20, we made M$ COMMON to both programs. Variable M$ was forwarded, but A was not. Program TWO ran thru Line 50 where it CHAINed back to Line 100 of ONE. ALL variables were forwarded, but A=0 in TWO, so that's what was printed this 2nd time by ONE.

The Line 99 STOP will never be executed and is not necessary. It was placed there as a reminder that program ONE in this case is really executed as 2 different programs under the same name.

The way to forward only selected variables without CHAINing them all is:

```
20 COMMON M$,A      (etc...)
```

Do it, Save and Run again.

LOAD program ONE and then program TWO, and study them very carefully.

CHAINing and MERGEing have real programming value. What you have learned here will satisfy most programming needs. If you need to use more "advanced" CHAIN and MERGE features, refer to the factory manual.

Learned In Chapter 47

Statements	Commands
MERGE	KILL
CHAIN	
COMMON	
NAME	

PART 8

PROGRAM CONTROL

Chapter 48

Flowcharting

Most of the programs written for this book were simple; but they met simple, specific needs. Suppose we want to write a program to play chess or bridge, evaluate complicated investment alternatives, keep records for a bowling league or a small business, or do stress calculations for a new building? How do we approach writing such a complex program?

We break down a complex program into a series of smaller programs. This is called *modular programming*, and the individual programs are called *modules*. But how are the modules related -- and how do we write them, anyway?

Module is just a 75-cent word for "section" or "building block."

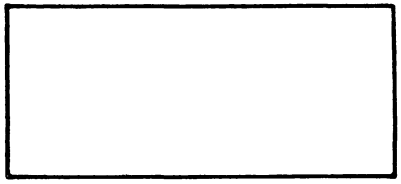
One way to plan a program is to make a picture displaying its logic. Remember, a picture is worth a thousand words (or is it the other way around)? The picture that programmers use is called a *flowchart*.

Flowcharts are most helpful when kept simple. A cluttered flowchart is hard to read and usually isn't much more helpful than an ordinary program LISTing. A good flowchart is also helpful for "documentation" to give us (or others) a picture of how the program works -- for later on, when we've forgotten.

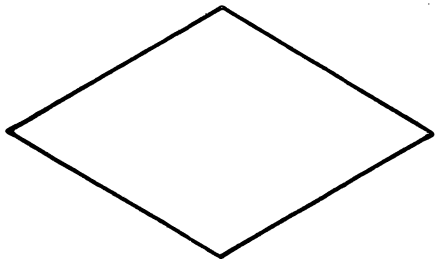
Flowcharts are so widely used that programmers have devised standard symbols. There are many specialized symbols in use, but we will examine only the most common ones.

TERMINAL BLOCK
(means Begin or End)

PROCESSING BLOCK
(something the Computer does without making any decisions)



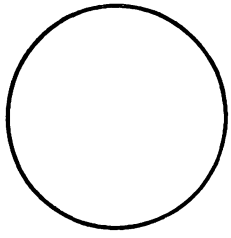
DECISION DIAMOND
(branches off in different directions, depending on the decision it makes)



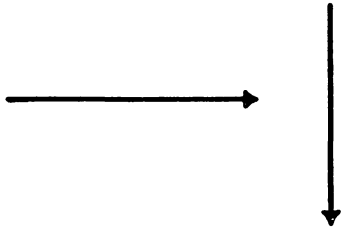
Each decision point asks a question such as *"Is A larger than B?"* or *"Have all the cards been dealt?"* The different branches are marked by YES or NO.

Another useful symbol is:

CONTINUATION
(usually contains a number which corresponds to a number on another page if the flowchart is too large for a single sheet)



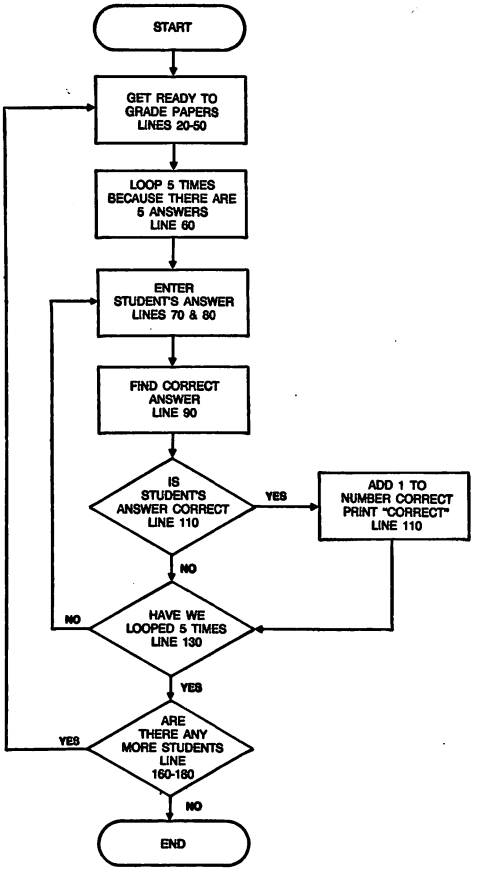
CONNECTOR ARROWS
(indicate the direction in which program execution proceeds)



There are no hard-and-fast rules about what goes into a flowchart and what doesn't. A flowchart is supposed to help, not be more work than it's worth. It helps us plan the *logic* of a program. When it stops helping and makes us feel like we're back in arts and crafts designing mosaics, we've gone as far as the flowchart will take us (or more typically, it's passed its point of usefulness).

Suppose we want to grade a 5-question test by comparing each of the *students'* answers with the *correct* answer. We can put the correct answers in a DATA statement in the program, enter a student's answers through the keyboard, compare (grade) them, then PRINT the % of correct answers. This procedure can be repeated until all the students' papers are graded.

The flowchart might look like this:



This flowchart has three decision diamonds. In the first, the Computer determines if an answer is correct. In the second, the Computer determines if all the questions in a single student's paper have been graded. The third terminates execution when all tests have been graded.

EXERCISE 48-1: Using the flowchart as a guide, write a program that grades a test having five questions.

For more complicated problems, we may subdivide the flowchart into larger modules. A *master flowchart* will show the relationship between the flowcharts of individual programs.

For example, let's say we want to write a program that calculates the return on various investments. The options might be:

1 - CERTIFICATE OF DEPOSIT

2 - BANK SAVINGS ACCOUNT

3 - CREDIT UNION

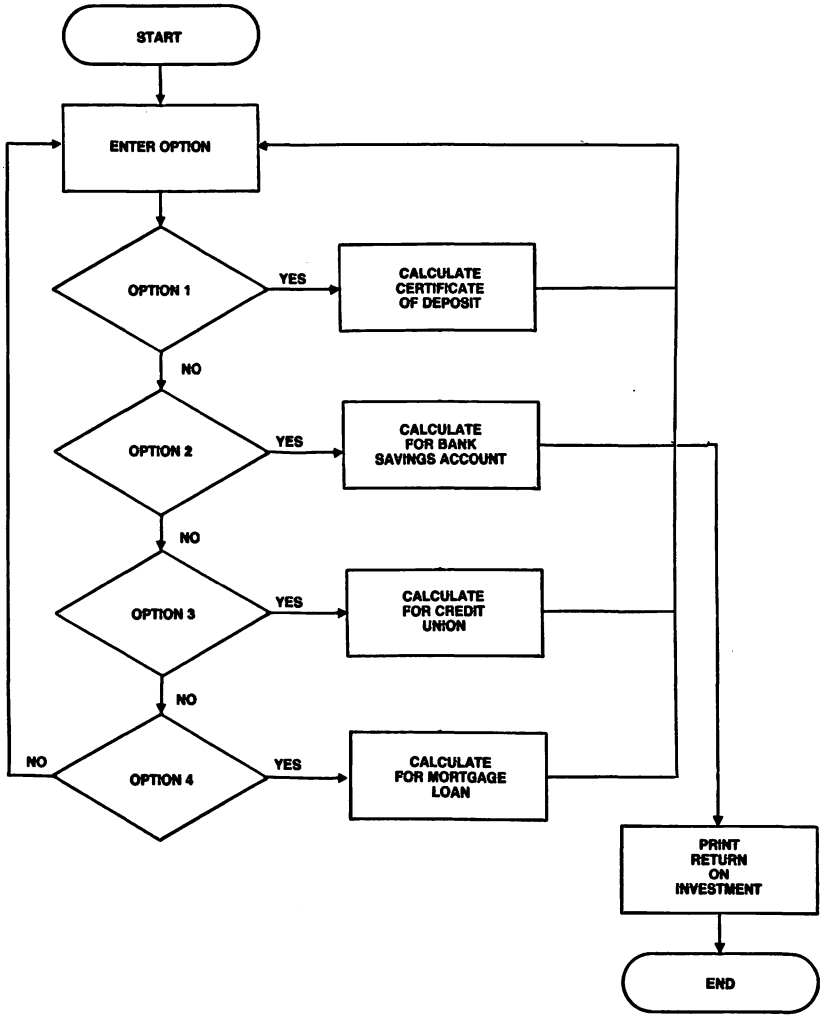
4 - MONEY MARKET FUND

The main (or Control) program will select one of these 4 options using an INPUT question, execute the correct sub-program, and PRINT the answer. Its flowchart might be as shown on the next page.

We can now flowchart each of the individual programs in the blocks separately. The Certificate of Deposit program would, for example, have to contain the rate of return, size of deposit, and maturity. The order in which that program INPUTs data and performs the calculations would be specified in its own flowchart.

EXERCISE 48-2: Write the master program as flowcharted, with a branch to a program to calculate the return on a Bank Savings Account paying simple interest.

EXERCISE 48-3: Choose a program from an early Chapter and design your own flowchart.



Learned In Chapter 48

Miscellaneous

Flowcharting

Chapter 49

Debugging Programs

Quick -- The Raid!

The Computer has given us plenty of nasty messages. We know something's wrong, but it isn't always obvious exactly where, or why.

How do we find it? The answer is simple -- *be very systematic. Even experienced programmers make lots of silly mistakes ... but experience teaches how to locate mistakes quickly.*

Hardware, Cockpit Or Software?

The first step in the "debugging" process is to isolate the problem as being either:

1. A hardware problem,
2. An operator problem, or
3. A software problem.

Is It Further To Cupertino Or By Bus?

Starting with the least likely possibility -- is the Computer itself malfunctioning? Chances are very high that the Computer is working perfectly. There are several very fast ways to find out.

A. Type:

```
PRINT FRE(0)
```

in the Command window.

If there is no program loaded into memory, the answer should be:

```
21000
```

Or the correct value previously noted for your system. If there is a program loaded, the answer should be some lesser value.

If the answer is too large (assuming, of course, you have not added more memory), there may be trouble. Or, it's possible that the answer is a *negative* number. Trouble.

Possible Solution

In either of the above cases, select Quit, Eject the disk and shut the Computer off. (Or, as they say in the big time, "Take it all the way down.") Let it sit for a full minute before turning it on.

> Yes, any program in memory will be lost, but at this point it's probably shot anyway. You could *try* to SAVE it before turning OFF the machine if it makes you feel any better.

Turn the machine back ON, and try the PRINT FRE(0) test again. If the results are the same, there is probably a chip failure that will require professional troubleshooting and replacement.

B. One Last Try

Before full panic sets in, choose New and enter this program. It assigns almost every free memory location in RAM a specific value, then reads that value back out, comparing it to adjacent values.

Type:

```
10 WIDTH 60
20 DIM A(1730)
30 FOR X = 1 TO 1730 : A(X) = X : NEXT X
40 FOR Y = 1 TO 1730 : PRINT A(Y);
```

```
50  IF A(Y) - A(Y-1) <> 1 THEN PRINT "BAD" :
    BEEP
60 NEXT Y
```

...and Run.

After a *short* wait, the monitor should display:

1 2 3 4 5 6 7 8 (etc. through the value of 1730)

If the "horn honks" and "BAD" appears, we *may* have found the problem ... a bad memory chip.

Type this test program into the Computer, Save As MEMTEST. Try it out *before you need it, and hope it will never be used.*

Video Display Problems?

The Video Display is very similar to its counterpart in a television set. It has an adjustment for brightness under the left side of the Macintosh's front panel above the ☼ symbol.

Idiot Here -- What's Your Excuse?

Of course, *you* don't make silly mistakes!

Now that's settled.

1. Is everything plugged in? Correctly? Firmly?

2. Is the printer turned ON and ON-Line?

If so...

go walk the dog, then check it all over again.

If...Then

If the trouble was not found in the cockpit or with the hardware, there is

probably something wrong with the program. Dump out the troublesome program. LOAD in one that is known to work, and Run it as a final hardware and operator check.

Common Errors

Here are some of the common sources of "computer-detected errors."

1. Assume the error is in a PRINT or INPUT statement.

   Did you:

   a. Forget one of the needed pair of quotation marks?
      EXAMPLE:

```
10 PRINT "ANSWER IS, X : GOTO 5
```

      ERROR: No ending quotation mark after IS

Yes, I know it's Ok if the missing quote is the last character in the Line.

---

   b. Use an illegal variable name?
      EXAMPLE:

```
10 INPUT 6G
```

      ERROR: Variable names must begin with a letter.

   c. Use a Line number larger than 65529?
      EXAMPLE:

```
65530 PRINT "BAD LINE NUMBER."
```

   d. Accidentally have a double quotation mark in the text?
      EXAMPLE:

```
10 PRINT "HE SAID "HELLO THERE."
```

   e. Type a Line more than 255 characters long?

   f. Misspell PRINT or INPUT? *(It happens!)*

   g. Accidentally type a stray character in the Line, especially an extra comma or semicolon?

2. If the error is in a READ statement, almost all the previous possibilities apply, plus:

   a. Is there really a DATA statement for the Computer to read? Remember, it will only read a piece of DATA once unless it is RESTOREd.
   EXAMPLE:

```
10 READ X,Y,Z
20 DATA 2,5
```

   ERROR: There are only two numbers for the Computer to read. If we mean for Z to be zero, we must say so.

```
20 DATA 2,5,0
```

3. If the bad area is a FOR-NEXT loop, most of the previous possibilities apply, plus:

   a. Is there a NEXT statement to match the FOR?
   EXAMPLE:

```
10 FOR A=1 TO N
```

   ERROR: Where's the NEXT A?

Some of these FOR-NEXT loop errors won't trigger actual error messages; the program may just wind up in an endless loop.

   b. Do you have all the requirements for a loop -- a starting point, an ending point, a variable name, and a STEP size if it's not 1?
   EXAMPLE:

```
10 A=1 TO N
```

   ERROR: Must have a FOR and a NEXT.

   c. Did you accidentally nest 2 loops using the same variable in both loops?
   EXAMPLE:

```
10 FOR X=1 TO 5
20  FOR X=1 TO 3
30   PRINT X
40  NEXT X
```

```
50 NEXT X
```

ERROR: The nested loops must have different variables.

d. Does a variable in a loop have the same letter as the loop counter?
EXAMPLE:

```
10 A=22
20 FOR R=1 TO 5
30  R=18
40  Y=R*A
50  PRINT Y
60 NEXT R
```

ERROR: The value of R was changed by another R inside the loop, and NEXT R was overRUN, since 18 is larger than 5.

e. Are the loops nested incorrectly with one not completely inside the other?
EXAMPLE:

```
10 FOR X=1 TO 6
20  FOR Y=1 TO 8
30   PRINT X,Y
40  NEXT X
50 NEXT Y
```

ERROR: NEXT Y must come before NEXT X.

4. If the goofed-up statement is an IF-THEN or GOTO

a. Does the Line number or Line label specified by the THEN or GOTO really exist? Be especially careful of this error when eliminating a Line in the process of "improving" or "cleaning up" a program.

5. The error comes back as "Out of memory" but PRINT FRE(0) indicates there is room left. If you are using an array and get an error, remember, **extra room (up to hundreds of bytes) has to be left for processing.** You have probably overRUN the amount of *available* memory.

6. The error comes back as "Subscript out of range."

    a. Did you forget to DIMension an array containing more than eleven elements?

7. The error comes back as "Illegal function call."

    a. Did you exceed the limits of one of the built-in functions?

8. Did one of the *values* on the Line exceed the maximum or minimum size for numbers?

To find out whether you did any of these things, PRINT the values for all the variables used in the offending Line. If you still don't see the error, try carrying out the operations indicated on the Line. For example, the error may occur during a multiplication of two very large numbers.

PRINT the operation in the Command window.

These certainly aren't all the possible errors one can make, but at least they give some idea where to look first. Since we can't completely avoid silly errors, it's necessary to be able to recover from them as quickly as possible.

By the way ... a one-semester course in beginning typing can do wonders for your programming speed and typing accuracy.

From The Ridiculous To The Sublime:

All the Computer can tell us is that we have (or have not) followed all of its rules. Assuming we have, the Computer will not protest even if we're asking it to do something that's quite silly and not at all what we intended. It will dutifully put out garbage all day long if we feed it garbage -- even though we follow its rules. Remember GIGO?

GIGO stands for Garbage In, Garbage Out.

If the program has no obvious errors, what might be the matter?

Typical "unreported" errors are:

1. Accidentally reinitializing a variable -- particularly easy when using loops.
   EXAMPLE:

```
10 FOR N=1 TO 3
20  READ A
30  PRINT A
40  RESTORE
50 NEXT N
60 DATA 1,2,3
```

2. Reversing conditions, i.e. using "=" when we mean "<>," or "greater than" when we mean "less than."

3. Accidentally including "equals," as in "less than or equals," when we really mean only "less than."

4. Confusing similarly named variables, particularly the variable A, the string A$, and the array A(X). *They are not at all related.*

5. Forgetting the order of program execution -- from left to right on each Line, but multiplications and divisions always having priority before additions and subtractions. Intrinsic functions (INT, RND, ABS, etc.) have priority over everything.

6. Counting incorrectly in loops. `FOR I=0 TO 7` causes the loop to be executed *eight*, not seven, times.

7. Using the same variable accidentally in two different places. This is okay if we don't need the old variable any more, but disastrous if we do. Be especially careful when combining programs or using the special subroutines.

But how do we spot these errors if the Computer doesn't point them out? Use common sense, and let the Computer help. The rules are:

1. Isolate the error. Insert temporary "flags." Add STOP, END, and extra PRINT statements until you narrow the error down to one or two Lines.

EXAMPLES OF USEFUL FLAGS:

```
299 PRINT ,"LINE #299"
399 IF X<0 THEN PRINT "X OUT OF RANGE AT
    #399" : STOP
```

Line 299 checks whether the Line immediately following Line 299 is executed. Line 399 might be used to locate the point where X goes out of range.

Although the details are different in every program, these techniques can be easily applied.

2. Make "tests" as simple as possible. Don't add "enhancements" until you've found the problem.

3. Check simple cases by hand to test the logic, but let the Computer do the hard work. Don't try to wade through complex calculations with pencil and paper. You'll introduce more new mistakes than you'll find. Use the Command window or a separate hand calculator for that work.

4. Remember that we can force the Computer to start a program at any Line number. Just enter the Command window and type:

```
GOTO ###
```

This is a useful tool for working back through a program. Give the variables acceptable values using statements entered in the Command window, then GOTO some point midway through the program. If the answers are what are expected, then the error is *before* the "test point." Otherwise, the error is after the test point.

5. Remember if you need to look at two sections of the program, you can LIST one section in the main List window and the other section in the second List window.

6. Practice "defensive programming." Just because a program "runs okay," don't assume it's dependable. Programs that accept INPUT data and process it can be especially deceptive. Make a point of checking a new program at all the critical places.

EXAMPLES: A square root program should be checked for INPUTs less than or equal to zero. Math functions should be checked at points where the function is undefined, such as TAN(90°).

Beware Of Creeping Elegance

Programs grow more elegant with the ego reinforcement of the programmer. This "creeping elegance" increases the chance of silly errors. It's fun to let the mind wander and add some more program here and some more there, but it's also easy to lose sight of the program's purpose. It is at times like this when the flowchart is ignored and the trouble begins. Nuff said.

Learned In Chapter 49

Miscellaneous

Defensive programming
Computer-detected errors
Flags
Hardware checkout procedures

Chapter 50

Chasing Bugs

We have seen that the EDITor is a powerful aid in changing programs once we find out what is wrong. In this and the next Chapter we will learn how to use built-in diagnostic tools to help hunt down the errors.

TRON/TROFF

The simplicity but power of TRON/TROFF is awesome. Enter this New program:

```
10 INPUT "PRESS RETURN TO WATCH TRON";X$
20 CLS
30 FOR N = 1 TO 5
40   PRINT "SEE TRON RUN"
50 NEXT N
60 GOTO 10
```

...and Run to be sure it's OK.

Stop the program, then choose Trace On from the Run menu, or enter the Command window and type:

```
TRON          Return
```

(which stands for TRacer ON), then Run.

After pressing Return, wait for the PRESS RETURN request to reappear and press ⌘. to Stop the program and ⌘L to show the listing. The INPUT statement in Line 10 is enclosed in a box indicating this is the Line the Computer was sitting on when the program was stopped.

OH, COME NOW!

Run the program again and after pressing Return, quickly press ⌘ . while the Computer is printing SEE TRON RUN. If you were quick enough, a Line within the FOR-NEXT will be enclosed in a box. Again, the Computer is showing us where it was when it stopped.

Go to the Run menu again, and this time choose Trace Off, or enter the Command window and type:

```
TROFF          Return
```

(for TRacer OFF) and Run.

The Tracing has stopped when the Listing appears, and it's business as usual. TRON is the very essence of simplicity.

TRON and TROFF can be imbedded as program statements as well as used as BASIC commands giving greater flexibility in program troubleshooting.

Stepping Through The Program

With large, complex programs, it is often difficult to Stop execution at the precise point where we suspect a problem. With the Step feature, we can watch each program Line as it is being executed and see the results on the screen.

Run our resident program. Select Step from the Run menu, or press ⌘T, then press Return as requested by the program. Once again, as with TRON, the INPUT statement in Line 10 is enclosed in a box. Press ⌘T and notice that the screen cleared as the CLS statement was enclosed in a box.

Continue pressing ⌘T to step through the remainder of the program while watching the action on the screen. Remember to press Return when the INPUT Statement is executed. Pressing ⌘T does not take the place of pressing Return.

Imagine its value in a program with dozens or hundreds of program Lines all tangled up with IF-THEN's, ON-GOTO's, etc. The errors that drive us wild are those we can't see.

Learned In Chapter 50

Commands/Statements	Menu
TRON	Run
TROFF	Trace On
	Trace Off
	Step ⌘T

Chapter 51

Chasing The Errors

Microsoft BASIC provides 46 different ERROR messages numbered between 1-38 for the Elementary and Intermediate BASIC we have learned, and between 50-74 for Advanced or Disk BASIC. There are so many we need a separate Chapter plus an Appendix just to understand what they mean.

Let's quietly tiptoe into the hall of ERRORs by typing this New little test program:

```
10 REM * TESTING ERROR CODES *
20 INPUT "CHECK WHICH ERROR CODE";N
30 ERROR N
```

Run the program a number of times (entering numbers between 1 and 74) forcing the Computer to print out the message for various types of ERRORs. Don't waste time trying to understand them now. You can study them in detail in Appendix C.

The only new BASIC word is in Line 30. ERROR has little use in life except as above, printing the Error Code from its code number.

ERROR Trapping

The ON ERROR GOTO statement is of more value. It is used when we think we're on the trail of a specific type of ERROR, but are not sure.

Microsoft ERROR CODES

BASIC ERRORS

Code	Error
1	NEXT without FOR
2	Syntax error
3	RETURN without GOSUB
4	Out of DATA
5	Illegal function call
6	Overflow
7	Out of memory
8	Undefined label
9	Subscript out of range
10	Duplicate Definition
11	Division by zero
12	Illegal direct
13	Type mismatch
14	Out of Heap Space
15	String too long
16	String formula too complex
17	Can't continue
18	Undefined user function
19	No RESUME
20	RESUME without error
21	Unprintable error
22	Missing operand
23	Line too long
26	FOR Without NEXT
29	WHILE without WEND
30	WEND without WHILE
35	Undefined subprogram
36	Subprogram already in use
37	Argument count mismatch
38	Undefined array
	DISK ERRORS
50	FIELD overflow
51	Internal error
52	Bad file number
53	File not found
54	Bad file mode

Code	Error
55	File already open
57	Device I/O error
58	File already exists
61	Disk full
62	Input past end
63	Bad record number
64	Bad file name
66	Direct statement in file
67	Too many opened files
68	Device Unavailable
70	Disk Write Protected
74	Unknown Volume

Suppose we suspect that someplace in the program there is an accidental square rooting of a negative number, and it's goofing up the results. Type in this New test program:

```
10 ON ERROR GOTO 70
20 PRINT
30 INPUT "FIND THE SQUARE ROOT OF";N
40 A = SQR(N)
50 PRINT "SQUARE ROOT OF";N;"=";A
60 GOTO 30
70 BEEP
80 PRINT "SQR ROOT OF NEGATIVE"
90 PRINT "IS ILLEGAL!"
999 END
```

...and Run.

Try positive values, and 0, then try a negative value.

ON ERROR GOTO is acting much as our old friend ON X GOTO did, so there are no big surprises here.

Change Line 10 to a REM Line and try assorted values, ending with a nega-

tive number. Again, no big surprise. An ERROR message was delivered, pinpointing both the nature and location of the ERROR, and execution was terminated. Lines 70, 80 and 90 were *not* executed, however.

Change Line 10 back to:

```
10 ON ERROR GOTO 70
```

and insert:

```
100 RESUME 20
```

...and Run with various values, including negative ones.

Although the Computer was forced to operate with an ERROR (negative square root), execution did not terminate. The ERROR message was delivered, but the Computer kept on going, thanks to RESUME. This is the essence of good ERROR trapping -- identifying the ERROR without "crashing" the program. There may be several interrelated ERRORs that can be found easily only by continuing the Run.

Stop the program, and change Line 100 to:

```
100 RESUME NEXT
```

...and Run.

Although the results are similar to those obtained with:

```
RESUME 20
```

there is a subtle difference.

RESUME NEXT causes execution to RESUME at the NEXT Line immediately following the Line which made the ERROR. Thus Line 50 is PRINTed, even though (in this case) it gives a wrong answer. RESUME 20 directed execution to a very specific Line. With a little head-scratching, we can quickly see how both of these features are useful in difficult debugging situations.

Next, change Line 100 to:

```
100 RESUME
```

...and Run.

As we see and hear, RESUME by itself (or RESUME 0) sends execution back to the Line in which the ERROR is being made. The Computer keeps trying to take the square root of the same negative number. To regain control click Stop from the Run menu or press ⌘. If you are having difficulty visualizing what is taking place in any of these examples, try Stepping (⌘T) through the program, and read the road map.

ERL

Change Line 100 back to:

```
100 RESUME 20
```

and insert:

```
95 PRINT "ERROR IS IN LINE #";ERL
```

...and Run.

The program now informs us that the

```
ERROR IS IN LINE # 40
```

ERL is a "reserved" word that produces the *Line number* in which the ERROR occurs. For my money, this little jewel in combination with ON ERROR GOTO to snag 'em and RESUME NEXT (or RESUME Line number) to keep the program from crashing, makes this whole hassle worthwhile.

ERR

A final esoteric touch may be obtained by adding the ERR (not ERL) statement. ERR produces the ERROR code number.

We've gone almost full cycle. Insert Line 97:

```
97 PRINT "AND ERROR CODE IS";ERR
```

...and Run.

...which brings us back to Do, a deer, a female deer...(it must be time to STOP this book -- getting too silly!)

EXERCISE 51-1: Enter the following New program:

```
20 CLS
30 FOR I=1 TO 10
40  X = INT(RND * 21) + 1 : F = X-10/X
50  PRINT I,"X =";X,"F(X) = ";F
60  IF F< THEN PRINT
70  IF X = 20 THEN READ A
80 NEXT I
90 INPUT "PRESS RETURN TO CONTINUE";Z :
   GOTO 20
```

Write an ERROR trapping routine that recovers from both ERRORs and PRINTs:

```
OUT OF DATA ERROR IN LINE ##
SYNTAX ERROR IN LINE ##
```
or as appropriate.

HINT -- Syntax error is code 2, and Out of DATA is code 4.

Learned In Chapter 51

Statements	Functions	Miscellaneous
ERROR	ERL	ERROR codes
ON ERROR GOTO	ERR	
RESUME		

SECTION B

ANSWERS TO EXERCISES

SAMPLE ANSWER FOR EXERCISE 5-1:

```
50 PRINT D
```

SAMPLE RUN FOR EXERCISE 5-1:

```
6000
```

Note: You may have used a different Line number in your answer, but the way to get the answer PRINTed on the screen is by using the PRINT statement. If you didn't get it right the first time, don't be discouraged. Type in Line 50 above, and RUN the program. Then return to Chapter 5 and continue.

SAMPLE ANSWER FOR EXERCISE 5-2:

```
10 REM * TIME SOLUTION KNOWING DISTANCE AND RATE *
20 D = 6000
30 R = 500
40 T = D / R
50 PRINT "THE TIME REQUIRED IS";T;"HOURS."
```

Note: Remember to hit Return after each Line.

SAMPLE RUN FOR EXERCISE 5-2:

```
THE TIME REQUIRED IS 12 HOURS.
```

Note: In order to arrive at the formula in Line 40, it is necessary to transpose D = R * T and express the equation in terms of T.

SAMPLE ANSWER FOR EXERCISE 5-3:

```
10 REM * CIRCUMFERENCE SOLUTION *
20 P = 3.14
30 D = 35
40 C = P * D
50 PRINT "THE CIRCLE'S CIRCUMFERENCE IS";C;"FEET."
```

SAMPLE RUN FOR EXERCISE 5-3:

```
THE CIRCLE'S CIRCUMFERENCE IS 109.9 FEET.
```

Note: Since pi is not included in Microsoft BASIC, we have to set a variable (in this case P was used) equal to the value pi (3.14).

SAMPLE ANSWER FOR EXERCISE 5-4:

```
10 REM * CIRCULAR AREA SOLUTION *
20 P = 3.14
30 R = 5
40 A = P * R * R
50 PRINT "THE CIRCLE'S AREA IS";A;"SQUARE INCHES."
```

SAMPLE RUN FOR EXERCISE 5-4:

```
THE CIRCLE'S AREA IS 78.5 SQUARE INCHES.
```

Note: Some BASICs do not have a function which means "raise to the power" to handle R^2. (Microsoft BASIC does.) In easy cases like this one, we can simply use R times R (R*R). You'll learn how to use the simple EXPONENTIATION function as we proceed.

SAMPLE ANSWER FOR EXERCISE 5-5:

```
10 B = 225
20 C = 17 + 35 + 225
30 D = 40 + 200
40 N = B - C + D
50 PRINT "YOUR NEW BALANCE IS $";N
```

SAMPLE RUN FOR EXERCISE 5-5:

```
YOUR NEW BALANCE IS $ 188
```

SAMPLE ANSWER FOR EXERCISE 6-1:

```
10 REM * CAR MILES SOLUTION PROGRAM *
20 N = 100,000,000
30 D = 10000000
40 T = N * D
50 PRINT "THE TOTAL NUMBER OF MILES DRIVEN IS";T
```

SAMPLE RUN FOR EXERCISE 6-1:

```
THE TOTAL NUMBER OF MILES DRIVEN IS 1E+15
```

Note: As discussed earlier, the answer is the number 1 followed by 15 zeros, or 1,000,000,000,000,000. That's one zillion. The Computer will not store any number larger than 9,999,999 without converting it to exponential notation.

SAMPLE ANSWER FOR EXERCISE 6-2:

```
20 N = 1E+08
30 D = 1E+07
```

SAMPLE RUN FOR EXERCISE 6-2:

```
THE TOTAL NUMBER OF MILES DRIVEN IS 1E+15
```

SAMPLE ANSWER FOR EXERCISE 7-1:

```
10 REM * FAHRENHEIT TO CELSIUS CONVERSION *
20 F = 65
30 C = (F-32) * (5/9)
40 PRINT F;"DEGREES FAHRENHEIT =";C;"DEGREES CELSIUS."
```

SAMPLE RUN FOR EXERCISE 7-1:

```
65 DEGREES FAHRENHEIT = 18.33333 DEGREES CELSIUS.
```

Observe carefully how the parentheses were placed. As a general rule, when in doubt -- use parentheses. The worst they can do is slow down calculating the answer by a few millionths of a second.

SAMPLE ANSWER FOR EXERCISE 7-2:

```
30 C = F - 32 * (5 / 9)
```

SAMPLE RUN FOR EXERCISE 7-2:

```
65 DEGREES FAHRENHEIT = 47.22222 DEGREES CELSIUS.
```

Note how silently and dutifully the Computer came up with the wrong answer. It has done as we directed, and we directed it wrong. A common phrase in computer circles is GIGO (pronounced "gee-goe"). It stands for "Garbage In - Garbage Out." We have given the Computer garbage, and it gave it back to us by way of a wrong answer. Phrased another way, "Never in the history of mankind has there been a machine capable of making so many mistakes so rapidly and confidently." A computer is worthless unless it is programmed correctly.

SAMPLE ANSWER FOR EXERCISE 7-3:

```
30 C = (F - 32) * 5 / 9
```

SAMPLE RUN FOR EXERCISE 7-3:

```
65 DEGREES FAHRENHEIT = 18.33333 DEGREES CELSIUS.
```

SAMPLE ANSWER FOR EXERCISE 7-4:

Two possible answers:

```
30 - (9 - 8) - (7 - 6) = 28
30 - (9 - (8 - (7 - 6))) = 28
```

Sample programs:

```
10 A = 30 - (9 - (8 - (7 - 6)))
20 PRINT A
```

Or Line 10 might be:

```
10 A = 30 - (9 - 8) - (7 - 6)
```

Try a few on your own.

SAMPLE ANSWER FOR EXERCISE 8-1:

```
10 A = 5
20 IF A <> 5 THEN 50
30 PRINT "A EQUALS 5."
40 END
50 PRINT "A DOES NOT EQUAL 5."
```

SAMPLE RUN FOR EXERCISE 8-1:

```
A EQUALS 5.
```

SAMPLE ANSWER FOR EXERCISE 8-2:

```
10 A = 6
20 IF A <> 5 THEN 50
30 PRINT "A EQUALS 5."
40 END
50 PRINT "A DOES NOT EQUAL 5."
60 IF A < 5 THEN 90
70 PRINT "A IS LARGER THAN 5."
80 END
90 PRINT "A IS SMALLER THAN 5."
```

SAMPLE RUN FOR EXERCISE 8-2:

```
A DOES NOT EQUAL 5.
A IS LARGER THAN 5.
```

Note: We had to put in another END statement (Line 80) to keep the program from running in to Line 90 after PRINTing Line 70.

SAMPLE ANSWER FOR EXERCISE 13-1:

```
1 REM * DELAY PROGRAM *
2 INPUT "HOW MANY SECONDS DELAY DO YOU WISH";S
3 P = 2200
4 D = S * P
5 FOR X = 1 TO D
6 NEXT X
7 PRINT "DELAY IS OVER.  TOOK";S;"SECONDS."
```

Explanation:

Line 2 used the INPUT statement to obtain desired delay, S in seconds.

Line 3 defined P, the number of passes required to for a one second delay.

Line 4 multiplied the delay for one second times the number of seconds desired and called that product D.

Line 5 began the FOR-NEXT loop from 1 to whatever is required.

Line 6 is the other half of the loop.

Line 7 reports the delay is over and prints S, the number of seconds. Obviously, S is only as accurate as the program itself since it merely copies the value of S you entered in Line 2.

SAMPLE ANSWER FOR EXERCISE 13-2:

```
50 PRINT "RATE", "TIME"
55 PRINT "(MPH)","(HOURS)"
```

If you honestly had trouble with this one, better go back and start all over because you've missed the real basics.

SAMPLE ANSWER FOR EXERCISE 13-3:

```
10 PRINT "***    S A L A R Y    R A T E
   C H A R T    ***"
20 PRINT
30 PRINT "YEAR","MONTH","WEEK","DAY"
40 PRINT
50 FOR Y = 5000 TO 20000 STEP 1000
55  REM * CONVERT YEARLY INCOME INTO MONTHLY *
60  M = Y/12
65 REM * CONVERT YEARLY INCOME INTO WEEKLY *
70 W = Y/52
75 REM * CONVERT WEEKLY INTO DAILY *
80 D = W/5
100  PRINT Y,M,W,D
110 NEXT Y
```

SAMPLE RUN FOR EXERCISE 13-3:

```
***    S A L A R Y    R A T E    CHART    ***

YEAR              MONTH        WEEK          DAY

5000              416.6667     96.15385      19.23077
6000              500          115.3846      23.07682
7000              583.3333     134.6154      26.92308
8000              666.6667     153.8462      30.76923
```

etc.

SAMPLE ANSWER FOR EXERCISE 13-4:

```
10 R = .01
20 D = 1
30 T = .01
40 PRINT "DAY","DAILY","TOTAL"
50 PRINT " # ","RATE","EARNED"
60 PRINT
70 PRINT D,R,T
80 IF R > 1000000 THEN END
90 R = R * 2
100 D = D + 1
110 T = T + R
120 GOTO 70
```

SAMPLE RUN FOR EXERCISE 13-4:

```
DAY             DAILY          TOTAL
  #             RATE           EARNED

 1               .01            .01
 2               .02            .03
 3               .04            .07
 4               .08            .15
 5               .16            .31
 6               .32            .63
```

etc.

SAMPLE ANSWER FOR EXERCISE 13-5:

```
10 PRINT "WIRE FENCE","LENGTH","WIDTH","AREA"
20 PRINT " (FEET)","(FEET)","(FEET)","(SQ. FEET)"
30 F = 1000
40 FOR L = 0 TO 500 STEP 50
50  W = (F - 2 * L)/2
60  A = L * W
70  PRINT F,L,W,A
80 NEXT L
```

SAMPLE RUN FOR EXERCISE 13-5:

```
WIRE FENCE    LENGTH        WIDTH          AREA
(FEET)        (FEET)        (FEET)         (SQ. FEET)
1000          0             500            0
1000          50            450            22500
1000          100           400            40000
1000          150           350            52500
1000          200           300            60000
```

etc.

ADDENDUM TO EXERCISE 13-5:

Here's a program that lets the Computer do the comparing:

```
5 REM * SET MAXIMUM AREA AT ZERO *
10 M = 0
15 REM * SET DESIRED LENGTH AT ZERO *
20 N = 0
25 REM * F IS TOTAL FEET OF FENCE AVAILABLE *
30 F = 1000
35 REM * L IS LENGTH OF ONE SIDE OF RECTANGLE *
40 FOR L = 0 TO 500 STEP 50
45  REM * W IS WIDTH OF ONE SIDE OF RECTANGLE *
50  W = (F - 2 * L) / 2
60  A = W * L
65  REM * COMPARE WITH A CURRENT MAXIMUM,
    REPLACE IF NECESSARY *
70  IF A <= M THEN GOTO 110
```

```
80  M = A
90  REM * ALSO UPDATE CURRENT DESIRED LENGTH *
100  N = L
110 NEXT L
120 PRINT "FOR LARGEST AREA USE THESE DIMENSIONS:"
130 PRINT N;"FT. BY";500-N;"FT. FOR TOTAL AREA OF";M;
    "SQ. FT."
```

SAMPLE ANSWER FOR OPTIONAL EXERCISE 13-6:

```
10 REM * FINDS OPTIMUM LOAD TO SOURCE MATCH *
20 PRINT "LOAD","CIRCUIT","SOURCE","LOAD"
30 PRINT "RESISTANCE","POWER","POWER","POWER"
40 PRINT "(OHMS)","(WATTS)","(WATTS)","(WATTS)"
50 PRINT
60 FOR R = 1 TO 20
70  I = 120 / (10 + R)
80  C = I * I * (10 + R)
90  S = I * I * 10
100  L = I * I * R
110  PRINT R,C,S,L
120 NEXT R
```

SAMPLE RUN FOR OPTIONAL EXERCISE 13-6:

```
LOAD            CIRCUIT         SOURCE          LOAD
RESISTANCE      POWER           POWER           POWER
(OHMS)          (WATTS)         (WATTS)         (WATTS)

 1               1309.091        1190.083        119.0083
 2               1200            1000            200
 3               1107.692        852.071         255.6213
 4               1028.571        734.6938        293.8775
 5               960             640             320
 6               900             562.5           337.5
 7               847.0588        498.2699        348.7889
 8               800             444.4444        355.5555
 9               757.8948        398.892         359.0028
 10              720             360             360
 11              685.7143        326.5306        359.1837
```

etc.

SAMPLE ANSWER FOR EXERCISE 14-1:

```
10 PRINT "THE                TOTAL          SPENT"
20 PRINT "BUDGET","YEAR'S","THIS"
30 PRINT "CATEGORY";TAB(15);"BUDGET";TAB(29);"MONTH"
```

SAMPLE ANSWER FOR EXERCISE 14-2:

```
10 PRINT "  ***   S A L A R Y   R A T E
   C H A R T   ***"
20 PRINT
30 PRINT " YEAR";TAB(15);"MONTH";TAB(28);"WEEK"
40 PRINT TAB(41);"DAY";TAB(54);"HOUR"
50 FOR Y = 5000 TO 20000 STEP 1000
55  REM * CONVERT YEARLY INCOME INTO MONTHLY *
60 M = Y/12
65 REM * CONVERT YEARLY INCOME INTO WEEKLY *
70 W = Y/52
75 REM * CONVERT WEEKLY INCOME INTO DAILY *
80 D = W/5
85 REM * CONVERT WEEKLY INCOME INTO HOURLY *
90 H = W/40
100 PRINT Y;TAB(14);M;TAB(27);W;TAB(40);D;TAB(53);H
110 NEXT Y
```

SAMPLE RUN FOR EXERCISE 14-2:

```
***   S A L A R Y   R A T E   C H A R T   ***

YEAR      MONTH        WEEK          DAY          HOUR
5000      416.6667     96.15385      19.23077     2.403846
6000      500          115.3846      23.07692     2.884615
7000      583.3333     134.6154      26.92308     3.365385
8000      666.6667     153.8462      30.76923     3.846154
9000      750          173.0769      34.61538     4.326923
10000     833.3333     192.3077      38.46154     4.807693
11000     916.6667     211.5385      42.30769     5.288462
12000     1000         230.7692      46.15385     5.769231
13000     1083.333     250           50           6.25
14000     1166.667     269.2308      53.84615     6.730769
15000     1250         288.4615      57.69231     7.211539
```

etc.

SAMPLE ANSWER FOR EXERCISE 15-1:

```
10 FOR A = 1 TO 3
20  PRINT "A LOOP"
30   FOR B = 1 TO 2
40    PRINT ,"B LOOP"
42     FOR C = 1 TO 4
44      PRINT ,,"C LOOP"
48     NEXT C
50   NEXT B
60 NEXT A
```

SAMPLE ANSWER FOR EXERCISE 15-2:

The program will be the same as the answer to Exercise 15-1 with the following additions:

```
45          FOR D = 1 TO 5
46           PRINT ,,,"D LOOP"
47          NEXT D
```

Note: To get the full impact of this "4-deep" nesting, stop the RUN frequently to examine the nesting relationships between each of the loops.

SAMPLE ANSWER FOR EXERCISE 16-1:

Addition of the following single Line gives a nice clean PRINTout with all the values "rounded" to their integer value:

```
55 A = INT(A)
```

Worth all the effort to learn it, wasn't it?

SAMPLE ANSWER FOR EXERCISE 16-2:

```
55 A = INT(10 * A ) / 10
```

When 3.14159 was multiplied times 10 it became 31.4159. The INTeger value of 31.4159 is 31. 31 divided by 10 is 3.1, etc.

SAMPLE ANSWER FOR EXERCISE 16-3:

This was almost too easy.

```
55 A = INT(100 * A) / 100
```

SAMPLE ANSWER FOR EXERCISE 17-1:

```
10 INPUT "TYPE ANY NUMBER";X
20 T = SGN(X)
30 ON T+2 GOTO 50,70,90
40 END
50 PRINT "THE NUMBER IS NEGATIVE."
60 END
70 PRINT "THE NUMBER IS ZERO."
80 END
90 PRINT "THE NUMBER IS POSITIVE."
```

SAMPLE ANSWER FOR EXERCISE 18-1:

```
10 RANDOMIZE TIMER
20 INPUT "PRESS <Return> TO CONTINUE";A
30 CLS
40 GOSUB TOSS
50 P = N
60 PRINT "YOU ROLLED ";P
70 ON P GOTO 80,140,140,90,90,90,110,90,90,90,110,140
80 REM USED FOR THE ON STATEMENT IF P = 1 (WHICH IT
CAN'T)
90 PRINT "YOUR POINT IS ";N
100 GOTO 170
110 PRINT "YOU WIN!"
120 PRINT
130 GOTO 20
140 PRINT "YOU LOSE."
150 PRINT
```

```
160 GOTO 20
170 GOSUB TOSS
180 M = N
190 PRINT
200 PRINT "YOU ROLLED ";M
210 IF P=M THEN 110
220 IF M=7 THEN 140
230 GOTO 170
TOSS:
 A = INT(RND*6+1)
 B = INT(RND*6+1)
 N = A + B
 RETURN
```

SAMPLE ANSWER FOR EXERCISE 21-1:

```
10 PRINT CHR$(77);CHR$(65);CHR$(67);
20 PRINT CHR$(73);CHR$(78);CHR$(84);
30 PRINT CHR$(79);CHR$(83);CHR$(72)
```

SAMPLE ANSWER FOR EXERCISE 21-2:

```
10 INPUT "ENTER A NUMBER";A$
20 A = ASC(A$)
30 IF A<47 THEN 10
40 IF A>57 THEN 10
50 PRINT "ASCII VALUE OF ";A$;" IS";A
```

SAMPLE ANSWER FOR EXERCISE 22-1:

```
10 INPUT "FIRST STRING";A$
20 INPUT "SECOND STRING";B$
30 PRINT : PRINT "ALPHABETICAL ORDER:"
40 IF A$<B$ THEN PRINT A$,B$ : END
50 PRINT B$,A$
```

SAMPLE ANSWER FOR EXERCISE 23-1:

```
10 INPUT "INPUT STRING";A$
20 IF LEN(A$)>10 THEN PRINT "THE 10 CHARACTER
   LIMIT WAS EXCEEDED."
```

SAMPLE ANSWER FOR EXERCISE 23-2:

```
10 INPUT "ENTER PASSWORD";A$
20 FOR X=1 TO 11
30  READ N
40  P$ = P$ + CHR$(N)
50 NEXT X
60 IF A$ = P$ THEN 90
70 PRINT "WRONG PASSWORD. TRY AGAIN."
80 END
90 PRINT "CORRECT PASSWORD; YOU MAY ENTER."
100 DATA 79,80,69,78,32,83,69,83,65,77,69
```

SAMPLE ANSWER FOR EXERCISE 24-1:

```
10 INPUT "INPUT YOUR STREET ADDRESS";A$
20 A = VAL(A$)
30 PRINT: PRINT "YOUR NEIGHBOR'S STREET NUMBER IS ";
   A+4
```

SAMPLE ANSWER FOR EXERCISE 24-2:

```
10 WIDTH 60
20 FOR X = 101 TO 120
30  A$ = STR$(X)
40  PRINT A$+"WT",
50 NEXT X
```

SAMPLE RUN FOR EXERCISE 24-2:

```
101WT      102WT      103WT      104WT
105WT      106WT      107WT      108WT
109WT      110WT      111WT      112WT
113WT      114WT      115WT      116WT
117WT      118WT      119WT      120WT
```

SAMPLE ANSWER FOR EXERCISE 25-1:

```
10 INPUT "ISN'T THIS A SMART COMPUTER";A$
20 B$ = LEFT$(A$,1)
30 IF B$ = "Y" THEN PRINT "AFFIRMATIVE":END
40 IF B$ = "N" THEN PRINT "NEGATIVE":END
50 PRINT "THIS IS A YES OR NO QUESTION"
60 GOTO 10
```

SAMPLE ANSWER FOR EXERCISE 25-2:

```
10 MAX$ = ""
20 FOR I = 1 TO 3
30  READ A$
40  N$ = MID$(A$,2,3)
50  IF N$>MAX$ THEN MAX$ = N$: P$ = A$
60 NEXT I
70 PRINT "THE PART NUMBER WITH THE LARGEST NUMERIC
   PORTION IS ";P$
80 DATA N106WT,A208FM,Z154DX
```

SAMPLE ANSWER FOR EXERCISE 25-3:

Choice C. P-

SAMPLE ANSWER FOR EXERCISE 25-4:

```
10 PRINT STRING$(30,42)
```

SAMPLE ANSWER FOR EXERCISE 26-1:

```
10 CLS
20 PRINT TAB(15);"DATE: ";DATE$,"TIME: ";TIME$;
30 FOR X = 1 TO 1000 : NEXT
40 GOTO 10
```

SAMPLE ANSWER FOR EXERCISE 28-1:

```
10 A = 5 : B = 12
20 C = SQR(A^2 + B^2)
30 PRINT "THE SQUARE ROOT OF";A;
40 PRINT "SQUARED PLUS";B;"SQUARED IS";C
```

SAMPLE ANSWER FOR EXERÇISE 28-2:

```
10 INPUT "ENTER A NUMBER";N
20 PRINT "LOG (EXP (";N;") ) =";LOG(EXP(N))
30 PRINT "EXP (LOG (";N;") ) =";EXP(LOG(N))
40 PRINT : GOTO 10
```

SAMPLE ANSWER FOR EXERCISE 31-1:

```
10 INPUT "STARTING HORIZONTAL PIXEL (0 TO 490)";H
20 INPUT "ENDING HORIZONTAL PIXEL (0 TO 490)";I
30 INPUT "STARTING VERTICAL PIXEL (0 TO 300)";V
40 INPUT "ENDING VERTICAL PIXEL (0 TO 300)";W
50 CLS
60 FOR X = H TO I
70  FOR Y = V TO W
80   PSET(X,Y)
90  NEXT Y
100 NEXT X
999 GOTO 999
```

SAMPLE ANSWER FOR EXERCISE 32-1:

A. MOVE THE DOT UP

```
10 INPUT "HORIZONTAL STARTING ADDRESS (0 TO 490)";H
20 INPUT "VERTICAL STARTING ADDRESS (0 TO 300)";V
30 CLS
40 PRESET(H,V+1)
50 PSET(H,V)
60 V = V - 1
70 IF V > -1 GOTO 40
99 GOTO 99
```

B. MOVE THE DOT TO THE LEFT

```
10 INPUT "HORIZONTAL STARTING ADDRESS (1 TO 490)";H
20 INPUT "VERTICAL STARTING ADDRESS (0 TO 300)";V
30 CLS
40 PRESET(H+1,V)
50 PSET(H,V)
60 H = H - 1
70 IF H > -1 GOTO 40
99 GOTO 99
```

SAMPLE ANSWER FOR EXERCISE 32-2:

```
50 LINE (75,80)-(75,100)
60 LINE -(125,100)
70 LINE -(125,80)
```

SAMPLE ANSWER FOR EXERCISE 33-1:

Insert the following Lines:

```
85 P$ = "PING!"
95 IF V = 250 GOTO 150
105 IF V = 50 GOTO 160
135 P$ = "      " : BEEP
137 ON D+2 GOTO 150,,160
150 LOCATE 15,26 : PRINT P$; : GOTO 80
160 LOCATE 5,26 : PRINT P$ : G TO 80
```

5 spaces

Note that P$ both prints the "PING" and makes it disappear by printing blanks in its place. Also, Line 135 introduces the BEEP function which will have many similar applications later in the book.

SAMPLE ANSWER FOR EXERCISE 36-1:

```
40 U$ = "####.##  5  "
40 U$ = "$####.##  5  "
40 U$ = "$$###.##  5  "
40 U$ = "$$,###.##  5  "
40 U$ = "**$,###.##  5  "
```

SAMPLE ANSWER FOR EXERCISE 36-2:

```
10 PRINT TAB(20)"CREDITS   6   TAX     9     TOTAL"
20 FOR I = 1 TO 3
30  READ A$,X,Y,Z
40  U$ = "\        17         \    6    ##.##   11  .##  8
     ##.##"
50  PRINT USING U$; A$,X,Y,Z
60 NEXT I
70 READ A$,N
80 V$ = "  3 \2 \  6   ###.##"
90 PRINT TAB(29); : PRINT USING V$;A$,N
100 DATA ASTRAL COMPUTER, 18.3, .7, 19.0
110 DATA BIOFEEDBACK ADAPTER, 1.8, 0, 1.8
120 DATA PERSONALITY MODULE, 7.2, .3, 7.5
130 DATA "DUE:", 28.3
```

NOTE: Decimal points will not line up due to proportional spacing.

SAMPLE ANSWER FOR EXERCISE 37-1:

```
10 A$ = "REVENUES" : B$ = "EXPENSES" : C$ = "ASSETS"
20 U$ = " 2\   7    \       14       \   7   \  5  \ 8
        \"
30 PRINT USING U$; A$,B$,C$
40 A# = 1203104.22 : B# = 560143.8 : C = 0
50 V$ = "######,###.##  5  ####,###.##  5
     #####,###.##-"
60 PRINT USING V$; A#,C,A#
70 PRINT USING V$; C,B#,-B#
```

NOTE: Remember, decimal points will not line up due to proportional spacing.

SAMPLE ANSWER TO EXERCISE 39-1:
Add or change the following Lines:

```
5 DIM A(210)
10 INPUT "WHICH CAR TO EXAMINE ";W
20 FOR L = 1 TO 10
30  READ A(L)
40 NEXT L
50 FOR S = 101 TO 110
60  READ A(S)
70 NEXT S
72 FOR B = 201 TO 210
74  READ A(B)
76 NEXT B
80 PRINT
90 PRINT "CAR#","ENG. SIZE","COLOR","BODY STYLE"
100 PRINT W,A(W),A(W+100),A(W+200)
500 DATA 300,200,500,300,200
510 DATA 300,400,400,300,500
520 DATA 3,1,4,3,2,4,3,2,1,3
530 DATA 20,20,10,20,30,20,30,10,20,20
```

SAMPLE ANSWER FOR EXERCISE 39-2:

Delete Lines 500 through 540 and change Line 30 to:

```
30 FOR C=1 TO 52 : A(C)=C : NEXT C
```

SAMPLE ANSWER FOR EXERCISE 40-1:

Change Line 50 to:

```
50 IF A$(F) >= A$(S) THEN 90 'TEST FOR LARGER ASCII #
```

An approach is to reverse the order of printing:

```
120 FOR D=N TO 1 STEP-1 : PRINT A$(D), : NEXT D
```

but that's not what we had in mind.

SAMPLE ANSWER FOR EXERCISE 41-1:

```
10 FOR E=1 TO 4
20  FOR D=1 TO 3
30   REM ENTRY DATA: NAME, NUMBER, $$$$
40   READ R$(E,D)
50   PRINT R$(E,D),
60  NEXT D : PRINT
70 NEXT E : PRINT
1000 REM * DATA FILE *
1010 DATA "JONES, C.", 10439, 100.00
1020 DATA "ROTH, J.", 10023, 87.24
1030 DATA "BAKER, H.", 12936, 398.34
1040 DATA "HARMON, D.", 10422, 23.17
```

SAMPLE ANSWER FOR EXERCISE 41-2:

Insert:

```
100 REM *** SORT ***
110 FOR F=1 TO 3
120  FOR S=F+1 TO 4
130   IF R$(F,1) <= R$(S,1) THEN 190
140    FOR J=1 TO 3
150     T$ = R$(F,J)
160     R$(F,J) = R$(S,J)
170     R$(S,J) = T$
180    NEXT J
190  NEXT S
```

```
200 NEXT F
210 PRINT : PRINT "ALPHA SORT" : PRINT
220 FOR E=1 TO 4
230   FOR D=1 TO 3
240    PRINT R$(E,D)
250   NEXT D : PRINT
260 NEXT E : PRINT
```

SAMPLE ANSWER FOR EXERCISE 41-3:

Change these Lines:

```
130 IF VAL(R$(F,3)) <= VAL(R$(S,3)) THEN 190
210 PRINT : PRINT "NUMERIC SORT": PRINT
```

SAMPLE ANSWER FOR EXERCISE 43-1:

```
10 INPUT "IS GATE 'X' OPEN";A$
20 INPUT "IS GATE 'Y' OPEN";B$
30 INPUT "IS GATE 'Z' OPEN";C$
40 PRINT
50 IF A$="Y" OR B$="Y" OR C$="Y" THEN 80
60 PRINT "OLD BESSIE IS SECURE IN PASTURE #3."
70 END
80 PRINT "A GATE IS OPEN."
90 PRINT " OLD BESSIE IS FREE TO ROAM."
```

SAMPLE ANSWER FOR EXERCISE 46-1:

```
10 OPEN "SHOPPING" FOR OUTPUT AS 1
20 FOR X = 1 TO 5
30  PRINT "ENTER ITEM #";X;
40  INPUT A$
50  PRINT #1,A$
60 NEXT X
70 CLOSE
```

SAMPLE ANSWER FOR EXERCISE 46-2:

```
80 OPEN "SHOPPING" FOR INPUT AS 1
90 FOR X = 1 TO 5
100  INPUT #1,A$
110  PRINT "ITEM #";X;"IS ";A$
120 NEXT X
130 CLOSE
```

SAMPLE ANSWER FOR EXERCISE 46-3:

```
10 OPEN "NAMEAGE" FOR OUTPUT AS 1
20 INPUT "ENTER A NAME OF 'DONE' TO END";N$
30 IF N$ = "DONE" THEN 80
40 PRINT #1,N$
50 INPUT "HOW OLD IS HE/SHE";A
60 PRINT #1,A
70 GOTO 20
80 CLOSE 1
90 OPEN "NAMEAGE" FOR INPUT AS 1
100 IF EOF(1) THEN 140
110 INPUT #1,N$,A
120 PRINT N$," IS";A;"YEARS OLD"
130 GOTO 100
140 CLOSE
```

SAMPLE ANSWER FOR EXERCISE 48-1:

```
10 REM * TEST GRADER *
20 PRINT "THIS IS A TEST GRADING PROGRAM"
30 PRINT "ENTER THE STUDENT'S FIVE ANSWERS AS REQUESTED"
40 RESTORE
50 N = 0
60 FOR C=1 TO 5
70  PRINT "ANSWER NUMBER";C;
80  INPUT A
90  READ B
100  PRINT A,B,
110  IF A=B THEN PRINT "CORRECT": N=N+1 ELSE PRINT ,
     "WRONG"
120  PRINT
130 NEXT C
```

```
140 PRINT N;"RIGHT OUT OF 5 WHICH IS";
150 PRINT N/5 * 100;"%"
160 PRINT "ANY MORE TESTS TO GRADE";
170 INPUT "--1=YES, 2=NO";Z
180 IF Z=1 THEN CLS: GOTO 40
190 DATA 65,23,17,56,39
```

SAMPLE ANSWER FOR EXERCISE 48-2:

```
100 CLS
110 PRINT : PRINT
120 PRINT "SELECT ONE OF THE FOLLOWING INVESTMENTS"
130 PRINT
140 PRINT "   1 - CERTIFICATE OF DEPOSIT"
150 PRINT "   2 - BANK SAVINGS ACCOUNT"
160 PRINT "   3 - CREDIT UNION"
170 PRINT "   4 - MORTGAGE LOAN"
180 PRINT : INPUT "INVESTMENT (1-4):";F
190 ON F GOTO DEPOSITS,SAVINGS,C.U.,MORTGAGE
200 GOTO 100 : REM * IF NUMBER NOT BETWEEN 1 AND 4 *
DEPOSITS:
  REM * CERTIFICATE OF DEPOSIT PROGRAM GOES HERE *
  PRINT "THE C.D. PROGRAM HAS YET TO BE WRITTEN."
  GOSUB DELAY : GOTO 100
SAVINGS:
  REM * BANK SAVINGS ACCOUNT PROGRAM *
  CLS : PRINT : PRINT "THE ROUTINE CALCULATES SIMPLE
  INTEREST ON"
  PRINT "DOLLARS HELD IN DEPOSIT FOR SPECIFIED PERIOD"
  PRINT "USING A SPECIFIED PERCENTAGE OF INTEREST." :
  PRINT
  PRINT : INPUT "HOW LARGE IS THE DEPOSIT (IN
  DOLLARS)";P
  INPUT "HOW LONG WILL YOU LEAVE IT IN (IN DAYS)";D
  INPUT "WHAT INTEREST RATE DO YOU EXPECT (IN %)";R
  CLS : PRINT : PRINT
  PRINT "A STARTING PRINCIPAL OF $";P;"AT A RATE OF"
  REM INTEREST = (% PER YR)/(DAYS PER YR) * DAYS *
  PRINCIPAL
  U$ = "$$#####.##"
  I = R / 100 / 365 * D * P
  PRINT : PRINT TAB(17) : PRINT USING U$;I
  END
```

```
C.U.:
  REM * CREDIT UNION PROGRAM GOES HERE *
  PRINT "THE C.U. PROGRAM HAS YET TO BE WRITTEN."
  GOSUB DELAY : GOTO 100
MORTGAGE:
  REM * MORTGAGE LOAN PROGRAM GOES HERE *
  PRINT "THE M.L. PROGRAM HAS YET TO BE WRITTEN."
  GOSUB DELAY : GOTO 100
DELAY:
  FOR D = 1 TO 5000 : NEXT : RETURN
```

SAMPLE ANSWER FOR EXERCISE 51-1:

```
10 ON ERROR GOTO 100
20 CLS
30 FOR I = 1 TO 10
40  X = INT(RND*21) + 1 : F = X-10/X
50  PRINT I, "X =";X,"F(X) =";F
60  IF F< THEN PRINT
70  IF X = 20 THEN READ A
80 NEXT I
90 INPUT "PRESS ENTER TO CONTINUE";Z : GOTO 20
100 IF ERR=2 THEN 140
110 IF ERR=4 THEN 130
120 PRINT "ERROR" : END
130 PRINT "OUT OF DATA ERROR IN LINE ";ERL: RESUME NEXT
140 PRINT : PRINT "SYNTAX ERROR IN LINE ";ERL:
    RESUME NEXT
```

SECTION C

APPENDICES

Appendix A

ASCII Chart

Dec.	Hex	Geneva	Chicago	Monaco
32	20			
33	21	!	!	!
34	22	"	"	"
35	23	#	#	#
36	24	$	$	$
37	25	%	%	%
38	26	&	&	&
39	27	'	'	'
40	28	(	(	(
41	29	)	)	)
42	2A	*	*	*
43	2B	+	+	+
44	2C	,	,	,
45	2D	-	-	-
46	2E	.	.	.
47	2F	/	/	/
48	30	0	0	0
49	31	1	1	1
50	32	2	2	2
51	33	3	3	3
52	34	4	4	4
53	35	5	5	5

Dec.	Hex	Geneva	Chicago	Monaco
54	36	6	6	6
55	37	7	7	7
56	38	8	8	8
57	39	9	9	9
58	3A	:	:	:
59	3B	;	;	;
60	3C	<	<	<
61	3D	=	=	=
62	3E	>	>	>
63	3F	?	?	?
64	40	@	@	@
65	41	A	A	A
66	42	B	B	B
67	43	C	C	C
68	44	D	D	D
69	45	E	E	E
70	46	F	F	F
71	47	G	G	G
72	48	H	H	H
73	49	I	I	I
74	4A	J	J	J
75	4B	K	K	K

Dec.	Hex	Geneva	Chicago	Monaco
76	4C	L	L	L
77	4D	M	M	M
78	4E	N	N	N
79	4F	O	O	O
80	50	P	P	P
81	51	Q	Q	Q
82	52	R	R	R
83	53	S	S	S
84	54	T	T	T
85	55	U	U	U
86	56	V	V	V
87	57	W	W	W
88	58	X	X	X
89	59	Y	Y	Y
90	5A	Z	Z	Z
91	5B	[	[	[
92	5C	\	\	\
93	5D	]	]	]
94	5E	^	^	^
95	5F	_	_	_
96	60	`	`	`
97	61	a	a	a
98	62	b	b	b
99	63	c	c	c
100	64	d	d	d
101	65	e	e	e
102	66	f	f	f
103	67	g	g	g
104	68	h	h	h
105	69	i	i	i
106	6A	j	j	j
107	6B	k	k	k
108	6C	l	l	l
109	6D	m	m	m

Dec.	Hex	Geneva	Chicago	Monaco
110	6E	n	n	n
111	6F	o	o	o
112	70	p	p	p
113	71	q	q	q
114	72	r	r	r
115	73	s	s	s
116	74	t	t	t
117	75	u	u	u
118	76	v	v	v
119	77	w	w	w
120	78	x	x	x
121	79	y	y	y
122	7A	z	z	z
123	7B	{	{	{
124	7C	|	|	|
125	7D	}	}	}
126	7E	~	~	~
127	7F			
128	80	Ä	Ä	Ä
129	81	Å	Å	Å
130	82	Ç	Ç	Ç
131	83	É	É	É
132	84	Ñ	Ñ	Ñ
133	85	Ö	Ö	Ö
134	86	Ü	Ü	Ü
135	87	á	á	á
136	88	à	à	à
137	89	â	â	â
138	8A	ä	ä	ä
139	8B	ã	ã	ã
140	8C	å	å	å
141	8D	ç	ç	ç
142	8E	é	é	é
143	8F	è	è	è

Dec.	Hex	Geneva	Chicago	Monaco
144	90	ê	ê	ê
145	91	ë	ë	ë
146	92	í	í	í
147	93	ì	ì	ì
148	94	î	î	î
149	95	ï	ï	ï
150	96	ñ	ñ	ñ
151	97	ó	ó	ó
152	98	ò	ò	ò
153	99	ô	ô	ô
154	9A	ö	ö	ö
155	9B	õ	õ	õ
156	9C	ú	ú	ú
157	9D	ù	ù	ù
158	9E	û	û	û
159	9F	ü	ü	ü
160	A0	†	†	†
161	A1	°	°	°
162	A2	¢	¢	¢
163	A3	£	£	£
164	A4	§	§	§
165	A5	•	•	•
166	A6	¶	¶	¶
167	A7	ß	ß	ß
168	A8	®	®	□
169	A9	©	©	□
170	AA	™	™	™
171	AB	´	´	´
172	AC	¨	¨	¨
173	AD	≠	≠	□
174	AE	Æ	Æ	Æ
175	AF	Ø	Ø	Ø
176	B0	∞	∞	∞
177	B1	±	±	±

Dec.	Hex	Geneva	Chicago	Monaco
178	B2	≤	≤	≤
179	B3	≥	≥	≥
180	B4	¥	¥	¥
181	B5	µ	µ	µ
182	B6	∂	∂	∂
183	B7	Σ	Σ	Σ
184	B8	Π	Π	Π
185	B9	π	π	π
186	BA	∫	∫	∫
187	BB	ª	ª	ª
188	BC	º	º	º
189	BD	Ω	Ω	Ω
190	BE	æ	æ	æ
191	BF	ø	ø	ø
192	C0	¿	¿	¿
193	C1	¡	¡	¡
194	C2	¬	¬	¬
195	C3	√	√	√
196	C4	ƒ	ƒ	ƒ
197	C5	≈	≈	≈
198	C6	Δ	Δ	Δ
199	C7	«	«	«
200	C8	»	»	»
201	C9	…	…	…
202	CA			
203	CB	À	À	À
204	CC	Ã	Ã	Ã
205	CD	Õ	Õ	Õ
206	CE	Œ	Œ	Œ
207	CF	œ	œ	œ
208	D0	–	–	–
209	D1	—	—	—
210	D2	“	“	“
211	D3	”	”	”

Dec.	Hex	Geneva	Chicago	Monaco
212	D4	‘	‘	‘
213	D5	’	’	’
214	D6	÷	÷	÷
215	D7	◊	◊	◆
216	D8	ÿ	ÿ	ÿ
217	D9		□	≡

Appendix B

Reserved Words

Using reserved words as variable names will cause a Syntax error.

ABS
ALL
AND
APPEND
AS
ASC
ATN
AUTO
BACKPAT
BASE
BEEP
BREAK
BUTTON
CALL
CDBL
CHAIN
CHR$
CINT
CIRCLE
CLEAR
CLOSE
CLS
COMMON
CONT
COS
CSNG
CSRLIN
CVD
CVDBCD
CVI
CVS
CVSBCD
DATA
DATE$
DEF
DEFDBL
DEFINT
DEFSNG
DEFSTR
DELETE
DIALOG
DIM
EDIT
ELSE
END
EOF
EQV
ERASE
ERASEARC
ERASEOVAL
ERASEPOLY
ERASERECT
ERASEROUNDRECT
ERL
ERR
ERROR
EXIT
EXP
FIELD
FILES
FILLARC
FILLOVAL
FILLPOLY
FILLRECT
FILLROUNDRECT
FIX
FN
FOR
FRAMEARC
FRAMEOVAL
FRAMEPOLY
FRAMERECT
FRAMEROUNDRECT
FRE
GET
GETPEN
GOSUB
GOTO
HEX$
HIDECURSOR
HIDEPEN
IF
IMP
INITCURSOR
INKEY$
INPUT
INSTR
INT
INVERTARC
INVERTOVAL
INVERTPOLY
INVERTRECT
INVERTROUNDRECT
KILL
LBOUND
LCOPY

LEFT$
LEN
LET
LIBRARY
LINE
LINETO
LIST
LLIST
LOAD
LOC
LOCATE
LOF
LOG
LPOS
LPRINT
LSET
MENU
MERGE
MID$
MKD$
MKI$
MKS$
MOD
MOUSE
MOVE
MOVETO
NAME
NEW
NEXT
NOT
OBSCURECURSOR
OCT$
OFF
ON
OPEN
OPTION
OR
OUTPUT
PAINTARC
PAINTOVAL
PAINTPOLY
PAINTRECT
PAINTROUNDRECT
PEEK
PENMODE
PENNORMAL
PENPAT
PENSIZE
PICTURE
POINT
POKE
POS
PRESET
PRINT
PSET
PTAB
PUT
RANDOMIZE
READ
REM
RESET
RESTORE
RESUME
RETURN
RIGHT$
RND
RSET
RUN
SAVE
SCROLL
SETCURSOR
SGN
SHARED
SHOWCURSOR
SHOWPEN
SIN
SPACE$
SPC
SQR
STATIC
STEP
STOP
STR$
STRING$
SUB
SWAP
SYSTEM
TAB
TAN
TEXTFACE
TEXTFONT
TEXTMODE
TEXTSIZE
THEN
TIME$
TIMER
TO
TROFF
TRON
UBOUND
UCASE$
USING
USR
VAL
VARPTR
WAIT
WAVE
WEND
WHILE
WIDTH
WINDOW
WRITE
XOR

Appendix C

Error Messages

Code	Error
1	NEXT without FOR
2	Syntax error
3	RETURN without GOSUB
4	Out of DATA
5	Illegal function call
6	Overflow
7	Out of memory
8	Undefined label
9	Subscript out of range
10	Duplicate Definition
11	Division by zero
12	Illegal direct
13	Type mismatch
14	Out of heap space
15	String too long
16	String formula too complex
17	Can't continue
18	Undefined user function
19	No RESUME
20	RESUME without error
21	Unprintable error
22	Missing operand
23	Line too long
26	FOR without NEXT
29	WHILE without WEND
30	WEND without WHILE
35	Undefined subprogram
36	Subprogram already in use
37	Argument count mismatch
38	Undefined array

Code	Error
50	FIELD overflow
51	Internal error
52	Bad file number
53	File not found
54	Bad file mode
55	File already open
57	Device I/O Error
58	File already exists
61	Disk full
62	Input past end
63	Bad record number
64	Bad file name
66	Direct statement in file
67	Too many files
68	Device unavailable
70	Disk write protected
74	Unknown volume

Argument count mismatch (Code 37): the arguments in a subprogram CALL statement do not equal the number of arguments in its corresponding SUB statement.

Bad file mode (Code 54): statements PUT or GET were used with a sequential file or a closed file to MERGE a non-ASCII file or to execute an OPEN with a file mode other than input, output, append, or random.

Bad file name (Code 64): An invalid form is used for the filename with BLOAD, BSAVE, KILL, OPEN, NAME, or FILES (e.g., a filename starting with a period).

Bad file number (Code 52): a statement references a file with a file number that isn't OPEN or is out of the range of possible file numbers which was specified at initialization; the device name in the file specification is too long or invalid, or the filename was too long or invalid.

Bad record number (Code 63): the record number in a PUT or GET statement is either greater than the maximum allowed (32767) or equal to zero.

Can't continue (Code 17): the command CONT is typed, but there is no program to continue, the program has just been modified, or the program was stopped due to an error.

Device I/O Error (Code 57): an error occurred on a device I/O operation. DOS can't recover from the error.

Device unavailable (Code 68): Reference has been made to a device that is not connected (i.e. external drive).

Direct statement in file (Code 66): a direct statement is encountered while LOADing or CHAINing to an ASCII format file. The LOAD or CHAIN is terminated.

Disk full (Code 61): there is no more storage space on diskette. When this error occurs, files will be closed.

Disk write protected (Code 70): an attempt was made to save data to a write protected disk.

Division by zero (Code 11): the Computer is asked to divide a number by 0.

Duplicate Definition (Code 10): the Computer is told to DIMension a numeric or string Matrix after it has already been DIMensioned earlier in the same program.

FIELD overflow (Code 50): a FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file in the OPEN statement, or the end of the FIELD buffer was encountered while doing sequential I/O (PRINT#,WRITE#,INPUT#,etc.) to a random file.

File already exists (Code 58): a NAME statement has specified a filename that is identical to a filename already used on the diskette.

File already open (Code 55): you tried to OPEN a file for sequential output or append, but the file is already OPEN, or you tried to KILL a file that is open.

File not found (Code 53): a statement such as LOAD, KILL, NAME, FILES, or OPEN has referenced a file that does not exist on the specified drive.

FOR without NEXT (Code 26): an attempt is made to RUN a program containing a FOR-NEXT loop, but the word "NEXT" is missing.

Illegal direct (Code 12): the Computer is asked to INPUT a value or string in the Immediate or Direct mode.

Illegal function call (Code 5): illegal values are used with the built-in math functions, or the Computer cannot figure out what to compute because of the values it received.

Input past end (Code 62): this is an end of file error. An input statement was executed for a null (empty) file or after all the data in a sequential file was already input. To avoid this error, use the EOF function to detect the end of file. This error also occurs if you try to read from a file that was opened for output or append.

Internal error (Code 51): there has been an internal malfunction in BASIC. The conditions under which the message appeared need to be reported to Microsoft.

Line too long (Code 23): a line has been entered that has too many characters.

Missing operand (Code 22): the Computer is not given all the information required to carry out its directive.

NEXT without FOR (Code 1): an attempt is made to RUN a program containing a FOR-NEXT loop, but the word "FOR" is missing.

No RESUME (Code 19): an ON ERROR GOTO statement is used to branch to a specified program line, and the Computer does not encounter a RESUME statement before the program stops.

Out of DATA (Code 4): the Computer is told to READ more items from the DATA statement than are available.

Out of heap space (Code 14): the Macintosh heap is out of memory which may result in the inability to use the desktop accessories.

Out of memory (Code 7): an attempt is made to store a program larger than the Computer's memory storage space. Also occurs when a matrix variable is assigned more elements than there is space in memory to store it.

Overflow (Code 6): the Computer is unable to use a number because it is either too large or too small. An overflow condition can also be created by routine mathematical calculations at either the statement or command levels.

RESUME without error (Code 20): the Computer encounters a RESUME statement without first finding an ON ERROR GOTO statement.

RETURN without GOSUB (Code 3): the Computer reads a RETURN statement but there is no corresponding GOSUB.

String formula too complex (Code 16): string manipulation has become too complicated or too long for the Computer.

String too long (Code 15): an attempt is made to store more than 32,767 characters in a string variable.

Subprogram already in use (Code 36): the subprogram that was called has already been called and has not yet been ended or exited.

Subscript out of range (Code 9): the elements in a numeric or string matrix are beyond the range of values reserved in the DIM statement.

Syntax error (Code 2): a command, statement or function is misspelled, or an operator is omitted.

Too many files (Code 67): SAVE or OPEN is used in an attempt to create a new file when all directory entries on the diskette are full or the file specification is invalid.

Type mismatch (Code 13): a numeric value is assigned to a string variable, or a string is assigned to a numeric variable.

Undefined array (Code 38): the array was not created before it was referenced by a subprogram SHARED statement.

Undefined label (Code 8): a branching statement such as GOTO or GOSUB calls for a line that does not exist.

Undefined subprogram (Code 35): the subprogram that was called is not in the program.

Undefined user function (Code 18): a function is called before it was defined with a DEF FN statement.

Unknown volume (Code 74): A reference has been made to a diskette that isn't in the drive.

Unprintable error (Code 21): the ERROR statement is used to self-inflict an error, and the resulting error code is not one used by the Computer.

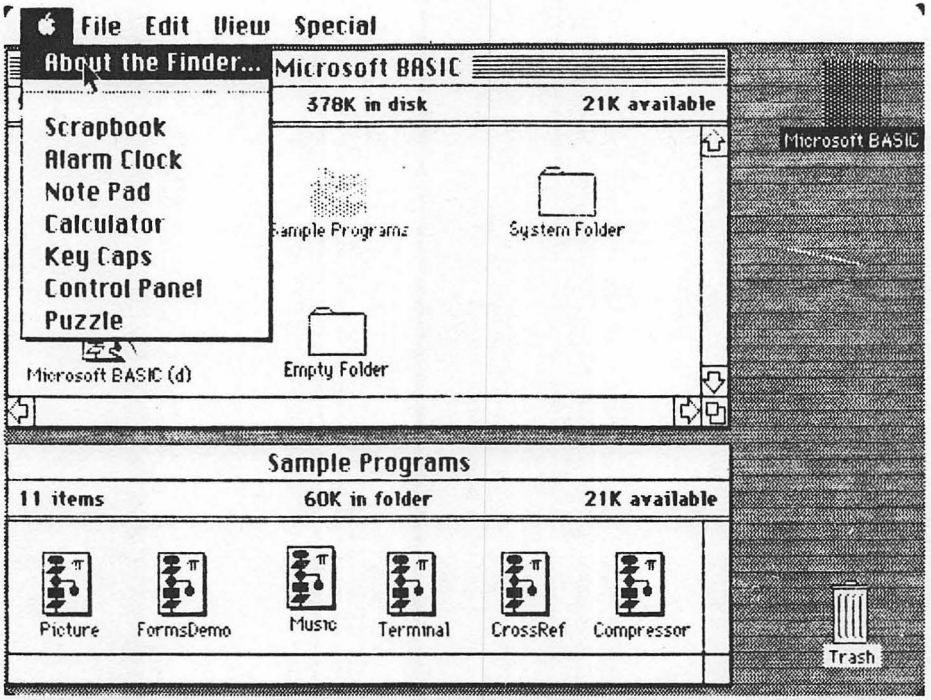
WEND without WHILE (Code 30): a WEND was found before a matching WHILE was executed.

WHILE without WEND (Code 29): a WHILE statement does not have a matching WEND. A WHILE was executing when an END, STOP, or RETURN statement was found.

Appendix D

The Apple Menu

In this section, we'll briefly cover the valuable functions of the (Apple) Menu.



Save any program that you may have in memory and Quit Microsoft BASIC to return to the Finder.

All About...

The very first item in the Apple Menu is the "All About" inquiry. This tells us some information about whatever applications we're currently using. For

example, if we were in the Finder, the first item in the Apple Menu would be "About the Finder..." and would tell us which version we're using.

If we were in Microsoft BASIC, this selection would be "About Microsoft BASIC..."

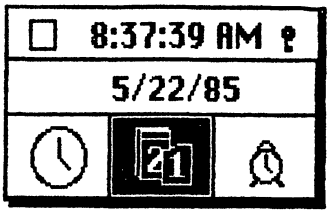
Alarm Clock

An extremely useful feature of the Macintosh is its built in alarm clock.

Select Alarm Clock from the Apple Menu.

After the disk drive stops spinning, a small box with the current time will be displayed. Move the pointer to the little switch that resembles a flag on a mailbox. Then click the mouse.

More boxes appear, with icons for changing the date and time, as well as the alarm clock. Right now, let's set the alarm.



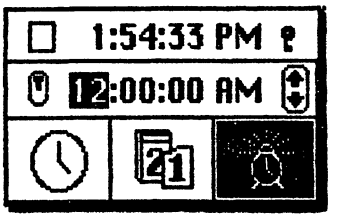
Move the mouse pointer to the box that has the illustration of an alarm clock and click the button. The box will darken and an hour/minute/second format will appear underneath the current time display.

To set the time, move the pointer to the hour's position and click the button. Two arrows will appear, one pointing up and the other pointing down. Move the mouse pointer to either one of these and press the button. If you click it once, the hours will change by one. But if you press and hold the button over one of the arrows, the hours will rapidly increase or decrease, depending on which arrow you point to.

Follow the same procedure for the minutes and seconds. Set the alarm to go off within five minutes of the current time.

Once the alarm settings are where you want them, point the mouse to the

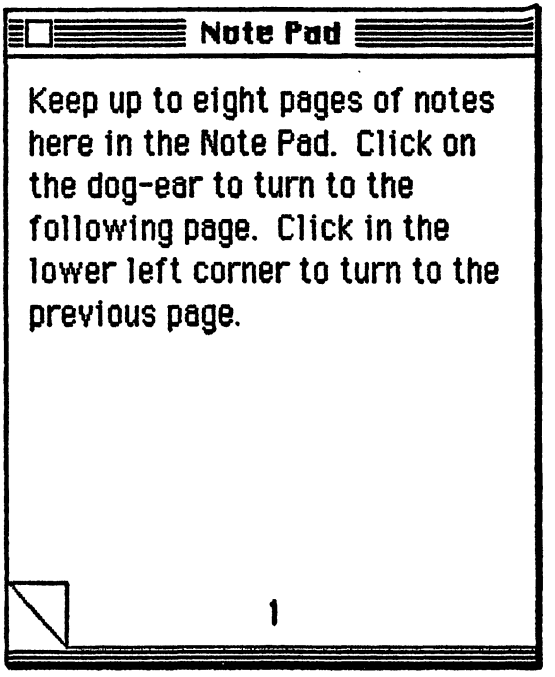
small switch to the left of the alarm time and click. If the switch is up, the alarm is set to go off.



To close the panel on the alarm clock, click the flag. To put the clock away, click the close box.

Note Pad

When the phone rings and you have to jot down a quick telephone number, you'll always have the Computer's eight page Note Pad ready. Select Note Pad from the Apple Menu.



To write on the Note Pad, simply type at the keyboard. (The disk must not be write-protected). Full editing and word wraparound features are included.

To turn to the next page, point the mouse button inside the small triangle of "paper" that is pulled back for us down in the lower left corner of the pad, then click the mouse button. Zip! Macintosh turns the page for us! To go back a page, point at the leftmost lower corner of the pad and click.

Let's save the instructions on page 1. Turn to page 2, and type in this short memo (we'll need it for testing another item of the desktop accessories):

```
6/1  Taxes due on ranch
6/9  Budget meeting
7/21 Meet with broker
7/22 Meet with Swiss Banker
7/23 Meet with IRS
7/24 Meet with travel agent
7/25 Leave for Brazil
```

Now, using the mouse-shading techniques we learned for editing lines, use the mouse to shade the whole memo. Press and hold the mouse button and move it up. When the whole memo is shaded, select Copy from the Edit menu. Then remove the Note Pad by clicking its close box. The memo will be saved when the box is closed.

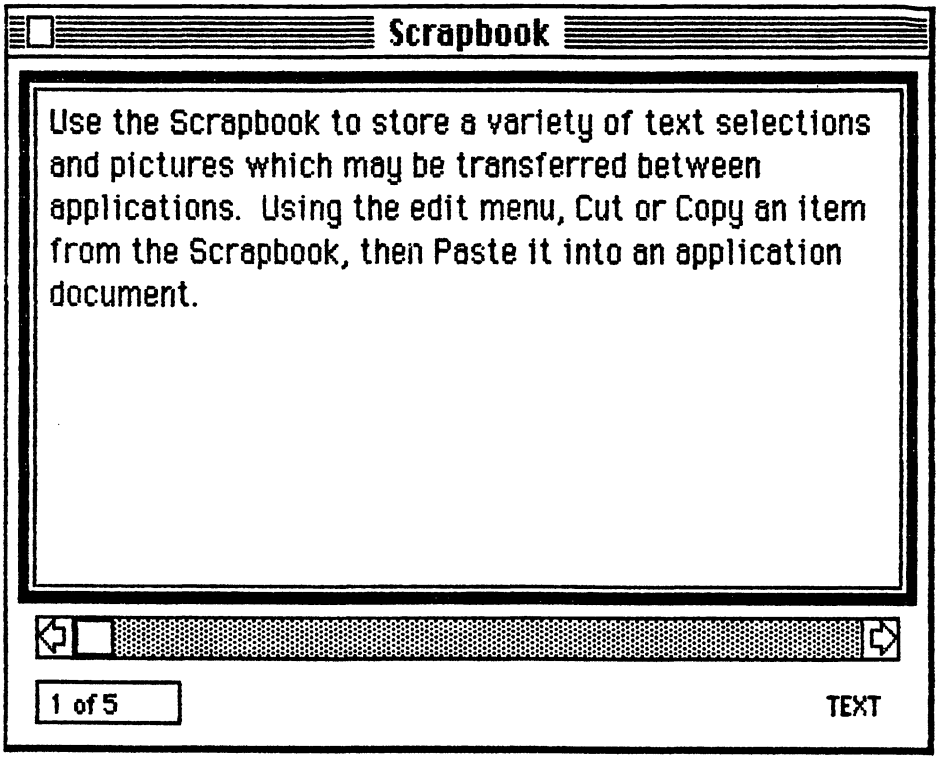
By the way, did the Computer beep while entering that memo? If not, then it probably will shortly. That is the Alarm on the clock.

Curious to see where that copy of the memo is? Pull the Edit menu down again and select Show Clipboard. Aha!

Scrapbook

Our most frequently used notes, letters, or MacPaint pictures can be Pasted into the desktop's Scrapbook. Select Scrapbook from the Apple menu, and take a moment to look it over. Below the scrapbook "page," note the scroll bar with its scroll box, the page counter which shows we are on page "1 of 5" pages, and the word "TEXT" which identifies the contents of this first page. To see what's on the other pages, put the pointer in the scroll box and drag it left. The page counter changes to indicate what page we are looking at and "TEXT" becomes "PICT."

All transactions with the Scrapbook are done through the Clipboard, and we just happen to have something in our Clipboard (remember when we copied the memo from the Note Pad in that last section?). Let's Paste it into the Scrapbook. Select Paste from the Edit menu. The disk drive will spin and soon a copy of page two of our Note Pad will be in the Scrapbook. Look at the page counter. Our memo has become page 1 of 6 pages.

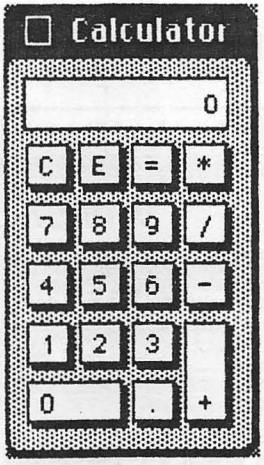


From here, we can take Scrapbook items, and Paste them into some applications programs (e.g. MacPaint, MacWrite). Click the close box to put the Scrapbook away.

Calculator

This is truly a handy one. Whip out the Calculator from the Apple Menu.

Using the mouse or keyboard, we can enter numbers and perform normal calculator functions. If you can imagine, this calculator will support a number up to 9E+4950 (that is, 9 with 4,950 zeros after it!) without generating an error.



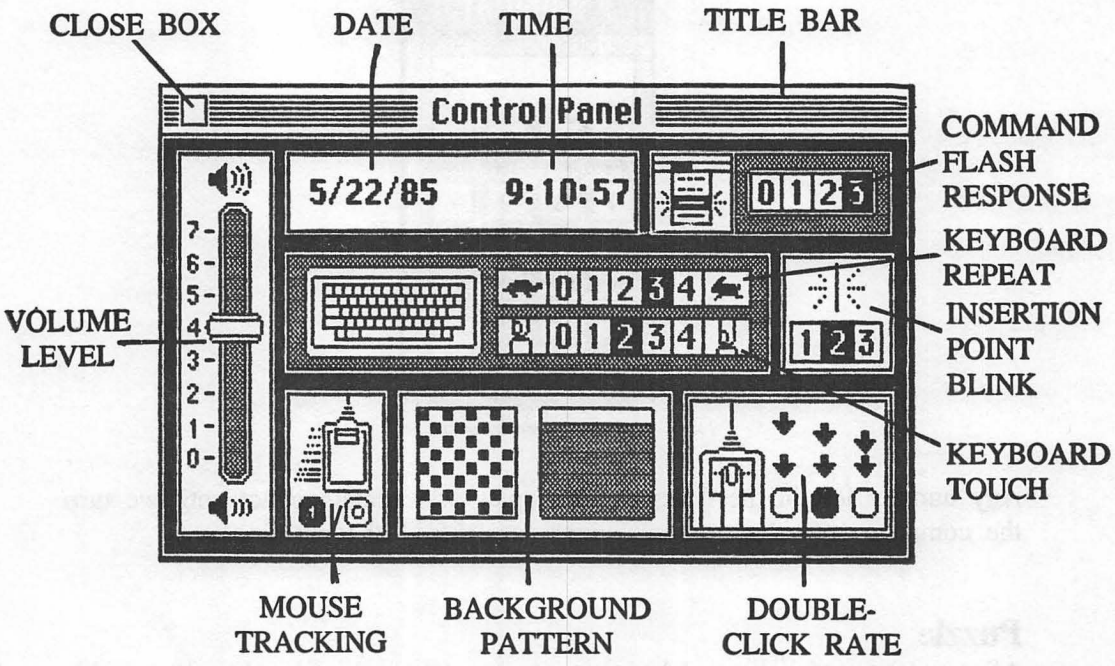
Any number left in the Calculator's display will remain intact until we turn the computer off, change applications, or RESET the machine.

Puzzle

After a long and tedious debugging session, take a breather by playing this frustrating game. (They used to be plastic when I was a kid. It was fairly easy to cheat by prying off those slippery digits and putting them back in the right order. Not so simple anymore!)

Control Panel

We can modify many of Macintosh's operating functions, like the volume level of Macintosh's speaker. Using the mouse, select Control Panel from the Apple Menu.



Volume Level

To adjust the volume from moderately loud to silent, move the pointer to the slide and press the mouse button. Drag the slide up for a higher volume level or stop it on zero to shut it off completely. When we let go of the button, Macintosh will beep, so we can check the new volume level.

Mouse Tracking

Mouse tracking can be adjusted for two speeds, proportional to movement (0) or exaggerated (1). Some applications that require precise mouse movement may need the tracking set to 0.

Background Pattern

The Control Panel will also allow us to customize the background pattern or choose one from a large selection. To edit the current pattern, click the mouse (+) inside the pattern editing window. To save the change, click the mouse in the pattern viewing window just to the right of the edit window. If we want to select a predefined pattern, click the white bar above the pattern viewing window.

Double-Click Rate
We may also change the speed of the double-click; necessary for Opening documents and applications programs.

Insertion Point Blink
Even the rate of the blinking insertion point can be adjusted. A higher number produces faster blinking.

Command Flash Response
Each time we select a menu item, Macintosh gives us some visual feedback by flashing the selection. The number of blinks can be changed in the Command Blinking box.

Setting the Date and Time
In the same manner as we set the alarm clock, we can choose the hours, minutes, or seconds in the clock window and then adjust them using the up and down arrows. Use the same procedure for changing the date. After the changes are set, click the mouse anywhere inside the Control Panel to implement the new date and time.

Keyboard Repeat
The rate of keyboard repeat can be adjusted by changing a value in the keyboard window (0 is the slowest, and 4, obviously, is fastest).

Keyboard Touch
Even keyboard "touch" can be adjusted. A setting of 4 will require us to press a little harder on the keys, whereas a 0 value will accept the lightest keypress.

(See Chapter 21 The ASCII Set for information about the Key Caps desktop accessory).

SECTION D

INDEX

A

Abbreviations 155
ABS 54,211
Alarm Clock 443
AND 329
Apple Menu 162,442
Answers to Exercises 405
APPEND 363
Arithmetic Functions 36,209
Arrays 288
ASC 164
ASCII 75,161,367
  Chart 162,285,430
Assembly Language CALL 340
ATN 221

B

Back Space Key 16,21
Backup 11
BASIC 13,15
BEEP 345
Brightness Adjustment 12,19,385
Byte 70

C

Calculator (Direct
  Command) Mode 65,446
CALL 340
Caps Lock 9
Caret 210,278
CDBL 205
CHAIN 372
Characters
  ASCII 161,430
  declaration 201
  special 161
CHR$ 163
CINT 205
CIRCLE 350
CLEAR 299
Clipboard 24
Clock
  setting 195,443
  DATE$ 195
  TIME$ 132,195
CLOSE 359
CLS 65
Codes
  ASCII 161,430
  error 398,436
Comma 42,268
Commands 20
  window 14,65
Command . 20,76
Command E (⌘E) 12
Command L (⌘L) 18,76
Command S (⌘S) 86
COMMON 374
Common Log 215
Compressed Format 75,367
Computer Program 15
Concatenate (+) 175
Conditional Tests 57
CONT (Continue) 87,92
Control Panel 447
COS 218
CSNG 205
CSRLIN 248
Cursor (insertion point) 14,16
Cut (⌘X) 22

D

DATA 141
Data Processing 357
DATE$ 195
Debugging 383

Define Statements
DEFDBL 202
DEF FN 223
DEFINT 204
DEFSNG 203
DEFSTR 174
DELETE 91
Desktop 7
Dialog box 9
DIM 294
Disk 8
Disk Eject (⌘E) 8
Division 36
Double-Precision 48,200

E

Edit 21
Ejecting A Disk (⌘E) 8
ELSE 159
END 32,107
EOF 364
EQV 335
ERASE 300
ERL 401
ERR 401
ERROR 397
Error Codes and Messages 27,397,436
EXP 215
Exponential notation 46,212
Expressions
logical 329
numeric 66
relational 54
string 145

F

FILES 73,367
File
buffer 358
extensions 371
length and characters 75
Find Next (⌘N) 153
FIX 209
Flowcharting 378
Fonts 341
FOR-NEXT 76
FRE(0) 14,69
Functions
arithmetic 209
defined 223
integer 109
intrinsic 209
string 145
trigonometric 218

G

GET 352
GOSUB 125
GOTO 54,77
Graphics 228
Guided Tour 5

H

HEX$ 167
Hexadecimal Conversion 166

I

IF-THEN 54,159
IF-THEN-ELSE 159
Image Line, PRINT USING 263
IMP 335
Initialize A Disk 8
INKEY$ 255

INPUT 60,157,276
INPUT# 361
INPUT$ 261
Insert Bar 14,21
INSTR 188
INT 49,109,209
Integer division 117
Integer precision 204
Interpreter 61
Interrupt (Reset) Switch 5,11,20
Inverse trigonometric functions 221

K

Key Caps 162
Keyboard Buffer 198
KILL 371

L

LCOPY 285
LEFT$ 183
LEN 173
LET 38
LINE 242
LINE INPUT 166
Line Labels 26,125
Line length 159
Line Numbers 26,32,39
Line Printer 282
LIST (⌘L) 18,76,89
List Window 14,31
LLIST 282
LOAD (Open) 73,357
LOCATE 246,251
LOG 212
Logical Operators 329
LPRINT 281
LPRINT TAB 282
LPRINT USING 284

M

Matrix 293
MERGE 368
MID$ 183
Modes
  Calculator 65
  Immediate Mode 65
Modular Programming 378
MODulo 212
MOUSE 345
Multi-Dimension Arrays 310
Multiple Statement Lines 151

N

NAME 372
Natural Log 213
Negation 36
NEW 14
NEXT 77
  optional 157
NOT 335
Note Pad 444
Numeric Variables 38

O

OCT$ 167
Octal Conversion 166
ON ERROR GOTO 397
ON GOSUB 127
ON GOTO 121
OPEN 358
Operators
  arithmetic 36
  logical 329
  relational 54
OPTION BASE 294

Option key 161
OR 329
Order of Operations 49,337
Output Window 14

P

Parentheses 49
Paste (⌘V) 22
PEEK 322
Pixel 228
POINT 351
POKE 322
POS 101,248
Precision 13,200
PRINT 16,30,156
PRINT# 361
PRINT TAB 97
PRINT USING 263
PRINT# USING 360
Print Zones 42,84,97
PRESET 228
PSET 228,355
PTAB 350
PUT 352
Puzzle 447

Q

Question Mark 156
Quitting BASIC 19,159
Quotation Marks 16,42,156,171

R

RANDOMIZE 134
READ 141,169
Relational Operators 54
REM 31,156
Reserved Words 434
Reset (Interrupt) Switch 5,11,20
Resident Program 38
RESTORE 145
RESUME 400
RETURN 126
RIGHT$ 183
RND 49,130
ROM Routines 340
RUN (⌘R) 18,90

S

SAVE (Save As) 72,357
Scientific Notation
(see Exponential Notation)
Scrapbook 445
Screen Dump 285
Searching 152
Semicolon 42,61,101
SGN 124
Show List Window (⌘L) 31
Show Second List 31
SIN 218
Single-Precision 48,200
Size Box 15
Sort 302
SPACE$ 193
SPC 193
Special Characters 161
SQR 210
Start (⌘R) 18
Statement
conditional 57
define 223
program 29
unconditional 57
STEP 78,237,351
Step (⌘T) 395

STOP (⌘ .) 20,76,90
String
arrays 298
comparisons 168,173
data 169
matrices 315
variables 145,155
STR$ 182
STRING$ 192
Subprogram 27,125
Subroutine 124
Suspend (⌘ S) 86
SWAP 344
Syntax Error 402
SYSTEM 159

T

TAB 97
TAN 218
Text Mode (ASCII) 75,367
THEN 55,58
TIME$ 132,195
TIMER 135
Title Bar 15
TO 77
Trigonometric Functions 218
TROFF (Trace Off) 393
TRON (Trace On) 393
Turning the Computer Off 12
Turning the Computer On 5

U

Unconditional branching 57
Upper Case Mode
Caps Lock Key 9
USING 263

V

VAL 179
Variables
classifying 38,145
define 174
names 39,154,155
reserved words 434
subscript 289
VARPTR 342
Video Display
clearing 65
graphics 228

W

WHILE-WEND 107
WIDTH 68,84
Windows
BASIC 7
Command 14
Enlarging 15
List 14
Output 14
Second List 31
WRITE 249
Write Protect (Locking Disk) 8

XYZ

XOR 335

+ 38,274
− 38,274
* 38,266
^ 210,278
/ 38
\ 117,270

( ) 49
> 54
< 54
= 54
<> 54
<= 54
>= 54
! 202,275
204,263
% 204,264
$ 145,265

Also Available From CompuSoft Publishing:

The BASIC Handbook, Third Edition,
Encyclopedia of the BASIC language
ISBN 0-932760-33-3 $24.95

Learning TRS-80 Model III BASIC *Also includes Model I*
ISBN 0-932760-08-2 $12.95

Learning IBM BASIC *for the Personal Computer*
ISBN 0-932760-13-9 $19.95

Learning TIMEX/Sinclair BASIC
ISBN 0-932760-15-5 $9.95

Learning TRS-80 Model 4/4P BASIC
ISBN 0-932760-19-8 $14.95

The TRS-80 Model 100 Portable Computer
ISBN 0-932760-17-1 $14.95

The IBM BASIC Handbook
ISBN 0-932760-23-6 $14.95

Learning Apple II BASIC *Includes II Plus and IIe*
ISBN 0-932760-24-4 $14.95

Learning Commodore 64 BASIC
ISBN 0-932760-22-8 $12.95

plus $2.00 Postage and Handling — $4.00 foreign orders

(California addresses add 6% sales tax)

COMPUSOFT® PUBLISHING
P.O. Box 19669, Dept. M2-1
San Diego, CA 92119

FREE

Update Information For

LEARNING MICROSOFT BASIC for the MACINTOSH

We can sit here and ponder and speculate and wonder all day long, but we'll never really know how we can improve this book in future editions unless you tell us. Please help us help you by giving us your suggestions for improvements. Honest, we really do read and learn from them!

What do you like about the book? _____

_____

What don't you like? _____

_____

Is the book complete? (If not, what should be added?) _____

_____

Did you find any mistakes? (If so, where?) _____

_____

_____

What other books, manuals or computer aids could be developed to help you? _____

Anything else? _____

_____

If you would like to receive the latest update memorandum (when available) and information regarding new releases, complete the following:

Name: _____

Address: _____

City/State/Zip: _____

Mail to:

CompuSoft Publishing
535 Broadway, Dept. M2-1
El Cajon, CA 92021

$19.95

MICROSOFT®

LEARNING BASIC FOR THE MACINTOSH™

The Macintosh represents a new way of thinking in the volatile world of personal computers. With its icons and mouse and windowing software, it may well be the beginning of a new generation. Learning to operate the computer and the available "canned software" is actually fun. But for many of us, the real fun comes from programming our *own* software in BASIC.

Learning Microsoft BASIC for the Macintosh picks up right where Apple's own *Guided Tour of Macintosh* leaves off. After a few minutes of introduction with the Guided Tour tape cassette, you move directly into writing your first computer program, and don't stop learning until you've mastered the language.

NO EXPERIENCE NECESSARY

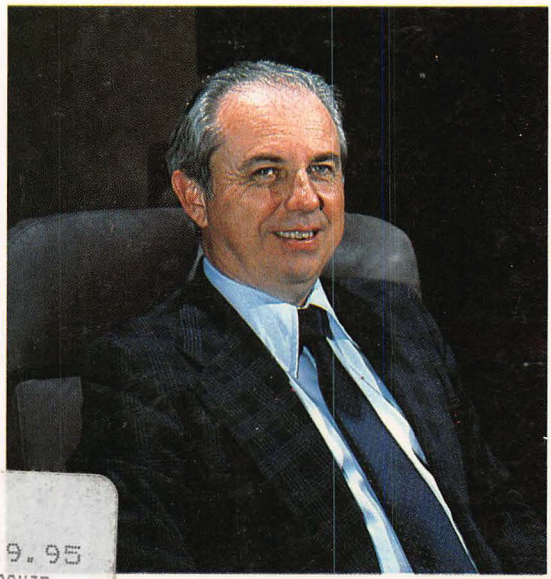
Open the box, then open this book. It's that easy to get started. You're never left to figure out what's going on by yourself, (although you will be challenged with exercises in nearly every chapter). No difficult concepts or "computer words" are used that aren't fully explained. Every chapter is built on previous ones, so you're always ready for what's coming next.

Get the full benefit of owning a Mac; learn to program it. Dave Lien has been teaching BASIC for years, and he's taken all the mystery out of it. His relaxed, hands-on style is responsible for millions of confident new BASIC programmers who once thought they'd never be able to learn such a "complicated" subject.

ABOUT THE AUTHOR

Dr. David A. Lien is one of the world's most widely acclaimed technical authors. His technique has been developed over many years of teaching Electronics, Mathematics, Computer Science and Programming.

He has well over a million book sales to his credit, including such popular titles as: *The BASIC Handbook, Learning Apple II BASIC, Learning IBM BASIC, Learning Commodore 64 BASIC, Learning TRS-80 Model 4/4P BASIC* and the *Epson MX Printer Manuals.*



PRINTED IN USA

0-932760-34-1