

COMPUSOFT
LEARNING SERIES

LEARNING COMMODORE 64™ BASIC

By David A. Lien



Learning

Commodore 64TM

BASIC

by
David A. Lien



COMPUSOFT[®]
PUBLISHING

A DIVISION OF COMPUSOFT, INC., SAN DIEGO

*Copyright © 1984 by CompuSoft Publishing,
A Division of CompuSoft, Inc. San Diego, CA 92119*

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior written permission of the publisher. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein. Portions of the material contained herein were originally created by the author for Radio Shack in support of the TRS-80 computer.

CompuSoft® is a registered trademark of CompuSoft, Inc.

This book contains official CompuSoft® software.

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

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

International Standard Book Number: 0-932760-22-8

Library of Congress Catalog Card Number: 84-71389

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America.

A Personal Note From The Author

Learning COMMODORE 64 BASIC combines the best of my earlier writings on the BASIC language plus much more, customized for the COMMODORE. It is written for the average person who has no experience with a Computer. The style is light and non-threatening since we have no insecurities to pass along. Learning should be fun, not intimidating...

And why shouldn't learning be fun...?

Sit back, relax, read slowly as though savoring a good novel, and above all, let your imagination wander. I'll supply all the routine facts and techniques we need. The real enjoyment begins when your imagination starts the creative juices flowing and the computer becomes a tool in your own hands. You become its master -- not the other way around. At that time it evolves from just an impressive looking box of parts into an extension of your own personality!

Enjoy your COMMODORE 64!

Dr. David A. Lien
San Diego -- 1984

Acknowledgements

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

Technical Director: Dave Waterman

Project Coordinator: Inez Goldberg

Project Editor: Terence L. Phife

Research and Development Team:

Steve Frostrom

Peter Krause

Terry Perry

Cathy Umphreys

Editorial Director: Gary Williams

Production Coordinator: Janice Scanlan

Cartoons: Bob Stevens

Introduction

Learning Commodore 64 BASIC is organized into five major sections:

- A. Thirty-nine chapters which teach how to use the many capabilities of your Commodore 64 ... in small enough bites so you won't choke. Many chapters include check points and examples.

At the end of most chapters 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 some User's Programs -- examples of interesting and practical programs ready to type right in and use. (Some are for fun, some for business, some for education, etc.).
- D. A section with Appendices which provide useful reference tables and charts.
- E. 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: Commodore 64 BASIC Tutorial	ix
Part 1. Getting Started	xi
1 Computer Etiquette	1
2 Expanded Program	7
3 The Screen Editor	15
Part 2. Speak To Me, Oh Great Computer	21
4 Math Operators	22
5 Scientific Notation	32
6 Using () And The Order Of Operations	35
7 Relational Operators	40
8 It Also Talks And Listens	46
9 Calculator Or Immediate Mode	51
10 Saving and Loading Using Disk	57
11 FOR-NEXT Looping	67
12 Son Of FOR-NEXT	77
13 Formatting With TAB	86
14 Grandson Of FOR-NEXT	92
15 The INTeger Function	98
16 More Branching Statements	108
17 Random Numbers	117
18 READING Data	126

Part 3. Strings	135
19 Smorgasbord	137
20 The ASCII Set	144
21 Strings In General	150
22 Measuring Strings	155
23 VAL And STR\$	159
24 Having A Ball With String	163
Part 4. Functions	175
25 Intrinsic Math Functions	176
26 The Trigonometric Functions	184
27 DEFined FuNctions	190
Part 5. Arrays	193
28 Arrays	194
29 Search And Sort	207
30 Multi-DIMension Arrays	215
Part 6. POKEing And Color Graphics	227
31 PEEK And POKE	228
32 Color With Commodore	235
33 Graphics with the 64	242
34 What's That Sound?	248
Part 7. Miscellaneous	255
35 Logical Operators	256
36 A Study Of Obscurities	263
37 GET	266
Part 8. Program Control	273
38 Flowcharting	274
39 Debugging Programs	280
Section B: Answers To Exercises	293
Section C: Prepared User Programs	313

Section D: Appendices	325
Appendix A -- ASCII Chart	326
Appendix B -- Reserved Words	329
Appendix C -- Error Messages	330
Appendix D -- Hex-to-Decimal Conversion Chart	333
Appendix E -- Screen Display Codes	334
Appendix F -- Musical Values	336
Appendix G -- Sound Settings	339
Section E: Index	343

SECTION A

**COMMODORE 64
BASIC
TUTORIAL**

PART 1

GETTING
STARTED



Computer Etiquette

From the moment we turn it on, our COMMODORE 64 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) 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 **64** 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!

Getting “READY.”

Hook up the Computer as shown in its manual, and turn it ON. We see:

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

```
READY.
```



The blinking rectangle is called the *cursor*. The Computer is telling us:

“I’m ready -- it’s your turn!”



To start off with a clean slate -- erasing all traces of prior programs or tests -- type **NEW** and press **RETURN**. The Computer again responds with:

READY.



Before we start programming, let's make sure the color is adjusted properly. Type a few letters (any letters) then press **RETURN**. Sort of hard to read, isn't it?

Hold down the **CTRL** key and press **2**. We just changed the color of printing from light blue to white, for better contrast. Type a few more letters to see the difference. Press **RETURN**. To change the background color from light blue to black, for even better contrast, carefully type:

POKE 53280,7 **RETURN**

If you're using a black-and-white TV, you'll just see a darker shade of grey.

We'll learn much more about color, changing background and cursor colors in a later chapter.

Holding down the **RUN/STOP** and **RESTORE** keys at the same time will return the display to "normal" light blue on dark blue.

What Is a Computer Program?

A program is a sequence of instructions the Computer stores until we command it to follow (or "execute") them. Most programs for the **64** are written in a language called BASIC, and the name itself tells how easy it is to learn!

Type a few letters at random. Notice they are in all capitals. Press the **SHIFT** key, type some more and see graphics characters! Interesting, but not very helpful to beginning BASIC programming, so let's write a simple one-line program and start a conversation with our Computer.

Type **NEW** and **RETURN** again, then type the following Line, *exactly* as shown:

```
10 PRINT "I ARE A COMPUTER PROGRAMMER"
```

Do *not* hit the **RETURN** key yet!

If you made a typing error, don't worry. Just use the **INST/DEL** key. Each time you press it, the rightmost character will be erased. If the error was at the beginning of the Line, erase way back to that point then retype the rest of the Line. (If you hold the **INST/DEL** key down, 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, telling us "I got the message".

If It's Too Late

If you found an error after pressing **RETURN**, the best way to fix it is by retyping the *entire* Line, correctly. When the **RETURN** key is pressed, this new Line will replace the old one since they both share the same starting number (in this case, 10). In several Chapters we'll learn how to "Edit" out errors instead of retyping entire Lines.

"Allow Me To Introduce Myself"

Let's tell the Computer to "execute" or RUN our program. The BASIC com-

mand for this is simple: RUN. So type:

RUN

and press **RETURN**.

If we made no mistakes, it will read:

I ARE A COMPUTER PROGRAMMER

If it doesn't work, try typing RUN again. If RUN still doesn't produce the statement, there's something wrong in your program. Type NEW **RETURN** to clear it out, then type it in and RUN again.

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

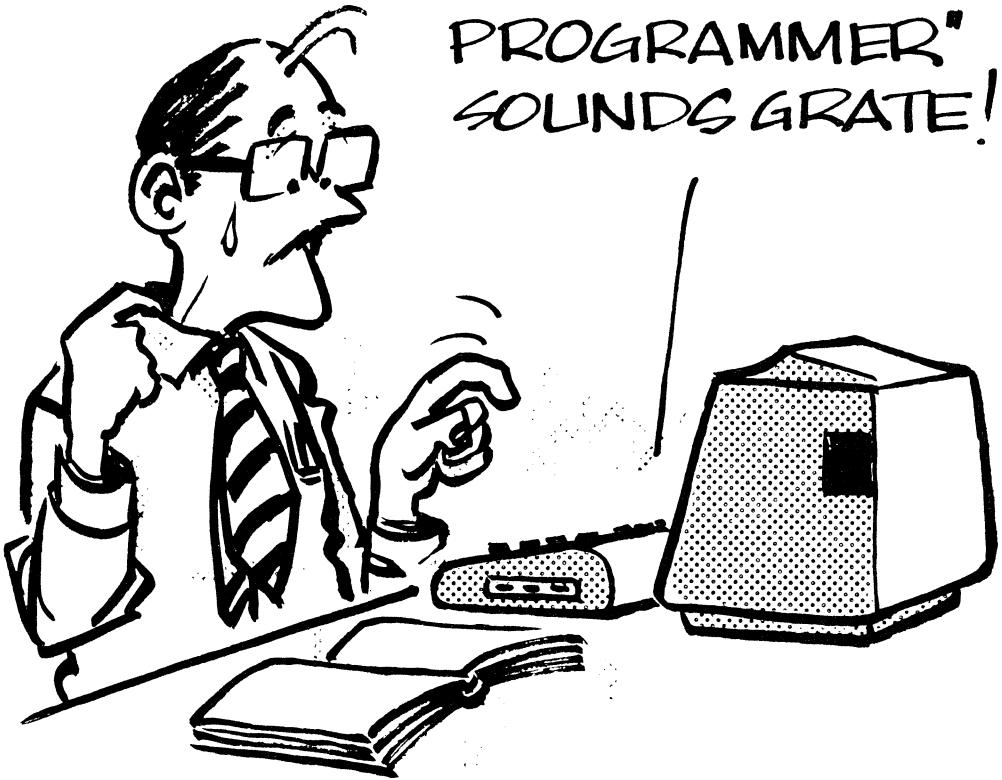
Note that the word PRINT is not displayed, nor are the Line number nor the quotation marks. They are part of the BASIC Language program's *instructions* and we didn't intend for them to be printed. Everything inside the quote marks is printed, including blank spaces.

Type the word RUN again and hit **RETURN**.

Type RUN **RETURN** 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.

Whether typing in a program or giving direct commands like RUN, we have to hit **RETURN** to tell the Computer to look at what we typed, and act accordingly.

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



Learned in Chapter 1

CommandsNEW
RUNStatementsPRINT
RETURN
POKE 53280,7Miscellaneous**INST/DEL**
cursor
" " quotation marks
Color
RUN/STOP and
RESTORE
CTRL

Commands (like RUN) are executed as soon as we type them and press **RETURN**.

Statements (like PRINT) are executed only after we type the RUN **RETURN** command.

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 give a READY message. To regain control, just press the **RUN/STOP** and **RESTORE** keys, like we did to reset the screen colors. If that doesn't work, turn the Computer OFF for 10 seconds, then turn it back ON again.

Expanded Program

We have a program in the Computer. It's only a one-Liner, so let's expand it by adding a second Line. In BASIC, every Line in a program *must* be numbered, and the instructions are executed in order from the lowest Line number to the highest. Type:

```
20 PRINT "YOU HAVE A COMMAND , MASTER?" RETURN
```

Check it carefully -- especially the quote marks, then:

```
RUN RETURN
```

Have you noticed that we use 0 for the number zero to distinguish between the letter O and number 0? The Video Display does it this way -- and it's standard throughout Computerdom.

If all was correct, the screen will read:

```
I ARE A COMPUTER PROGRAMMER  
YOU HAVE A COMMAND , MASTER?  
READY
```

■

If it ran Ok, answer the question by typing:

YES **RETURN**

Oh -- sorry about that! It "bombed", didn't it? The screen says:

? SYNTAX ERROR

We deliberately "set you up" to demonstrate the Computer's *error* troubleshooter. The Computer 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.



"Syntax" is an obscure word that refers to the pattern of words in a language. *Error* means we made one. The Computer is expecting a new program Line or a BASIC command. The word "YES" is neither. A bit later we'll learn how to make the Computer accept a "YES" or "NO" and respond accordingly.

There are many possible errors we can make, and in good time we will learn to understand the built in "Error Messages".

A complete listing of "Error Messages" is provided in Appendix C.

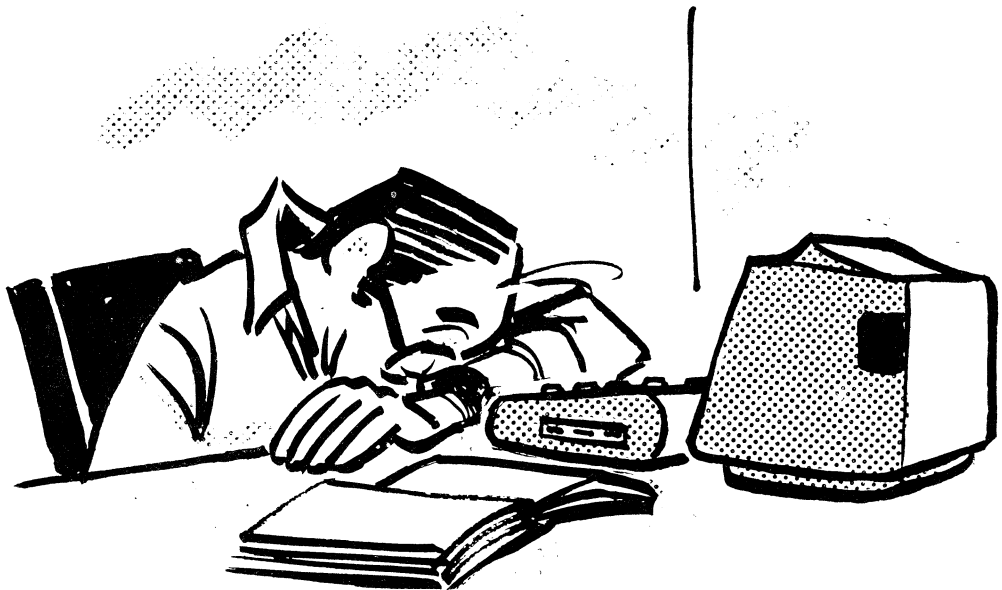
Meanwhile, there is one other important *error* situation which we must be able to recognize to pry ourselves out of accidental trouble. Let's type a temporary Line 30 and deliberately make a spelling error:

```
30 PRINT "TESTING."
```

and

RUN **RETURN**

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



Again we get an *error* message:

```
? SYNTAX ERROR IN 30
```

Erase the bad Line by typing:

```
30 RETURN
```

and it's gone. RUN to be sure.

And The Program Grows

It is customary, traditional (and all that) to space the Lines in a program 10 numbers apart. Note that our two-Line program uses the numbers 10 and 20. The reason ... it's much easier to modify a program if we leave room to insert new Lines in between the old ones. There is no benefit to numbering the Lines more closely (like 1,2,3,4). *DON'T DO IT.*

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:

```
15 PRINT RETURN
```

Then:

```
RUN RETURN
```

It now reads

```
I ARE A COMPUTER PROGRAMMER
```

```
YOU HAVE A COMMAND, MASTER?
```

Note: To make this book easier to read, we put more space between the program and display Lines than you actually see on the screen.

Looks neater, doesn't it? But what about Line 15? It says PRINT. PRINT what? Well -- PRINT *nothing*. That's what followed PRINT, and that's just what it PRINTed. But in the process of PRINTing nothing, it automatically inserted a space between the PRINTing ordered in Lines 10 and 20. (Hmmm...so *that's* how we space between lines.)

Didn't that room between Lines 10 and 20 come in handy?

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 by typing RUN **RETURN**, it skips right over any numbered Line which begins with a REM. A *REM statement has no effect whatsoever on the program*. Insert the following:

```
5 REM THIS IS MY FIRST COMPUTER PROGRAM RETURN
```

Then:

```
RUN RETURN
```

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

Well, this programming business is getting complicated and I've already forgotten what is in our "big" program. How can we get a LISTing of what our program now contains? Easy. A new BASIC command. Type:

```
LIST RETURN
```

The screen reads:

```
5 REM THIS IS MY FIRST COMPUTER PROGRAM
10 PRINT "I ARE A COMPUTER PROGRAMMER"
15 PRINT
20 PRINT "YOU HAVE A COMMAND MASTER?"
```

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 the end so they will know when to stop. With our **64**, an END statement is optional. By tradition, END is given the Line Number 99, or 999, or 9999 (or larger), depending on the largest number the specific computer will accept. The **64** accepts Line numbers up to 63999.

Let's add an END statement:

Type:

```
99 END      RETURN
```

Then:

```
RUN      RETURN
```

The sample RUN should read:

```
I ARE A COMPUTER PROGRAMMER
```

```
YOU HAVE A COMMAND, MASTER?
```

Question: "Why wasn't the word END PRINTed?" **Answer:** Because nothing is PRINTed unless it is the "object" of a PRINT statement. How could we make the Computer PRINT THE END at the end when program execution is finished? Think for a minute before reading on, and typing the next Line.

```
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 **64** will accept, 63999. It requires two separate steps.

First, we erase Line 99. Note that we're not just making a change or correcting an error in Line 99 -- we want to completely eliminate it from the program. Easier done than said.

Type:

```
99          RETURN
```

The Line is erased. How can we be sure? Think about this now. Got it? Sure -- "pull" a LISTing of the entire program by typing:

```
LIST       RETURN
```

The screen should show the program with Lines 5, 10, 15, 20, and 98. 99 should be gone. Any entire Line can be erased the same way.

The second step is just as easy. Type:

```
63999 END  RETURN
```

...and the new Line is entered. Pull a LISTing of the program to see if it was. Was it? Now RUN the program to see if moving the END statement changed anything. Did it? It shouldn't have.

Other Uses For END

Move END from number 63999 to Line number 17, LIST then RUN.

What happened? It ENDED the RUN after PRINTing Line 10 and a space. RUN it several times.

Now move END to Line 13, LIST and RUN. Then to Line 8, LIST and RUN.

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

Learned In Chapter 2

Commands

LIST

Statements

PRINT (Space)

REM

END

Miscellaneous

Error Messages

Line Numbering

The Screen Editor

The **64** uses what is commonly referred to as a screen editor. That means we can edit anything that is displayed on the screen by simply placing the cursor over it and making the change.

Cursor Movement

On the bottom right hand corner of the keyboard are two keys named **CRSR**. Press the key with the up and down arrows. The cursor should move down one space. Press it again, and it moves down another space. Hold down the key and it “repeats”.

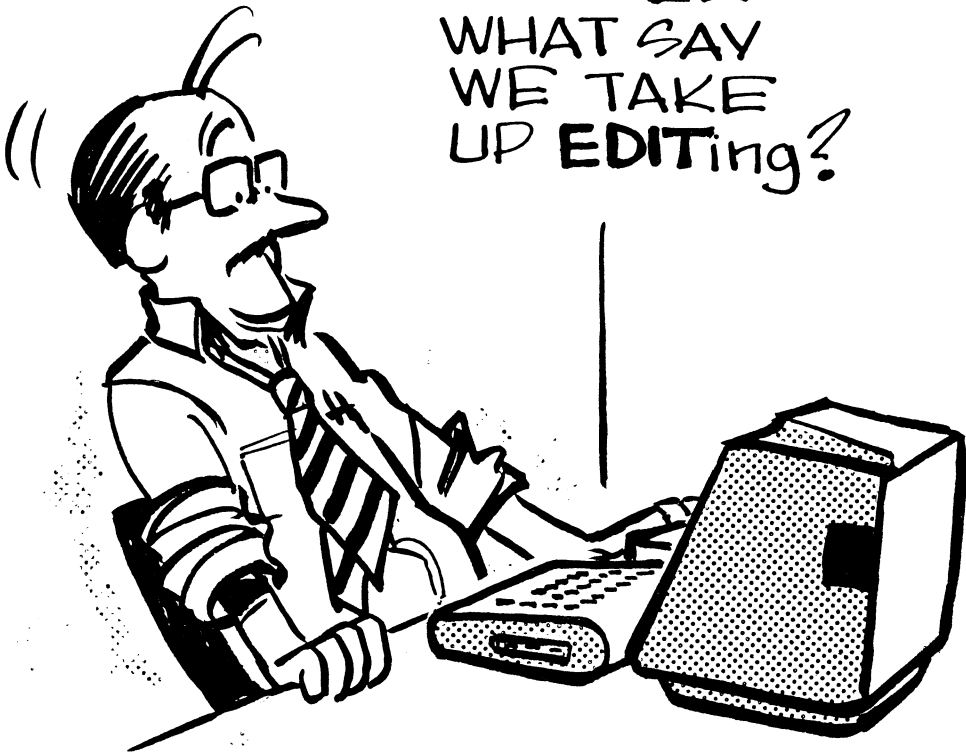
Try the **SHIFT** key in conjunction with the cursor key. The cursor moves upward. Now try the right and left arrow cursor key. The cursor moves right. With the **SHIFT** key the cursor moves to the left. That’s all there is to cursor movement.

The **CLR/HOME** key near the upper right hand corner of the keyboard returns the cursor to its “home” position at the upper left corner of the screen. Try it. Place the cursor in the middle of the screen and type some letters (any letters), then press **SHIFT CLR/HOME**. It CLearRed the screen and returned the cursor to its home. This CLR action does not erase program lines from memory, only from the screen. Type LIST to prove it.

Clear out the current program with NEW **RETURN** and type in this line very carefully (errors and all):

```
10 PRINT "THIS CONFUSER ARE GRATE!" RETURN
```

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



If you make a mistake, press **RETURN** and start over. Type RUN when finished.

It should run just fine, and if that's the way you usually talk you're probably wondering why we are doing this. If, on the other hand, you wish to change the sentence to something like:

```
THIS COMPUTER IS GREAT!
```

then we need to edit Line 10.

Line Editing

In order for a line to be edited it must be displayed on the screen. The cursor control keys are used to place the cursor on the line to be changed. After the changes are made, press **RETURN**. If **RETURN** is not pressed, the changes made will not be recorded. If you change your mind and decide not to record changes, simply move vertically off the line, or press **SHIFT RETURN**.

Overstriking

Let's try it. Type LIST to bring up the program line. Use cursor control keys to place the cursor over the N in CONFUSER. Type the letters MPUT and CONFUSER changes to COMPUTER. To change GRATE to GREAT, place the cursor on the A and type EAT. Now press **RETURN** to record both changes. LIST to be sure.

Erasing Characters

One more change is needed to make the program grammatically correct. Place the cursor on the A in ARE and type IS. Press the spacebar once to remove the E.

The **INST/DEL** Key

But there is now an extra blank space. Press the **INST/DEL** key and the space disappears.

The **INST/DEL** key also has another function. When used in conjunction with a **SHIFT** key it allows a single character to be inserted. To insert more than one character the **SHIFT INST/DEL** keys need to be pressed once for each additional character.

Let's add the word **NEW** to the program line. Place the cursor on the **C** in **COMPUTER** and press **SHIFT INST/DEL** four times. Now type **NEW** and hit **RETURN**. The finished program line should read:

```
10 PRINT "THIS NEW COMPUTER IS GREAT!"
```

Skipping Lines

Each time the **RETURN** key is depressed the **64** scans the beginning of the line the cursor is on for command words. Commands like **LIST** and **RUN** are **BASIC** keywords that tell the Computer to perform a specific action. If a command is encountered by the cursor and **RETURN** is pressed, then that action will be performed. To avoid this problem while editing, press **SHIFT RETURN** to skip thru command lines without executing them.

Quote Mode

The **64** enters quote mode when a double quote is encountered before the **RETURN** key is pressed. After the double quote the cursor controls become inoperable and instead display reversed characters. Type in the following **NEW** line but don't press **RETURN** yet:

```
10 PRINT "
```

Now press the down, up, left, and right arrow keys just once. Below is a chart of the reversed characters that appear:

CURSOR KEY	APPEARS IN REVERSE MODE AS
DOWN ARROW	◻
UP ARROW	◻
LEFT ARROW	◻
RIGHT ARROW	◻

When the **64** encounters these reverse characters, it performs the function associated with that cursor command. For example, when the cursor encounters the down arrow pressed in quote mode, it moves down one line. Press **RETURN** and try this **NEW** program using the cursor control keys as functions inside a **BASIC** program. Press the keys which are enclosed with **<>** signs.


```

10 PRINT "<SHIFT><CLR/HOME>"
20 PRINT "COMMODORE<CRSR> 64"
30 PRINT "<CRSR><CRSR>KEYS"
40 PRINT "<SHIFT><CRSR><SHIFT><CRSR>CURSOR"

```

And LIST. The display reads:

```

10 PRINT "♥"
20 PRINT "COMMODORE 64"
30 PRINT "  KEYS"
40 PRINT "  CURSOR"

```

And RUN. The display reads:

```

COMMODORE
CURSOR    64
  KEYS

```

Let's analyze what happened here.

Line 10 cleared the screen and put the cursor in the upper left corner of the display.

Line 20 PRINTed COMMODORE, moved down one line, created a space, and then printed 64.

Line 30 moved the cursor two spaces to the right before printing KEYS.

Line 40 moved the cursor up two lines and then PRINTed CURSOR.

Remember, if reverse characters appear on the screen, the **64** is saying "I'm in quote mode". Press **RETURN** before trying to use the cursor keys to move the cursor inside a program line. Also, the **INST/DEL** key functions are not affected when used in quote mode.

That's the screen editor for the Commodore 64. Practice using the edit features until you have them mastered, or at least understood. Refer to this chapter whenever strange characters appear on the screen, or when a refresher course on editing is needed.

Learned in Chapter 3

Cursor Keys

CLR/HOME

INST/DEL

SHIFT|RETURN

Line Editing

Key Overstriking

Key Erasing Characters

Inserting Characters

Skipping Lines

Quote Mode

Reversed Characters

PART 2

SPEAK TO ME,
OH GREAT
COMPUTER

Math Operators

But Can It Do Math?

Yes, it can. Basic arithmetic is a snap for the **64**. So are highly complex math calculations -- when we write special programs to perform them -- and we will.

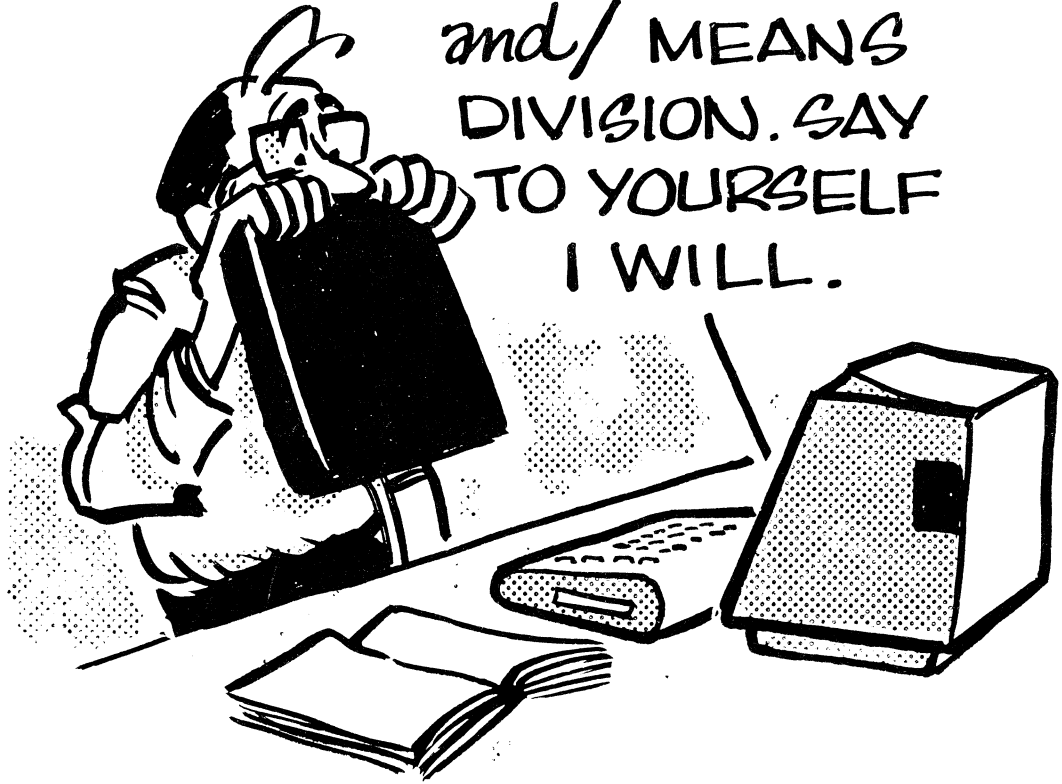
The BASIC Computer language uses the 4 fundamental arithmetic operations, plus 2 more complex ones which are just modifications of the others:

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 $\hat{=}$ (unveiled in the next chapter)
6. NEGATION, (meaning "multiply times minus one") using the - symbol

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



Note: all calculations on the **64** are performed to 10 places of accuracy, and are rounded to 9 when displayed on the screen.

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. In BASIC, "*" 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".

Type the command:

NEW **RETURN**

Then type:

LIST **RETURN**

to check that it's really gone. The Computer will respond with a simple:

READY.



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 acceptable to the Computer. Type in:

```
40 D = R * T          RETURN
```

Remember, we have to use * to specify multiplication.

What’s that 40 doing in our equation? That’s the program Line Number. Remember, every step in a program has to have one. We chose 40, but another number would have done just as well.

The extra spaces in the Line are there just to make the equation easier to read; BASIC ignores them. Later, when writing very long programs, you may want to eliminate extra Line spaces because they take up memory room. For learning, they are helpful, so leave them in.

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 for 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 **64** 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!)

Make the program read:

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

LIST and check the program carefully, then:

```
RUN RETURN
```


READY.



All it says is READY. **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 4-1: 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?)

`50 Print D`

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

Look in Section B of this Manual for an answer for this 1st Exercise. Also some notes and ideas.

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.

Note that we said `50 PRINT D`. There were no quotes around the letter D like we had 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. Type in the following:

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

Then:

```
RUN RETURN
```

(REMEMBER: Typing in a statement with a Line Number that already exists erases the original Line completely -- and that's what we want to do here. Could we have used the Editor instead of retyping? Yes.)

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. We can PRINT up to 4 items on the same Line, simply by inserting commas between them.

With this in mind, see if you can change 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. (Use the Editor).

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

Why is there all that extra space on both sides of the 6000 in the PRINTout? The reason is that the Computer divides up the screen width into 4 zones of 10 characters each. That adds up to 40 columns. 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. Use the Screen Editor to change the commas to semicolons (;).

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

```
RUN      RETURN
```

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 = + - * \div 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 4-2: 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.

EXERCISE 4-3: If the circumference of a circle is found by multiplying its diameter times π (3.14) write a program which will find the circumference of a circle with a diameter of 35 feet.

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

EXERCISE 4-5: 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.

Learned In Chapter 4

<u>Statements</u>	<u>Math Operators</u>	<u>Miscellaneous</u>
LET (Optional)	= + - * ^ /	, ; Variable Names 10 digit precision

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

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). This means, “the number 5 followed by six zeros.”

Technically, 5×10^6 , which is 5 times ten to the sixth power: $5 \times 10 \times 10 \times 10 \times 10 \times 10 \times 10$

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

This is 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 NEW, then type and RUN the following:

```
10 PRINT 5*10^11    (^ is next to the RESTORE key)
```

NO MORE NAGGING!
IT'S **RETURN** AFTER
EACH LINE OR f
COMMAND - TA DA f



What's this? The correct answer is $5E+11$, and we get $5.000000001E+11$! This is called "rounding error". Just accept it for now, we'll be talking about it in a later Chapter.

Type NEW before solving the following exercises:

EXERCISE 5-1: If 1 million cars drove 10 thousand 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) which will give the answer.

Didn't forget the **RETURN** did you? We've been reminding you to **RETURN** after each Line or command -- but from now on, we'll assume you can remember yourself.

EXERCISE 5-2: Change Lines 20 and 30 in the Car Miles Solution program (from Exercise 5-1) to express the numbers written there in exponential notation, or SSN (Standard Scientific Notation). Then RUN it.

Learned In Chapter 5

Miscellaneous

E - notation

(E stands for "exponent" and in our case it refers to the exponent of 10.)

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. We can have up to 10 levels of “nested” parentheses.

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). COMMODORE 64 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.

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 *
20 C = 25
30 F = (9/5)*C + 32
40 PRINT C;"DEG. CEL. =" ;F;"DEG. FAHR."
```

SAMPLE RUN:

```
25 DEG. CEL. = 77 DEG. FAHR.
```

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"

MY DEAR ALINT
SALLY- I DIDN'T
KNOW YOU
WERE PART OF
THIS!

THIS IS MY
S-L-O-W
NEPHEW!



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:

$$30 \quad F = 32 + (9/5)*C$$

RUN it again, without parentheses.

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 6-1: 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.”

$$30 \quad C = (F - 32) * \frac{5}{9}$$

EXERCISE 6-2: Remove the first set of parentheses in the Ex. 6-1 answer and RUN again.

EXERCISE 6-3: 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 6-4: Insert parentheses in the following equation to make it correct. Write a program and check it out on the **64**.

$$30 - 9 - 8 - 7 - 6 = 28$$

Learned In Chapter 6

Miscellaneous

()
Order of Operations

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 & 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 *if the value of A is exactly 5*, skipping over whatever might be inbetween Lines 20 and 50. 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.
```

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 the next Line.

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.

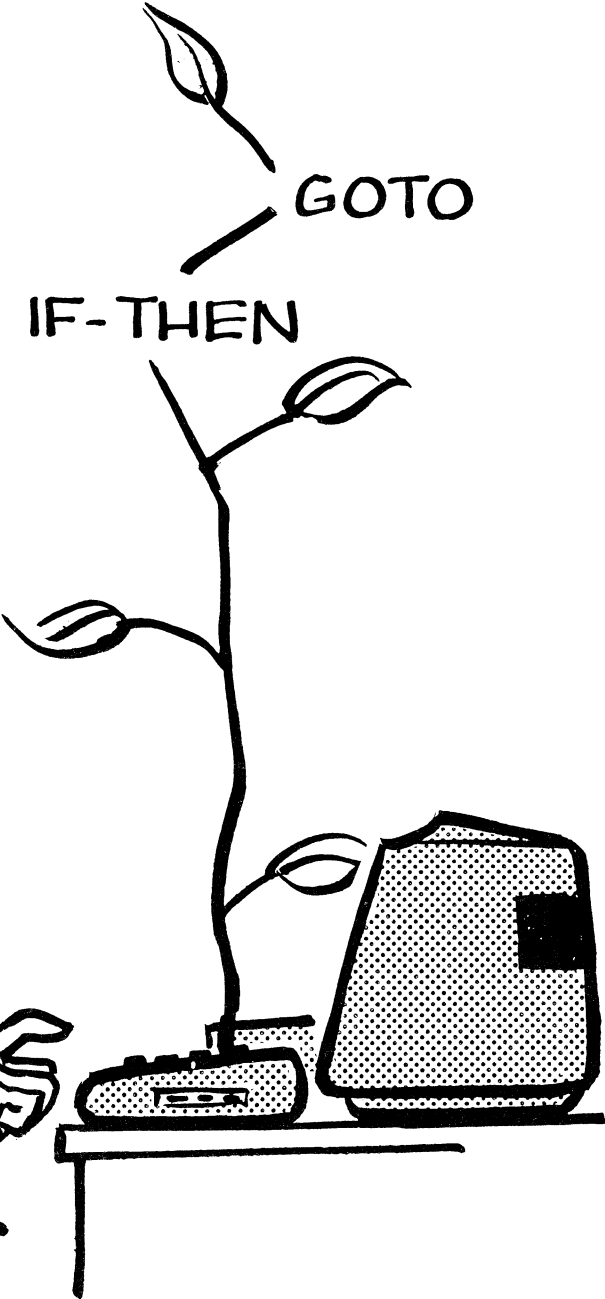
IF-THEN Vs GOTO

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
```


SORT OF
LIKE A FAMILY
TREE!



and add Line 99:

```
99 END
```

...whenever the Computer hit Line 40 it would *unconditionally* follow orders and GOTO 99, ENDing the RUN. Make the changes discussed above 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 7-1: 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 7-2: Change Line 10 to give A the value of 6. Leave the other four Lines from Exercise 7-1 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 7-3: 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 7

<u>Statements</u>	<u>Relational Operators</u>	<u>Miscellaneous</u>
IF-THEN	=	Conditional branching
GOTO	>	Unconditional branching
	<	Optional THEN
	<>	with GOTO
	<=	
	>=	

It Also Talks And Listens

By now you have probably become tired of having to retype 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 will print:

```
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, press **RETURN** and 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 "TYPE IN A VALUE FOR A";
```

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.

Did you use the Screen Editor to add the semi-colon?

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:

```
TYPE IN A VALUE FOR A? ■
```

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 COMMODORE 64 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 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. Also ask for the handbook I wrote especially for Commodore users, *The Commodore 64 BASIC Handbook*. It doesn't convert to other dialects, but is a thorough and very helpful reference for "your" BASIC.

Change Line 10 to read:

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

delete Line 20 by typing:

```
20 RETURN
```

...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 type RUN, the Computer executes the program, PRINTs the results (if any) and comes back with a READY. To repeat the program, we have to type RUN again. Can you think of another way to make the Computer execute a program two or more times?

BY GEORGE!
I THINK I'VE
GOT IT !!

I'LL KEEP ASK-
IN' FOR MORE
UNTIL YOU HIT

RUN/STOP

and

RESTORE



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 6). If you think GOTO is a powerful statement in everyday life, wait 'til you see what it does for a computer program!

Type NEW and the following:

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

...and RUN.

Hit **RUN/STOP** and **RESTORE** to exit the program loop.

The Computer will keep asking for more until we get tired, or the power goes off (or some other event beyond its control). This is the kind of thing a computer does best -- 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 5 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 life preserver.

Learned In Chapter 8

Statements

INPUT and
INPUT with built-in PRINT

Miscellaneous

; Trailing semi-colon

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 *omit* the Line number before certain statements and commands, the Computer will execute them, 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 the cursor ■.

EXAMPLE: How much is 3 times 4? Type in:

```
PRINT 3 * 4      RETURN
```

...the answer comes back:

```
12
```

EXAMPLE: How much is 345 divided by 123?

Type:

```
PRINT 345/123    RETURN
```

...the answer is:

2.80487805

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
```

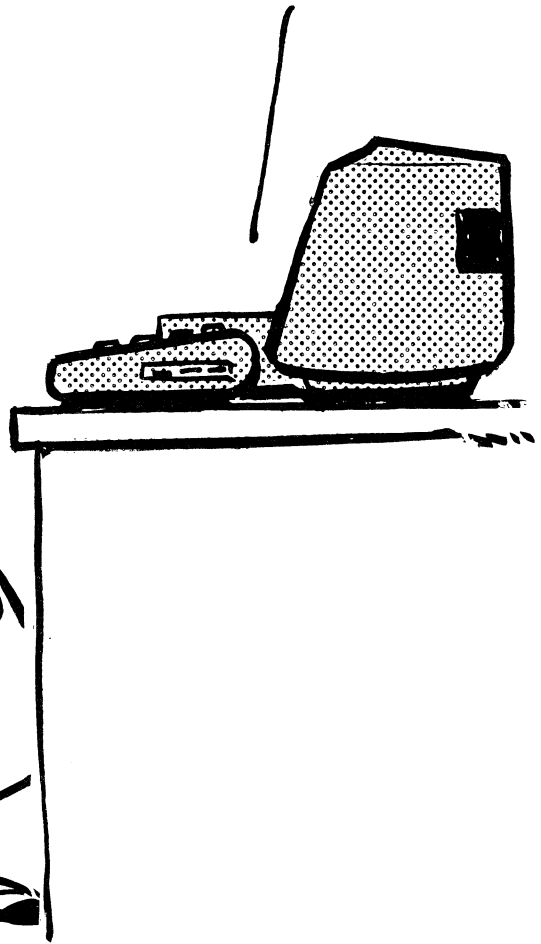
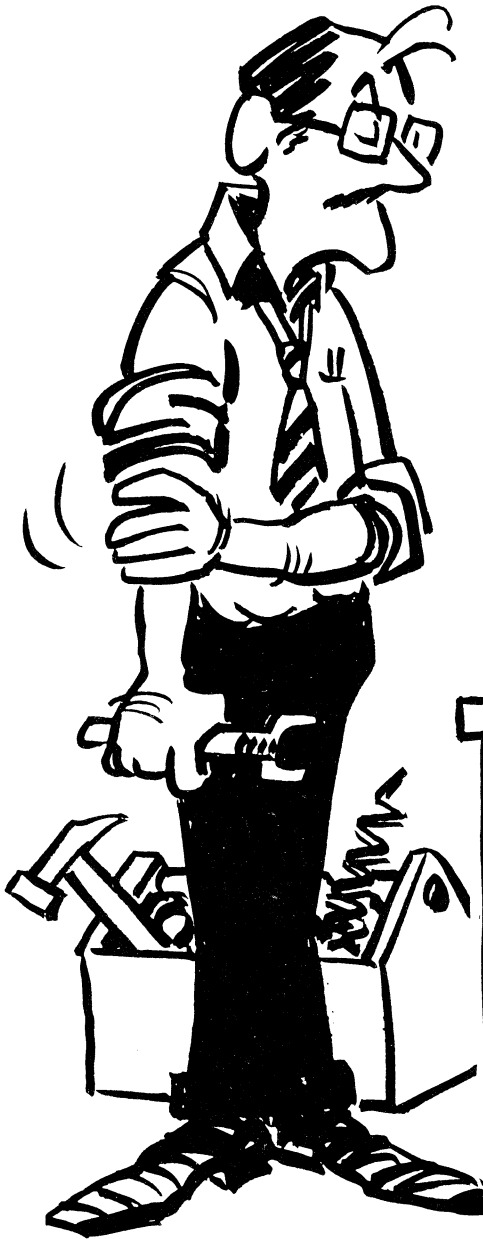
The Computer will PRINT the present value of X.

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

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

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



The answers 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. Typing RUN or NEW also "initializes" all variables to 0.

We will get all zeros if the machine was turned OFF since the last RUN.

The FRE(0) Command

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 of the available memory we have used up. That's what the FRE(0) Command is for.

In a "64K" computer there are about 64,000 different memory locations available to store and process programs. "64K" is just a shortcut phrase for the exact amount of memory, which is 65536.

The Computer uses some of the memory for program control. To see how much it has already used up, type:

```
NEW          RETURN
PRINT FRE(0) RETURN
```

...and the answer is:

```
-26627
```

With no program loaded, 26627 memory locations are being used by the BASIC language interpreter and overall management and "monitoring" of what the Computer is doing. To find out what's left for us to use, of course, we simply subtract 26627 from the total installed memory. Remember, in a 64K Computer, that's always 65536.

Type in this simple program:

```
10 A = 25
```

then measure the memory used by typing:

PRINT FRE(0) **RETURN**

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

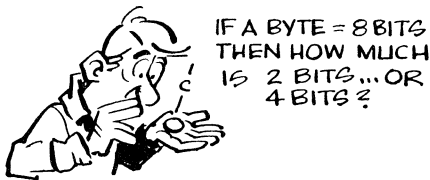
...the answer is:

-26638

The program we entered took $26638 - 26627 = 11$ 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 4 memory cells. The "carriage return" at the end of the Line takes 1 more byte, even though it does not PRINT on the screen. Thus, memory "overhead" for each Line, short or long is 5 bytes.

BYTE -- is the basic unit of storage in the COMMODORE 64 and most other microcomputers. In the **64** it is a string of eight binary digits (bits). Thus a byte = 8 bits.



2. Each letter, number and space takes 1 byte. In the above program 5 bytes for overhead + 6 bytes for the characters = 11 bytes.

Now, type RUN, then check the memory again with PRINT FRE(0). It's now up to 26645. 7 more bytes! When RUN, a simple variable like the A takes up 3 bytes and the numerical value takes another 4, totaling 7.

How could we find out how much memory we have left? Try immediate mode:
PRINT 65536 - 26645
Yep. 38891 bytes.

We will study 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 using `FRE(0)` when completing a program to develop a sense of its size and memory requirements.

Learned In Chapter 9

Functions

`FRE(0)` (+65536)

Miscellaneous

Calculator Mode
Memory
Byte

Saving and Loading Programs

We will soon write and run long and powerful programs. It becomes tedious to type them in accurately just once, let alone each time we want to use them. There has to be a better way...

Actually, there are two better ways: they're called tape and disk. Take this chapter slow and easy, and then reread it for better comprehension. There is a lot to know, and we don't want you to miss anything. If you are using a disk system, skip ahead now to SAVEing and LOADING on Disk.

SAVEing on Tape

In the following examples, we use the Commodore's 1530 Datassette Unit, Model C2N recorder. Be sure the **64** is off before you connect it to the Computer. Also, the recorder should be at least 2 feet away from the TV monitor to avoid radio and other interference.

Type in a short program. We will use it to experiment, so make sure it runs properly. Now all we do is follow the yellow brick road:

1. Make sure the DATASSETTE recorder is properly connected to the Cassette Interface.
2. Fully rewind the tape.
3. Type:

SAVE "PROGRAM ONE "

RETURN

"PROGRAM ONE" is the File Name used to store the program. File Names can be up to 16 characters long.

The **64** responds with:

PRESS RECORD & PLAY ON TAPE

4. Press the RECORD and PLAY buttons on the DATASSETTE simultaneously until they lock.

The screen will turn white (the color of the border) and the cassette motor will turn on. This indicates that the program is being SAVED. When the motor stops and the screen says:

SAVING PROGRAM ONE
READY.

STOP the recorder. The program is now on tape as well as in the Computer's memory. Type NEW and **RETURN** to erase the Computer's memory.

LOADing From Tape

This process is as easy as SAVEing. Follow steps 1 and 2 again, and then type:

LOAD "PROGRAM ONE" **RETURN**

The **64** responds with:

PRESS PLAY ON TAPE

Press the PLAY button and again the screen turns white (the color of the border). The **64** responds with:

SEARCHING FOR PROGRAM ONE
FOUND PROGRAM ONE

and the screen returns to normal display. (Note: in early cassette models the

G key needs to be pressed.) The program is loaded when the READY prompt appears. Be sure to STOP the recorder.

RUN the program to see that the LOAD was successful. In the event it was not, repeat the above steps, being sure that the connection to the Cassette Interface is good, and the recorder head is clean.

There are a couple of other methods for LOADING a program from cassette. If you want to just LOAD the next program on the tape, (first program if the tape is completely rewound), type LOAD and **RETURN**. Press PLAY on the recorder, the screen turns white, then back to the normal display. The **64** responds with:

```
SEARCHING
FOUND "Filename"
```

The screen turns white again, and the READY prompt appears. Type LIST to make sure the program is properly loaded.

If you want to LOAD and RUN the next program on tape, press **SHIFT RUN/STOP**. Press the STOP button on the recorder when finished.

Verify

VERIFY allows us to be sure the program SAVED on tape matches the one in memory, without errors. Type VERIFY, the filename enclosed in quotes, then press **RETURN**. Press PLAY to start the search. The **64** responds with:

```
SEARCHING FOR "Filename"
FOUND "Filename"
```

If the name of the file in the **64**'s memory matches the name of the file SAVED on cassette, the program is properly SAVED. The **64** responds with VERIFYING, OK, and then the READY prompt.

If the filename is omitted, the **64** checks the program in memory with the next program found on tape.

If at any time during the LOADING operation you change your mind, press the **SHIFT RUN/STOP** key. This aborts LOAD and returns the READY prompt.

Directory

The directory is a file which contains the names of all the files SAVED on tape. Its name is \$. Type:

```
LOAD "$"
```

and hit **RETURN**. Be sure the cassette tape is fully rewound, then press the PLAY button on the recorder. The **64** responds with:

```
SEARCHING FOR $  
FOUND "Filename"
```

When the **64** encounters a program, it responds with FOUND "Filename", and then continues to search for the next program. In between, the screen turns white. It continues until the user presses **RUN/STOP**. If there are several programs on cassette tape, it will take a while for all the program names to be listed. Press the STOP button to shut off the recorder. To "read" this DIRectory, simply LIST it.

Miscellaneous Tape Player

To minimize the chance of hitting a "soft spot" on a tape where the oxide may be thin or have flaked off, the **64** automatically SAVES programs twice. This also allows it to check for errors when it saves a program the second time. For extra safety, very important programs should be recorded on more than one tape.

If you record programs on long audio cassettes, use the counter on the recorder to aid in locating them. Just Fast Forward to the location ahead of where the program is stored and proceed as usual.

When not using the recorder for LOADing or recording, do not leave the RECORD or PLAY keys down (press STOP). It's hard on the tape drive mechanism.

To save a taped program permanently, break off the Erase Protect Tab on the cassette. When the tab(s) has been broken off, the RECORD button cannot be pressed down.

CAUTION! Do not expose recorded tapes to magnetic fields.

Non-Disk Users GOTO the next chapter.

SAVEing and LOADING on Disk

A big advantage of disk storage over cassette is that programs can be **SAVED** to or **LOADED** from disk quicker and with better reliability than cassette.

In order to use a diskette on the **64**, it must first be **FORMAT**ted.

Insert a blank diskette into the disk drive. Make sure there is no write-protect tab on it.

Type:

```
OPEN 15,8,15
```

and press **RETURN**. **OPEN** is a BASIC keyword for channeling information to or from a peripheral device (e.g. a disk drive or printer).

1. The first number is the file number and can range from 1 to 255. We chose 15 to make recall of the file number easier, as you will soon see.
2. The next number is the device number. The disk drive is identified by the number 8.
3. The final number is a "channel" number and can range from 2 to 15. 15 is the command channel and is used for sending information between the Computer and peripheral devices. Therefore, we must use 15. If the first number and the last number are the same, then the **OPEN** statement parameters will be easier to remember.

Anyhow, the Computer responds:

```
READY,
```

Now type:

```
PRINT#15,"NEWØ:Diskname,ID#"
```

What does this mean? Well, **PRINT#15** tells the Computer to print (write) information to the disk that we designed with the **OPEN** statement. When

PRINT# is used with the command channel (15) then information is sent to the disk drive.

Since this is a new diskette, we must put NEW0 into the statement. NEW0 erases the entire diskette and creates the directory.

Next comes the disk name and the ID number. The Disk Name can be any acceptable format up to 16 characters long. The ID number can be any two characters and is used to identify the disk name for data storage. Use "COMMODORE" for the disk name and 64 for the ID#. Press **RETURN**.

When the red light on the disk drive goes out, formatting is complete. Type:

```
CLOSE 15      RETURN
```

to close the channel. Practice formatting a couple of disks to be familiar with this new command, then proceed.

Now let's type a short program and SAVE it on disk.

```
10 PRINT "THIS IS A SHORT PROGRAM"  
SAVE "PROGRAM ONE",8      RETURN
```

The 8 tells the **64** to store the program on disk. If you forgot the ,8, press **RUN/STOP** and SAVE again. The Computer responds:

```
SAVING PROGRAM ONE  
READY.
```

Type:

```
NEW      RETURN
```

to erase the program that's in memory (LIST to check), then:

```
LOAD "PROGRAM ONE",8      RETURN
```

and we see:

```
SEARCHING FOR PROGRAM ONE
```

```
LOADING
READY.
```

Type:

```
LIST      RETURN
```

Yup, there is it.

```
10 PRINT "THIS IS A SHORT PROGRAM"
```

Backup

The **64** does not provide the ability to copy a complete diskette to another blank diskette. Earlier operating systems had dual drives (like the 4040) and backup was possible using the DUPLICATE command. Unfortunately, this command doesn't work with Commodore's 1541 single disk drive. Contact your Commodore software dealer for a special BACKUP utility program.

VERIFY

The VERIFY command checks the program in memory with the one SAVED on disk. If the files are not identical it gives us a VERIFY ERROR. Type:

```
VERIFY "PROGRAM ONE",8      RETURN
```

The program name is required. Remember, 8 specifies disk. If no number is used, the Computer will assume tape.

If everything is OK, the Computer will say:

```
SEARCHING FOR PROGRAM ONE
VERIFYING
OK
```

VERIFY is usually used to check a file immediately after it has been SAVED to make sure that all went well.

WELL, *OUR* SAVINGS
PROGRAM DOES *NOT*
INCLUDE DISKS!



The VERIFY command works in both Immediate and Program Modes (BASIC programs).

Scratch

Most microcomputers use the word KILL to erase unwanted files or programs from disk. The **64** prefers to SCRATCH them. Make sure the channel is open (OPEN command: OPEN 15,8,15).

To erase an unwanted file, type:

```
PRINT#15,"SCRATCHØ:Filename"      RETURN
```

The Ø after SCRATCH tells the Computer the disk drive number. Don't confuse this with the device number. The disk drive number is used instead of the device number if the channel is already open (i.e. with PRINT# commands). To check and see if the file is really gone, try loading it into memory.

Directory

The directory for disk files is created when a diskette is formatted. It is a file with the name \$. Type:

```
LOAD "Filename",8      RETURN
```

Now type LIST. There you go.

More Disk Stuff: ERROR

If the red light on the disk drive starts blinking, there is a problem related to the disk. To see what the error is, RUN the following program. (This would be a good program to save on disk for future use.) The Computer will respond with an error message. For a complete listing of error messages, see Appendix C.

```
5 REM * PRINTS ERROR STATUS OF DRIVE *
1Ø OPEN15,8,15
2Ø INPUT#15, DUMMY, E$
```

```
30 CLOSE15
40 PRINT:PRINT "ERROR: '";E$"' "
```

As an example, try to SAVE a program using a name that is already in use on the disk. Then RUN the program.

If we get into real trouble because of an error and the Computer won't let us do what we want, we may need to INITIALIZE the drive to its power-up state. This is accomplished by typing PRINT#15,"I" **RETURN**. It won't tell us what the error was, but will clear up the problem and put us back in business.

```
10 OPEN15,8,15
20 PRINT#15,"I"
30 CLOSE15
```

To protect a diskette from accidental changes, put a "Write Protect Tab" over the notch on the side. A "protected" diskette can't be written on.

See the User's Manual for a complete guide to using the disk system.

Learned in Chapter 10

Commands

OPEN
LOAD
SAVE
VERIFY
INITIALIZE
ERROR
PRINT#
SCRATCH
CLOSE

Miscellaneous

8 designates disk
formatting disks
File-Names
SHIFT RUN/STOP
Directory

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 and then the following program:

```
20 PRINT "HELP! MY COMPUTER'S BERSERK!"  
40 GOTO 20
```

...and RUN.

The Computer is PRINTing:

```
HELP! MY COMPUTER'S BERSERK!
```

and will do so indefinitely, until we tell it to STOP. When you have seen enough, hit **RUN/STOP**. This "breaks" the program RUN.

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'S BERSERK!"
40 NEXT N
60 PRINT "NO --- IT'S UNDER CONTROL."
```

...and RUN it.

The Line:

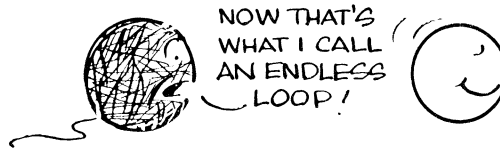
```
HELP! MY COMPUTER'S BERSERK!
```

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.



HELP! MY COMPUTER'S
GONE BERSERK!
HELP! MY COMPUTER'S
GONE BERSERK!
HELP! MY COMPUTER'S
GONE BERSERK!
HELP! MY COMPUTER'S
GONE BERSERK!
HELP! MY COMPUTER'S
GONE BERSERK!
HELP! MY COMPUTER'S
GONE BERSERK!
HELP! MY COMPUTER'S
GONE BERSERK!

1412

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.

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

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

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'S 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 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 pro-

gram 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 1500 mph, in increments of 100 mph. The program might look like this:

```
10 REM * TIME VS RATE FLIGHT CHART *
20 PRINT "SHIFT CLR/HOME"
```

See anything strange? When you pressed **SHIFT CLR/HOME** a “heart” appeared between the quotes! It’s the 64’s little graphic way of acknowledging that we want it to “Clear the Screen”.

```
30 D = 6000
40 PRINT "      LONDON TO SAN DIEGO"
50 PRINT "      DISTANCE =";D;" (MILES)"
60 PRINT "RATE (MPH)", "TIME (HOURS)"
70 PRINT
80 FOR R=100 TO 1500 STEP 100
90 T = D/R
100 PRINT R, ,T
110 NEXT R
```

Type in the program and RUN.

Try doing that one on the old slide rule or hand calculator!

It is really solving the $D = R \cdot T$ problem 15 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	8.57142857
800	7.5
900	6.66666667
1000	6
1100	5.45454546
1200	5
1300	4.61538462
1400	4.28571429
1500	4

Analyzing The Program

Look through the program and 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 clears the screen so we have a nice place to write. It allows us to write in a *top-down* manner. (RUN the program again leaving out this Line to contrast *top-down* with *scroll* mode, then put it back in.) You will want to use this often just to make your PRINTouts neat and impressive.

We can also Clear the Screen at the command level by pressing the **SHIFT** and **CLR/HOME** keys.

3. Line 30 *initializes* the value of D. D will remain at its initialized value.
4. Lines 40 through 70 PRINT the chart heading.
5. Line 60 uses *automatic zone spacing* to place those column headings (the comma).

Remember zone spacing? The comma (,) in a PRINT statement automatically starts the PRINTing in the next 10-space PRINT zone.

6. Line 80 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 1500 mph is reached. Line 110 is the other half of the loop.
7. Line 90 contains the actual formula which calculates the answer.
8. Line 100 PRINTs the two values. They are positioned under their headings by automatic zone spacing (the commas).

Take a deep breath and go back over any points you might have missed in this lesson. SAVE the program onto Tape or Disk as “LONDON1” because we will use it in the next Chapter, continuing our study of FOR-NEXT loops.

Learned In Chapter 11

Statements

FOR-NEXT
STEP

Miscellaneous

Increment
Decrement
Initialize
“Top down” Display
“Scroll” Display
“Do-Loop”

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 25 mph increment instead of the 100 mph increment presently in the program.

...and RUN.

Answer: 80 FOR R=100 TO 1500 STEP 25

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. Try pressing the **RUN/STOP** key before anything scrolls off the top. (To restart after a STOP, either type RUN to start the program all over again, or CONT to continue execution at the

“break” point, or GOTO (line number) to start at a specific Line but retain the values of the variables.

2. Holding down the **CTRL** key slows the scrolling process. Go ahead and try it, but temporarily forget I told you. We’re going to pretend we don’t know about this feature for the rest of the Chapter so we can learn some other important techniques.
3. For a really classy display, we can build a “pause” into the program. The screen will fill, halt a moment, and automatically go on if we don’t interrupt execution.

The Timing Loop

It takes time to do everything. Even this clever box takes time to do its thing, 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 resident program. The new one must END without plowing ahead into the “LONDON1” program, thus, Line 9. Type:

```
4 PRINT "DON'T GO AWAY"  
5 FOR X = 1 TO 7000  
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 the “LONDON1” program, we wouldn’t have this space available to demonstrate the point.

How long did it take? Well, it did take time, didn’t it? About 10 seconds? The **64** can execute approximately 700 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.

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



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 21000

EXERCISE 12-1: Using the space in Lines 1 through 8, design 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. Let's pull a LIST to look at them. My, my -- they almost fill the screen. Wonder what would we do if the programs were a few Lines longer so they couldn't both fit on the screen at the same time?

Rather than wring our hands about the problem, let's add some dummy Lines and learn how. Add:

```
1000 REM
2000 REM
3000 REM
4000 REM
5000 REM
6000 REM
7000 REM
8000 REM
```

```
9000 REM
10000 REM
```

and LIST. Sure enough, the first Lines of the program are chopped off.

For Every Problem, A Solution

Try each of the following variations of the LIST command, and study the screen very carefully as each version does its thing:

LIST 50 (Lists only Line 50)

LIST -50 (Lists all Lines up through 50)

LIST 50- (Lists all Lines from 50 to end)

LIST 30-70 (Lists all Lines from 30 thru 70)

LIST 15-85 (Note that these numbers are not even in the program)

How's that for something to write home about?

Question: How would you look at the resident program only up through Line 9?

Answer: Type LIST -9 (Talk about a give-away!)

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 useful variation of RUN called:

```
RUN (line number)
```

...and as you might suspect, it is similar to LIST (line number). To RUN the program starting with Line 10, type:

```
RUN 10
```

...and that's just what happens.

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.

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 Lines 1-9 and 1000-10000.

Now:

```
LIST
```

to make sure we got 'em all.

Wrong Way Computer

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

```
85 IF R = 500 THEN STOP
```

...and RUN.

We know R is going to increment from 100 to 1500. 500 is about a third of the way to the end. See how the chart PRINTed out to 475 mph, then hit the STOP as 500 came racing down to Line 85? The screen displays the first third of the chart and:

```
BREAK IN 85
```


means the program is STOPped, or broken in Line 85. To restart the program merely type:

```
CONT      RETURN
```

...and execution picks up where it left off and PRINTs the rest of the chart, or until it hits another STOP. Where would you place the next STOP?

Yep.

```
87 IF R = 1000 THEN STOP
```

...and RUN.

At Last

The ultimate plan is to build timers into the program which don't STOP execution, but merely delay it for study.

For the first timer, add:

```
83 IF R<> 500 THEN 90  
84 FOR X = 1 TO 3500  
85 NEXT X
```

...and RUN.

Hey! It really works! As long as R does *not* equal 500 the program skips over the delay loop in Lines 84 and 85. When R *does* equal 500, the test "falls through" and Lines 84 and 85 "play catch" 3500 times, delaying the program's execution for about 5 seconds.

You have learned enough to design the 2nd delay at Line 1000 by yourself. Have fun.

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 time to prove your stuff.

EXERCISE 12-2: 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 12-3: Design, write and RUN a program which will calculate and PRINT income at yearly and monthly rates, based on a 1/12th-year month. Do this for yearly incomes between \$5,000 and \$20,000 in \$1,000 increments. Document your program with REM statements as necessary to explain the equations you create.

Some of our programs are becoming a little too long for us to leave space in the manual for you to write in your ideas. From now on, use a pad of paper for working up your answers.

EXERCISE 12-4: 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 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 12-5: 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 $\text{Area} = \text{Length times Width}$, or $A=L*W$.

Learned In Chapter 12

Commands

LIST (line number)
RUN (line number)
CONT

Statements

STOP

Miscellaneous

Timer Loop
"Brute Force" method
CTRL slows listing

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 4 different PRINT "zones".

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(15);"TOTAL";  
15 PRINT TAB(30);"SPENT"  
20 PRINT TAB(5);"BUDGET";  
25 PRINT TAB(15);"YEAR'S";TAB(30);"THIS"
```



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

```

30 PRINT TAB(5);"CATEGORY";
35 PRINT TAB(15);"BUDGET";
40 PRINT TAB(30);"MONTH"

```

The RUN should appear:

THE	TOTAL	SPENT
BUDGET	YEAR'S	THIS
CATEGORY	BUDGET	MONTH

EXERCISE 13-1: 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.

HINT: This isn't as easy as it looks, so it may require extensive editing. Since automatic zone formatting is not adjustable, the other formats will have to be keyed to it.

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

...and RUN.

Appears:

```

      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?

EXERCISE 13-2: Rework the answer to Exercise 12-3 to include the weekly rate of pay in the PRINTout. Use the TAB function to make the weekly amounts fit on the screen.

Whole numbers are most commonly used as TAB values, but on those rare occasions when a fraction is used, the Computer rounds the fraction up to a 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. (We already did it a number of times in the 2 programs above.)

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 H = 7
100 PRINT "A"TAB(5)"B"TAB(10)"C"TAB(15);
100 PRINT "D"TAB(20)"E"TAB(25)"F";
110 PRINT TAB(30)"G"TAB(35)"H"
120 PRINT A;TAB(5)B;TAB(10)C;TAB(15)D;
130 PRINT TAB(20)E;TAB(25)F;TAB(30)G;
140 PRINT TAB(35)H
```

The trailing semi-colons (;) in Lines 90, 100, 120 and 130 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.

SPC

There's one more PRINT-related function we should know. SPC is a contraction for SPaCe. SPC(number) works just like TAB(number) except instead of printing the number of spaces from the left margin, it prints the number of spaces from the cursor position. Try the following program:

```
10 PRINT TAB(10);"SEE    SPACE"  
20 PRINT SPC(10);"SEE";SPC(4);"SPACE"
```

Experiment with this on your own, too. (You might try switching some of those TABs from our last program and see what happens.)

Pretty slick stuff, huh?

Learned In Chapter 13

Print Modifiers

TAB
SPC

Miscellaneous

Trailing semi-colon

Chapter 14 ---

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.

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.

We can nest FOR-NEXT loops 9 levels deep.

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.

Line 30 establishes the second loop, called B, and directs that it be executed twice.

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) busi-

NOW HE'S
INTO NEST-
ED LOOPS



ness, let's play with the program. Change Line 10 to read:

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

...and RUN.

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 **CTRL** key to slow the display.

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 says:

```
? NEXT WITHOUT FOR ERROR IN 60
```

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, just be careful.

Breaking Out Of Loops

Improper nesting is illegal, but breaking out of a loop when a desired condition has been met is OK. Add and change these Lines:

```
50 NEXT B
55 IF A = 2 GOTO 100
60 NEXT A
99 END
100 PRINT "A EQUALLED 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 14-1: 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 14-2: Use the program which is the answer to Exercise 14-1. 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.

Learned In Chapter 14

Miscellaneous

Nested FOR-NEXT loops

Protective END blocks

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 people make a whole career of complicating simple ideas. We try to do just the opposite.

The INTeger function, $\text{INT}(X)$, allows us to “round off” any number, large or small, positive or negative, to 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 . $\text{INT}(X)$, on the other hand, gives us the **largest** whole number which is **less than or equal to** X . This is a very versatile form of rounding. In fact, we can use it to produce the other “ordinary” kind of rounding.

Type NEW 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 π .

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 powerful 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 this 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 = .141590001
```

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

What's that **1** doing on the end? Well, that's a rounding error -- meaning computers have only limited precision too. The 8th and 9th digits are not always to be trusted.

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

There *is* a way to control the accuracy of the display. 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 π accurate to only 3 decimal places. (Of course, we can specify it as 3.142, but that's not the point.) Type 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 the 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 for X (just be careful $X*1000$ 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, reLOAD and RUN INTEGER1.

```
10 X = 3.14159
20 Y = INT(X)
30 Z = X - Y
```

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

It split X into an INTEger and fractional part.

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 = .141590001
L = 1
```

We peeled off the leftmost digit in the decimal value of X. 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...)

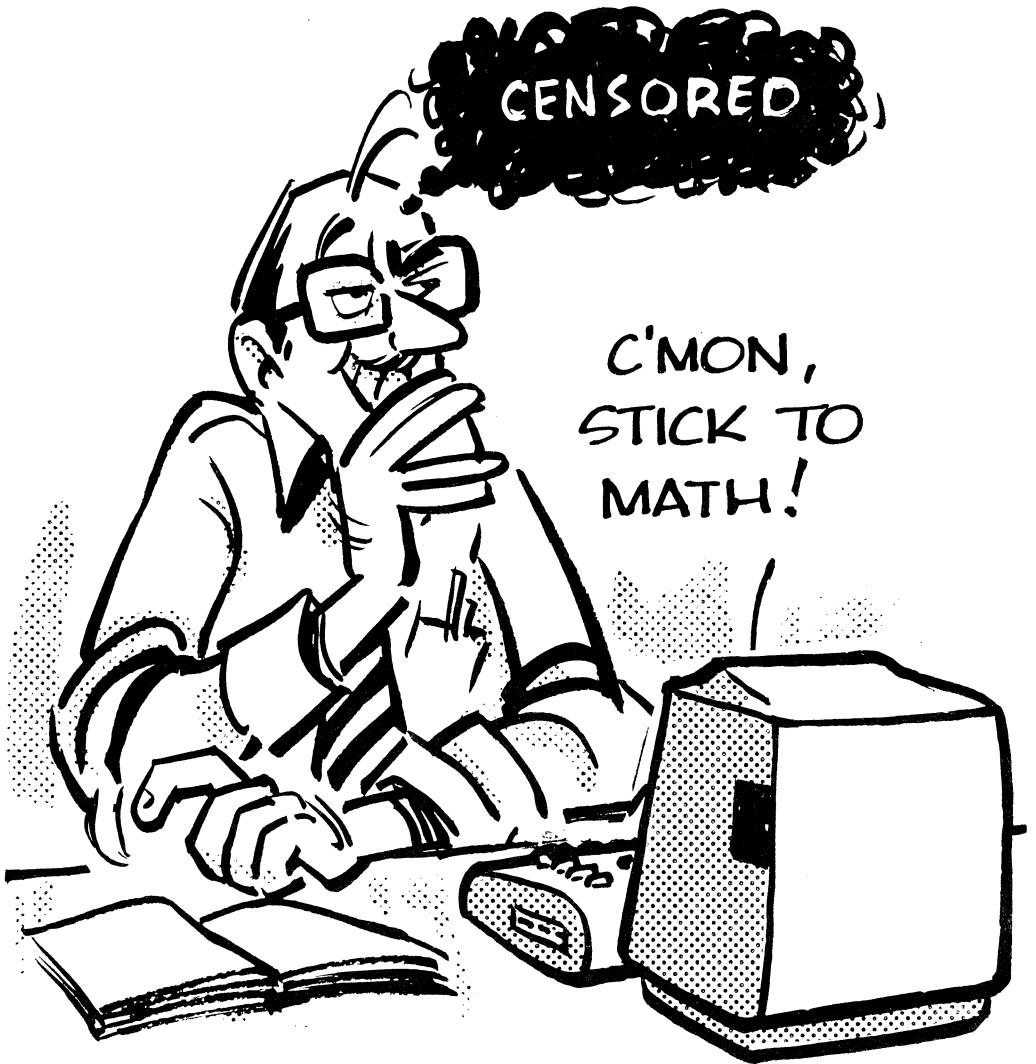


After all, these digits might not be just an accurate value of pi, but a coded message from a cereal box. If you don't have the decoder ring it's tough luck, Charlie -- unless you have a computer!

TIME OUT FOR
CREATIVE THINKING

CENSORED

C'MON,
STICK TO
MATH!





(...More recorded music...)



Enough thinking there on company time! Add these Lines:

```
75 FOR A = 1 TO 9
100 Z = Z - L
110 Z = Z * 10
120 NEXT A
```

...and RUN.

VOILA! The “PRINTout” reads:

```
X = 3.14159
Y = 3
Z = .141590001
L = 1
L = 4
L = 1
L = 5
L = 9
L = 0
L = 0
L = 0
L = 1
```

Line 75 began a FOR-NEXT loop with 9 passes, one for each of the 9 digits right of the decimal.

Line 100 creates a new decimal value of Z by stripping off the INTegeR part. (Plugging in the values, $Z = 1.41590001 - 1 = .41590001$)

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.

($Z = .41590001 * 10 = 4.1590001$)

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" technique is such a wonderful tool for troubleshooting stubborn programs that you will want to make a habit of using 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 Line 77 as follows:

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

...and RUN.

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

```
105 PRINT , "#105 Z =" ;Z  
115 PRINT , , "#115 Z =" ;Z
```

...and RUN.

There it is. All the data we can handle (and then some). By using the **RUN/STOP** key to temporarily halt execution, we can study the data at every step to understand how the program works (or doesn't work). Do it, typing CONT to continue the RUN. Understand this program and all its little lessons completely. When you are satisfied, go back and erase the "flags".

EXERCISE 15-1: Enter this straightforward NEW program for finding the area of a circle.

```
10 PRINT "RADIUS", "AREA"  
20 PRINT  
30 FOR R=1 TO 10
```



```

40 A = π * R * R
50 PRINT R,A
60 NEXT R

```

...and RUN.

Note, we use the π key in this exercise. We didn't use the value of the π key earlier in the chapter because it would not have demonstrated the INT function very well.

Area equals π (3.14159) 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. Without giving any big hints, modify the resident program to suppress all the numbers to the right of the decimal point.

EXERCISE 15-2: Now, knowing just enough to be dangerous, and in need of a lot of humility, change Line 45 so that each value of *area* is rounded (down) to be accurate to one decimal place. For example:

RADIUS	AREA
1	3.1

EXERCISE 15-3: Carrying the above Exercise one step further, modify the program Line 45 to round (down) the value of *area* to be accurate to 2 decimal places.

Learned In Chapter 15

Functions

INT(X)

Miscellaneous

Flags

More Branching Statements

It Went That-A-Way

Enter this NEW program:

```
10 INPUT "TYPE A # BETWEEN 1 AND 5";N
20 IF N = 1 GOTO 110
30 IF N = 2 GOTO 130
40 IF N = 3 GOTO 150
50 IF N = 4 GOTO 170
60 IF N = 5 GOTO 190
70 PRINT " THE # WAS NOT BETWEEN "
80 PRINT "1 AND 5 --- DUMMY!"
99 END
110 PRINT "N = 1"
120 END
130 PRINT "N = 2"
140 END
150 PRINT "N = 3"
160 END
```

```
170 PRINT "N = 4"  
180 END  
190 PRINT "N = 5"
```

RUN it a few times to feel comfortable and be sure it is “debugged”.

Debugged is an old Latin word which, freely translated, means “getting all the errors out of your 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 function called ON-GOTO which cuts out lots of Lines of programming. Let’s examine an ON-GOTO after you do the following:

Erase Lines 20, 30, 40, 50 and 60.

Enter this new Line:

```
20 ON N GOTO 110,130,150,170,190
```

...and RUN a few times, as before.

Works just the same, doesn’t it?

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

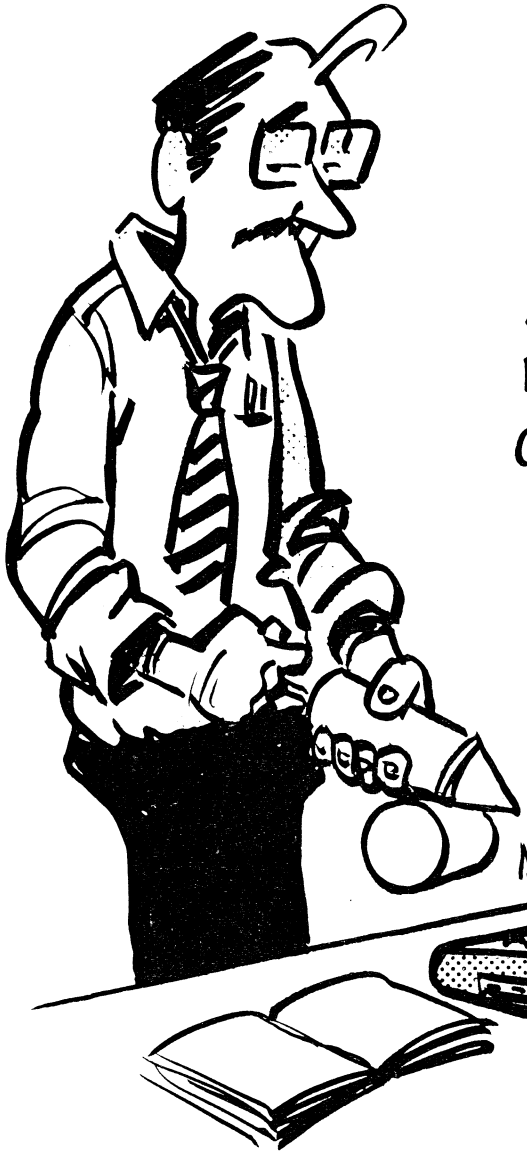
if the INTEGER value of N is 1 then GOTO Line 110.

if the INTEGER value of N is 2 then GOTO Line 130.

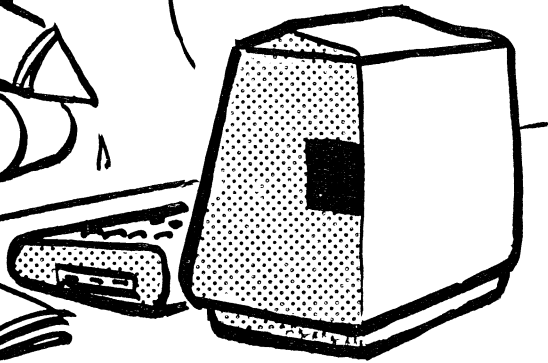
if the INTEGER value of N is 3 then GOTO Line 150.

if the INTEGER value of N is 4 then GOTO Line 170.

if the INTEGER value of N is 5 then GOTO Line 190.



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



if the INTEGER value of N is not one of the numbers LISTed above, then move on to the next Line.

Remember, an INTEGER is just a whole number.

The ON-GOTO statement has a built-in INT statement. It really acts like this:

```
20 ON INT(N)GOTO , , ETC ,
```

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

```
1.5
3.99999
0.999
5.999
6.0001
```

Get the picture?

Variations On A Theme

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

```
20 ON N GOTO 110,130,150,170,190
25 ON N-5 GOTO 210,230,250,270,290
30 ON N-10 GOTO 310,330,350,370,390
```

and fill in the proper responses at those Line numbers.

In Line 25, 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 30, since we had already provided for INPUTs between 1 and 10, we subtract 10 from INPUT N to cover the range from 11 through 15. By using the ON-GOTO statement, we have programmed into 3 Lines what would

otherwise have taken 15 Lines. By packing more branching options into each ON-GOTO Line, we could have done it in 2 Lines or less, depending on the number of digits in the Line numbers of the branch locations.

As in most of our examples, we could have used any letter after “ON”, not just N. As we just saw, N can be the value of a letter variable, or a complete expression, either calculated in place (as here) or calculated in a previous Line.

Trade Secret

Due to the vagaries of rounding error and the chance the error might just round a number like “N” a tad below the INTeger value expected, it is common to see something like this:

```
50 ON N+.2 GOTO 100,200,300 ETC.
```

The effect of this shifty move is to add just a “pinch” to the incoming value of N, knowing full well that the ON-GOTO statement contains its own INT function. If N happens to have been rounded down to say 1.98 (instead of the 2.000 expected), 0.2 will be added to it making $N = 1.98 + .2 = 2.18$ which the built-in INT will round down to the desired 2. Pretty sneaky. Values between .1 and .5 are often added to the N for this purpose in well-written programs.

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. BASIC stores many of them in a special place in memory ready for us to call up as needed.

As an example of how to create functions that are *not* included in our BASIC, we are going to use a five-Line subroutine instead of the "SGN" function to accomplish the same thing. (Even though COMMODORE 64 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 dummies, you know.)

"Scratch" the program now in the Computer by typing NEW, then -- very carefully, so as not to make any mistakes, type in the SGN subroutine:

```
3000 END
3010 REM INPUT X, OUTPUT T= -1,0, OR +1
3020 IF X < 0 THEN T = -1
3030 IF X = 0 THEN T = 0
3040 IF X > 0 THEN T = +1
3050 RETURN
```

"CALLING" A Subroutine -- (Sort of like calling hogs.)

To use a subroutine, use the GOSUB (line number) statement.

This statement directs the Computer to GO TO that Line Number, execute what it says there and in the Lines following, and when done RETURN back to the Line containing the GOSUB statement. We will use Line 20 here.

```
20 GOSUB 3010
```

A RETURN is always part of a subroutine, and ours is at Line 3050. We have reserved Line Number 3000 to hold a protective END block for all of our subroutines, so the Computer doesn't come crashing into them when it is done with the main program. Try taking it out when we're done and see what happens.

Getting Down To Business

Okay, now let's combine GOSUB and SGN (using a subroutine) to see what all this fuss is about. Add:

```
10 INPUT "TYPE ANY NUMBER";X
20 GOSUB 3010
30 ON T+2 GOTO 50,60,70
45 END
50 PRINT "THE NUMBER IS NEGATIVE."
55 END
60 PRINT "THE NUMBER IS ZERO."
65 END
70 PRINT "THE NUMBER IS POSITIVE."
```

...etc. (the subroutine is already typed in)...and RUN.

Try this same program using ON-GOSUB in Line 30. Remember, change Lines 55 and 65 to RETURN and add Line 75 RETURN.

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 Line 3010 by 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.

Lines 3010 through 3050 contain this rather simple SUBroutine.

Line 3050 holds RETURN, which sends control back to Line 20, which silently acknowledges the return and allows movement 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 60, and if it is +1 GOTO Line 70." By adding 2 to each of those values we have "matched" them up with the 1, 2, and 3 series which is built into the ON-GOTO.

Lines 45, 55, and 65 are routine protective blocks.

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

Preview Of Coming Attractions?

Like so much of what we are learning, this is just the tip of the iceberg. The ON-GOTO and SGN functions 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. Hmm...

Routines Vs Subroutines

We studied a special-purpose routine used as a SUBroutine. It is one of the few that we can both use and really understand. All the routines, understandable or not, can be built directly into any program instead of being set aside and "called" as subroutines. Their main value as 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, and do not need the GOSUB and RETURN statements.

One value of using special routines as SUBroutines is that some are exceedingly complex to type without error, and if each is typed once and SAVED on tape or disk, it can be quickly and accurately LOADED into the Computer as the first step in creating a new program.

We'll have more to say in a later Chapter. When you see just how powerful subroutines are, you'll feel like your 64 is even smarter than it thinks it is.

Now it's your turn.

EXERCISE 16-1: 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: $T = \text{SGN}(X)$

Learned In Chapter 16

Functions

SGN(X)

Statements

ON-GOTO
GOSUB
ON-GOSUB
RETURN

Miscellaneous

Debugging
Subroutines

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 typed between the parentheses or brought in as a variable from elsewhere in the program.

Type this NEW program:

```
40 FOR N = 1 TO 10  
50 PRINT RND(0)  
60 NEXT N
```

...and RUN. Did you observe:

1. 10 different numbers appeared?

2. All numbers were between 0 and 1?
3. Very small numbers were expressed in Exponential notation?

Change RND(0) to RND(10) and RUN again. More random numbers between 0 and 1.

Change Line 50 again and RUN several times using negative numbers, like:

```
50 PRINT RND(-20)
```

We get different sets of numbers -- but all with the same value. RUNNING again, the numbers are unchanged. Using a different negative number with RND produces the same result, but the value will change slightly.

Put a semi-colon behind the PRINT statement and increase the FOR-NEXT loop to 50 passes to put more numbers on the screen at one time.

```
40 FOR N = 1 TO 50
50 PRINT RND(0);
60 NEXT N
```

...and RUN.

This Is Fairly Exciting!

Well, maybe so, but we 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 cases we need a Random INTEGER, not a Random Number between 0-1. To create numbers larger than 1, we resort to mathematical chicanery.

Change Line 50 to read:

```
50 PRINT INT(RND(0) * 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:

$$R = \text{INT}(\text{RND}(\emptyset) * (B - A + 1) + A)$$

Where R = The RaNDom number

B = the *largest* INTEger

A = the *smallest* INTEger

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 again. Every toss is totally independent of what happened before it. *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 a complete 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 PRINT "SHIFT CLR/HOME"
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(0) * 2 + 1)
60 ON X GOTO 90,110
70 PRINT "BOMBED! NEITHER A 1 NOR A 2."
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 H/F*100; "%", T/F*100; "%", 100; "%"
```

...and RUN.

“Flip the coin” 100 times on the first RUN to get a feel for the program and the RUN time. RUN as many times as it takes to convince you that the random number generator produces really random numbers. Then try 1000 flips. When it’s time for lunch, or you can wait quite a while for the answer, try 25,000 flips or more.

Program Analysis:

Line 10 clears the screen.

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 of our 1’s and 2’s (heads and tails). Here’s what happens. Since the RND(0) gives us random numbers between 0 and 1 (non-inclusive) we multiply RND(0) by 2. Now we have random numbers between 0 and 2. There are two more steps, so don’t get bored yet! Next we add 1 to the value of RND(0)*2 and this puts our random numbers between 1 and 3 (remember, the actual range is 1.000...1 to 2.999...). Finally, we take the integer value of RND(0)*2+1. This gives us a 1 if the random number is in the interval of 1-2 and a 2 if the random number is in the interval of 2-3. And that gives us our heads or tails.

Line 60 has an ON-GOTO test, if X=1 we GOTO Line 90 where the “Heads” are counted, and if X=2 we GOTO Line 110 where the

“Tails” are counted.

Lines 70 and 80 are default Lines. If $X =$ other than 1 or 2, this error message will be PRINTed and execution will END. It will never happen, but this test helps prove the point.

Line 90 sets up H as a “Heads” counter. Each time the ON-GOTO test sends control to this Line (because $X = 1$), H is incremented by one.

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 moves on to Line 130. Until then, the NEXT N sends it back to Line 40.

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 120 passes control to Line 130 when the last “N” is “used up”.

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. Multiple generators have 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 ... would it?

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(0) * 6 + 1)
20 B = INT(RND(0) * 6 + 1)
30 N = A + B
40 PRINT N,
50 GOTO 10
```

...RUN.

As you can see, each number PRINTed falls between 2 and 12. We only need to PRINT N since the dice are always both thrown at the same time, and only the *sum* of the two is what counts.

Remember to press **RUN/STOP** and **RESTORE** to stop the Computer.

Why would the following be wrong? It creates numbers between 2 and 12.

```
10 PRINT INT(RND(0)*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 2-12. 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 he rolls a sum of 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 17-1: You already know far 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 a while to finish, but give it your best before we turn over to Section C (User’s Programs) under CRAPS for a sample solution. Good luck!

Learned In Chapter 17

Functions

RND(X)

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 from Line 10 to Line 25 and RUN. No change in the PRINTout, right?
2. Move the DATA Line from Line 25 to Line 10000. 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 and there are additional Line numbers available to add more DATA Lines as the need arises. 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

Y'KNOW SOMETHIN'
FRIEND? YOU'RE
NOT HALF AS
SCARY AS IN THE
BEGINNING.



with NEW and type in this program:

```
10 DATA 1,2,3,4,5
20 FOR N = 1 TO 5
30 READ A
40 PRINT A;
50 NEXT N
```

...and RUN.

That DATA Line is outside the loop. Now move it to Line 25 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 20 in the program to read:

```
20 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 screen shows.

```
1 2 3 4 5
? OUT OF DATA ERROR IN 30
```

Change Line 20 so the number of READs is *less* than the DATA available.

```
20 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 20 back to:

```
20 FOR N = 1 TO 5
```

and insert:

```
35 RESTORE
```

...and RUN.

Oh-oh! The screen PRINTs five 1's instead of 1 2 3 4 5. Can you figure out why?

Line 30 READ A as 1, but Line 35 immediately RESTORED the DATA Line to its *original unREAD 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. Type 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.

Change the program to read:

```
10 READ A$
20 DATA COMMODORE 64 COMPUTER
30 PRINT "SEE MY CLEVER ";A$
```

...and RUN.

SEE MY CLEVER COMMODORE 64 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 COMMODORE 64,COMPUTER
30 PRINT "SEE MY CLEVER ";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 CLEVER", 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 is A\$, and it prints "COMMODORE 64".

The third PRINT inserts the space which is enclosed in quotes.

The fourth PRINT is "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 18

Statements

READ
DATA
RESTORE

Miscellaneous

String Variables A\$, B\$, ...
Numeric Variables



PART 3
STRINGS

Intermediate BASIC

Intermediate Features Of COMMODORE 64 BASIC

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.

Smorgasbord



Multiple Statement Lines : (Now he tells us!)

BASIC allows us to put more than one consecutive statement on each numbered Line, separating them 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.

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 two letter variables are usually enough (this gives us $26 * 26$ variables), adding the numbers gives 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 may combine to make a grand total which we can just call D, and if that isn’t enough to solve all your problems, nothing will. Nearly a thousand possible variable names so far, and we’ll discover many more before we’re through.

$$B = 2$$

$$H = 4$$

$$A = .5 * B * H \qquad \text{Area} = .5 * \text{base} * \text{height}$$

In addition, we can use any combination of upper case letters and numbers for a name, up to 2 characters long. For example:

$$C4 = 19.95$$

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\$

OK\$

etc.

As with numeric variables, string variables can have any combination of up to 2 upper case letters and numbers followed by the \$ sign.

Shorthand

There are several little “shorthand” tricks available.

The first is the use of ? in place of the very common word, PRINT. Type NEW, then this Line:

```
10 ?"QUESTION MARK"
```

...and LIST it.

Awk! The pumpkin turned into a coach. The Computer rewrote it to read:

```
10 PRINT"QUESTION MARK"
```

It also works at the command level. Try:

```
?3*4          and we get:
```

```
12
```

The **RETURN** Key

If you're the very observant type, you noticed that program execution begins when the **RETURN** key is *pressed*, not when it's released.

Use Of Quotes & Semicolons

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. Leave it off at your own peril.

A BASIC interpreter that is "too forgiving" is like an airplane that is "too for-

giving.” 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, then **RETURN**, then **RUN/STOP** and **RESTORE**.

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
50 PRINT : LIST
```

See how smartly we can use LIST in a program so after the RUN it LISTs itself?
Great for learning and troubleshooting.

50 Characters per Line

The **64** permits up to 50 characters in a single program Line.

Learned In Chapter 19

Miscellaneous

- Multiple statement Lines
- Variable Names
- Some Shorthand
- Quotes and Semi-colons
- Multiple INPUTting
- Optional NEXT
- String Variables

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”. (No, it’s nothing like the “horsey 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 COMMODORE 64. Take a minute to review the chart.

Type in this short program:

```
10 FOR N=0 TO 255
20 IF N = 29 THEN PRINT CHR$(5)
30 IF N = 32 THEN PRINT CHR$(5)
40 IF N = 145 THEN PRINT CHR$(5)
50 IF N = 146 THEN PRINT CHR$(5)
60 IF N = 150 THEN PRINT CHR$(5)
70 IF N = 152 THEN PRINT CHR$(5)
80 PRINT "ASCII NUMBER";N;
```

```

90 PRINT "STANDS FOR ",CHR$(N)
100 FOR T=1 TO 500 : NEXT
110 NEXT N

```

...and RUN.

Observe that the characters between ASCII code numbers 32 and 90 are normal characters. The numbers from 0 to 31 and 91 to 255 are graphics and special characters -- refer to Appendix A and look at all the neat stuff they do.

Note: Lines 20-90 simply change the color to white so that the characters will be easy to read (otherwise we couldn't see where we were going...)

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.

Did you notice that character codes 192-223 are the same as the codes 96-127? Also, codes 224-254 are the same as codes 160-190 and code 255 is the same as code 126. Are you thinking what I'm thinking? Yes, these codes and any blank codes can be reprogrammed to create your own set of codes. If we contemplate the problems faced by the Japanese, Arabs, and others who need special letters and characters, it's easy to see how good use can be found for these ASCII values.

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 program:

```

10 INPUT "TYPE ANY NUMBER (0-255)";N
20 PRINT "THE CHARACTER IS ";CHR$(N)
30 PRINT : RUN

```

...and RUN.

See how RUN can be used *inside* a BASIC program?

EXERCISE 20-1: Using the ASCII chart (Appendix A) and the CHR\$ function, create a program which will PRINT the name: COMMODORE 64.

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:

```

1 REM * LICENSE PLATE GENERATOR *
10 PRINT "SHIFT CLR/HOME"
20 FOR N=1 TO 3
30 PRINT CHR$(INT(RND(0)*10)+48);
40 NEXT : PRINT " ";
50 FOR N=1 TO 3
60 PRINT CHR$(INT(RND(0)*26)+65);
70 NEXT : PRINT : GOTO 20

```

...and RUN.

The RND generator in Line 30 PRINTs numbers between 0 and 9 (achieved by accessing character codes 48-57). Line 60 spits out letters between A and Z (achieved by accessing character codes 65-90). Now look at the ASCII conversion chart for codes 48-57 and 65-90. Makes sense, doesn't it?

See if this sparks even more ideas (type NEW first):

```

10 PRINT "THIS IS "
20 PRINT CHR$(144);"BLACK OR ";
30 PRINT CHR$(5);"WHITE!"

```


NOW, WHAT'S ALL
THIS BABBLING ABOUT
"CHR\$(N) and ASC(\$)"



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*.

Type:

```
10 INPUT "TYPE NEARLY ANY CHARACTER";A$
20 PRINT "ITS ASCII NUMBER IS";ASC(A$)
30 PRINT : GOTO 10
```

...and RUN.

It will PRINT the ASCII number of almost all characters. Try lower case letters. Press **⌘** and the **SHIFT** key together. Now you are in lower case mode. Try "r" and then return to upper case, **⌘** **SHIFT**, and try "R". Same number, right?

Try the function keys and special control characters, too. You may have to consult the ASCII Chart in the Appendix to find out how to access these characters.

Some characters, like comma (,), space bar, cursor controls, etc., must be enclosed in quotes. Have fun!

An obscure way to use ASC is to imbed the character within quotes, thus:

```
PRINT ASC("A")
```

but this latter method is rarely convenient.

EXERCISE 20-2: 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 20

Functions

CHR\$
ASC

Miscellaneous

ASCII Codes
lower case
upper case

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:

```
1 PRINT "SHIFT CLR/HOME"  
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 21-1: 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 PRINT "SHIFT CLR/HOME"
20 READ A$,B$,C$
30 PRINT A$

```

SEE YOU'RE
INTO STRINGS
NOW, OL' BOY.



```
40 PRINT B$
50 PRINT C$
100 DATA COMPUSOFT, SAN DIEGO, CA, 92119
```

...and RUN.

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.

Using the Screen Editor, change Line 100 to read:

```
100 DATA COMPUSOFT, "SAN DIEGO, CA", 92119
```

...and RUN.

We will have to add them one at a time since the first one will turn Quote Mode on.

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.)

Let's make it read:

```
100 DATA "COMPUSOFT, "SAN DIEGO, CA", 92119"
```

...and RUN.

Awwk! Disaster. A Syntax error in Line 100? Yes, there is no straightforward 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 ERROR IN 20? 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 20 to just read one piece:

```
20 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.

As we continue to modify this program "over the cliff", that classic ballad from the hills is heard echoing:

*"Ah-cigareets, and whisky, and wild computers, they'll drive you
crazy, they'll drive you insane!"*

But, undaunted by this high class philosophy, we steer our vessel towards the next Chapter.

... as the sun sinks slowly in the west, warm breezes fill our sails and waves slap the bow. Stars twinkle, and around the beach fires plaintive native chants are heard, calling ...

Learned In Chapter 21

Miscellaneous

String comparison
INPUTting strings
READING strings

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 PRINT "TYPE A STRING OF CHARACTERS";
20 INPUT A$
30 L = LEN(A$)
40 PRINT A$;" HAS";L;"CHARACTERS"
90 PRINT : RUN
```

Try it several times, entering 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. Delete Line 20 and change Lines 10, 30 and 40 to read:

```
10 INPUT "ENTER A NUMBER";A
30 L = LEN(A)
```

```
40 PRINT A;" HAS";L;"CHARACTERS"
```

...and RUN.

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.

Concatenation

Concatenation? Concatenation??? Now what is that supposed to mean? Did you ever wonder who pays whom to sit around and think up such nondescriptive words? It must have been done on a government grant. Wait till Senator Proxmire hears about it.

Concatenation (pronounced con-cat-uh-na’tion) is a national debt-sized word which means ADD, as in “add strings together”. It’s easier to do than to pronounce.

CONCATENATION ?

CONCATENATION ??



FORGET THE
WEBSTER'S, PAL.
IT MEANS, "ADD
STRINGS TO-
GETHER"

Type this NEW program:

```
10 PRINT "SHIFT CLR/HOME" : FOR N = 1 TO 15
20 READ A$ : B$ = B$ + A$
30 PRINT B$ : NEXT N
100 DATA ALPHA,BRAVO,CHARLIE,DELTA
110 DATA ECHO,FOXTROT,GOLF,HOTEL
120 DATA INDIA,JULIETT,KILO,LIMA
130 DATA MIKE,NOVEMBER,OSCAR
```

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 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 22-1: 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 22-2: INPUT a word from the keyboard and compare it to a secret password. If there is a match, PRINT "CORRECT PASSWORD, ENTER". If not, PRINT "WRONG PASSWORD. GET LOST!". 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 22

Functions

LEN

Miscellaneous

Concatenation (+)

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:

```
1 PRINT "SHIFT CLR/HOME" : PRINT
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.

Now try RUNNING it some more with lots of different INPUTs, such as:

```
12345
ASDF
123ASD
ASD123
1,2,3
A,B,C
-456.823
6.23E27GARBAGE
```

and the same ones over again, but enclosed in quotes.

The screen tells all.

Using the Screen Editor, take the \$ out of Lines 10, 20, and 30 and RUN.

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! VAL's purpose remains very narrowly defined.

VAL ??
STR \$??
NOW HOW ABOUT
GIVING *ME* THAT
WEBSTER'S ?



Properly used, VAL converts a string holding a number into that number.

Remember this irritating frustration and narrow definition when you get in the thick of debugging a nasty string-loaded program.

STR\$

Now let's try the opposite, converting a *numeric* variable to a *string* variable. Change the program to read:

```

1 PRINT "SHIFT CLR/HOME" : PRINT
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 powerful two functions will come back manifold in future debugging efforts. VAL and STR\$ have very specific, but narrow abilities.

EXERCISE 23-1: 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 23-2: 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 23

Functions

VAL
STR\$

Having A Ball With String

L 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:

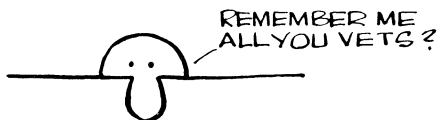
```
10 PRINT "SHIFT CLR/HOME" : PRINT
20 S$ = "KILROY WAS HERE"
50 PRINT LEFT$(S$,6),
60 PRINT MID$(S$,8,3),
70 PRINT RIGHT$(S$,4)
90 PRINT : LIST
```

...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 50 and 60 are to PRINT everything on the same Line.

Let's move some Lines around to exercise our new-found power. Move Line 60 to Line 40:

```
40 PRINT MID$(S$,8,3),
```

RUN ... and get:

```
WAS                KILROY                HERE
```

Now move Line 70 to Line 30 and add a trailing comma.

```
30 PRINT RIGHT$(S$,4),
```

RUN ... and get:

```
HERE                WAS                KILROY
```

These 3 functions can really do wonders with strings. Type in this NEW pro-

gram and examine each one in more detail:

```
10 PRINT "SHIFT CLR/HOME"  
20 S$ = "KILROY WAS HERE"  
30 FOR N = 1 TO 15  
40 PRINT "N =" ; N ,  
50 PRINT LEFT$(S$,N)  
90 NEXT : LIST
```

...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 30 to read:

```
30 FOR N = 1 TO 20
```

...and RUN.

Even though there are only 15 characters in the string, the overRUN is ignored. Change Line 30 back to N = 1 TO 15.

RIGHT\$ works the same way, but from the RIGHT:

Change Line 50 to read:

```
50 PRINT RIGHT$(S$,N)
```

...and RUN.

It's the mirror image of LEFT\$.

24 CHAPTERS
and ALL YOU CAN
DISPLAY IS:

"KILROY WAS
HERE"?



Now let's exercise MID\$ and see where it goes. Change Line 50 to:

```
50 PRINT MID$(S$,N,1)
```

...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 50 to:

```
50 PRINT MID$(S$,1,N)
```

...and RUN.

It PRINTed N characters, counting from number 1 on the left.

MID\$ can also simulate RIGHT\$. Change Line 50:

```
50 PRINT MID$(S$,16-N,N)
```

...and RUN.

Would you believe RIGHT\$ backwards, one at a time?

```
50 PRINT MID$(S$,16-N,1)
```

...and RUN.

How about a sort of "histogram" type graph:

```
50 PRINT MID$(S$,N,N)
```

...and RUN.

Make notes below 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 PRINT "SHIFT CLR/HOME"  
20 S$ = "KILROY WAS HERE"  
30 INPUT "CHARACTER # TO PRINT";N  
40 PRINT MID$(S$,N,1)  
90 PRINT : LIST
```

...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:

```
40 V$ = MID$(S$,N,1)  
50 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 24-1: 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 24-2: READ in the following part numbers: N106WT, A208FM, AND Z154DX. Use MID\$ to find the numbers. PRINT the number with the largest value.

INSTRING Routine

It would be of great value to be able to compare one string (or every string) against another (or all) string to see if they have anything in common -- this is sometimes called an INSTRING routine. We're going to build our own routine made up of LEN and MID\$ in order to learn them better.

Suppose we have a list of names and want to see if another certain name (or part of that name) is included 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 with 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 our NEW program by entering the list of names:

```
1000 DATA SMITH , JONES , FAHRQUART , BROWN
1010 DATA JOHNSON , SCHWARTZ , FINKELSTEIN
1020 DATA BAILEY , SNOOPY , JOE BFTSPLK , *
```

That was the easy part. We have to provide a means of READING these names, one at a time and comparing them, or parts of them, with the name or part of a name which we INPUT. Add these lines:

```
10 PRINT "SHIFT CLR/HOME"
20 INPUT "WHAT LETTER(S) IS WANTED " ; N$
30 PRINT
40 READ D$
50 IF D$ = "*" GOTO 2000
60 GOTO 100
70 PRINT , N$ ; " IS PART OF " ; D$
80 GOTO 40
2000 PRINT : PRINT "END OF SEARCH"
```

Now this takes a bit of explaining:

Line 10 clears the screen.

Line 20 INPUTs the name, or part of the name we are trying to locate, and PRINTs a blank space for easier reading to give this book some class.

Line 40 READs a single name from our DATA file.

Line 50 checks to see if we're at the end of the DATA file. If so, it says so and ENDS execution in Line 2000.

Line 60 shoots us to the INSTRING subroutine (covered next) which does all the sorting.

Line 70 PRINTs both what we're looking for and what we found.

Line 80 sends us back to READ another name from DATA.

That last part of the program isn't nearly as shaggy as the sort routine itself. Enter these final three lines:

```
100 FOR T = 1 TO LEN(D$) - LEN(N$) + 1
170 X$ = MID$(D$,T,LEN(N$))
180 IF N$ = X$ THEN 70
190 NEXT T : GOTO 40
```

Enter single characters, parts of names, and full names.

RUN it a few times to get the hang of what's going on, then we'll take it apart.

Line 100 starts off by setting up a FOR-NEXT loop. How far that loop continues depends on what number comes out of the difference between the length of D\$ (from the DATA line) and the length of N\$ (the name or part we entered). If that number comes out zero, we'll still go through the next three lines and then return to Line 40.

Line 170 evaluates the characters in D\$ and stores them into X\$. On the first loop, X\$ stores from the first character in D\$ to the "length of the inputted string" character. In our program, if JOH

was our inputted string (length is 3), then X\$ = SMI. Remember, D\$ = SMITH on the first pass. On the second loop, X\$ = MIT.

Line 180 compares X\$ to the inputted string. If they are equal, the statement in Line 70 is executed. Otherwise Line 190 is executed.

Line 190 causes the next loop to be executed. If there are no more loops, then the next data item is read in Line 40

Now that wasn't too bad, was it? ('Twarnt nothin', really.) A little time beside the pool reflecting on the logic will do wonders.

For those with only a silver fingerbowl, but no pool, this extra Line will show the inner machinations of the INSTRING routine.

```
70 PRINT N$;" IS CHARACTER#" ;T;"IN" ;D$
```

RUN it through a number of times, trying different letters. It really does make sense!

To see the effect the starting number (T) has on our INSTRING routine, change Line 100 to:

```
100 FOR T=2 TO LEN(D$) - LEN(N$) + 1
```

The INSTRING routine will now look at D\$ starting with the second character.

RUN and type in the letter S. Notice how it skipped SMITH, SCHWARTZ, and SNOOPY. Play around with the starting number in the INSTRING routine until you have a good handle on what it is doing.

EXERCISE 24-3: ReLOAD the INSTRING program and change the DATA Lines to:

```
1000 DATA P-RUTH, OF-MANTLE, SB-MORGAN
1010 DATA SS-LEOTHELIP, P-KOUFAX
1020 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

TIMES\$

Type the following:

```
PRINT TIME$
```

Hmmm, that's interesting. TIME\$ is a six-digit number which tells how long the Computer has been on in hours (first two digits), minutes (second two digits), and seconds (last two digits). It can also be used as a clock. To set the time, type:

```
TIME$ = "14:30:07"      RETURN
```

for 7 seconds past 2:30 in the afternoon (that's what my watch says). Wait a few minutes and type PRINT TIME\$ again. Pretty accurate clock, eh?

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 PRINT "SHIFT CLR/HOME" : N=0 : 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(38-N);B$
```

```

80 IF N=38 GOTO 100
90 N=N+1 : IF N<38 GOTO 120
100 L=L-1 : F=F+1 : IF L<0 THEN L=0
110 IF L=15 GOTO 20
120 PRINT "SHIFT CLR/HOME"
130 PRINT : PRINT : PRINT : GOTO 40
500 DATA "LUCKY LINDY HAS LANDED IN PARIS"
510 DATA "MET BY LARGE 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!

.....Fsssss!

Learned In Chapter 24

Functions

LEFT\$
MIDS
RIGHT\$
TIMES

Miscellaneous

INSTRING routine



PART 4
FUNCTIONS

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.

SQR(N)

The Square Root function is simple to use.

Type this:

```
10 INPUT "THE SQUARE ROOT OF";N
20 PRINT "IS";SQR(N)
30 PRINT : GOTO 10
```

...and RUN some familiar numbers.

Note if we try to take the Square Root of a negative number we will get an ?ILLEGAL QUANTITY ERROR.

Another way to find the square (or any) root of a number is by the use of the \wedge (up-arrow). 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 $\wedge$ (1/2)
```

...and RUN some familiar numbers.

The same logic which allows us to find the *square* root with the \wedge 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 $\wedge$ (1/21)
```

...and RUN.

Now there is real horsepower! Problem is, how are we 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 $\wedge$ (1/21)
30 PRINT "IS" ; R
40 PRINT
50 PRINT R;"TO THE 21ST POWER=" ; R $\wedge$ 21
60 PRINT : GOTO 10
```

...and RUN.

The INPUT and output numbers check out pretty closely, don't they? This

“proof” process might not stand up under rigorous scrutiny, but the answers are correct.

EXERCISE 25-1: 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 the 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?

LOG(N)

No, a LOG isn’t what they build cabins with, but even the swiftest among us have to refresh their 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 isn’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 2.71828183 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:

```

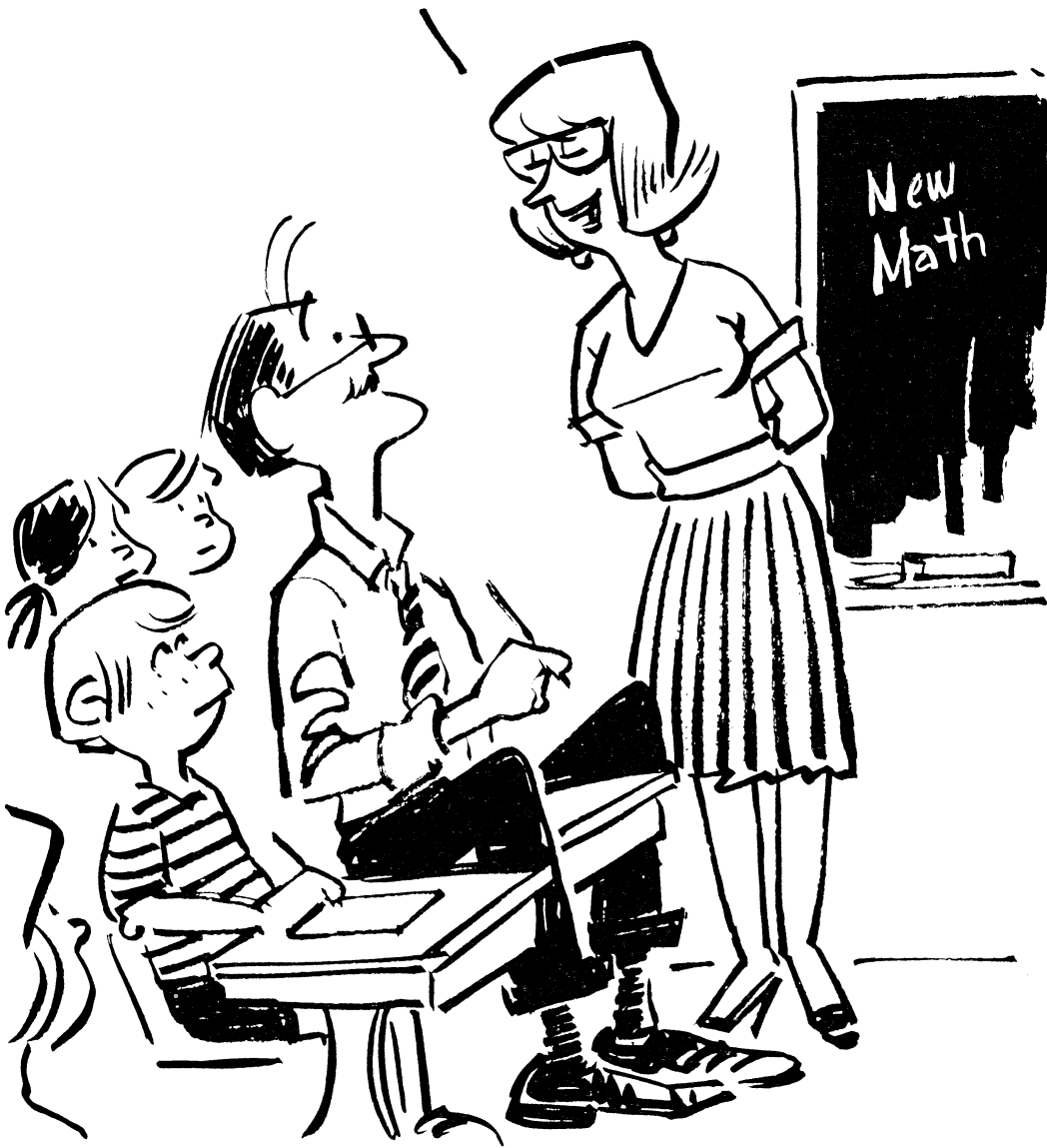
10 INPUT "ENTER ANY POSITIVE NUMBER";N
20 PRINT : L = LOG(N)
30 PRINT "THE LOG OF";N;
40 PRINT "TO THE NATURAL BASE"
50 PRINT "IS";L
60 PRINT : GOTO 10

```

The LOG function is not valid for negative numbers or zero.

...and RUN.

LET ME GUESS --
NEW COMPUTER?



Ummm Hmmm. Can't relate to the conclusion? Respond with the number 100 and you should get the answer 4.60517019. What that means is, 2.71828183 to the 4.60517019 power = 100. Lay that one on them at the next meeting of the Audubon Society and they'll know you're a strange duck.

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.

Decimal-based LOGs are called "common", or "base 10" Logs. Add these lines:

```
55 PRINT "THE LOG OF";N;
57 PRINT "TO THE BASE 10 =" ;L*.,434294482
```

...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 .434294482.

To convert a common log to a natural log, multiply the common log by 2.302585093.

And that's the name of that tune.

This final NEW program scoops it up and spreads it out:

```
10 REM * LOGARITHM DEMO *
20 PRINT "SHIFT CLR/HOME" : PRINT
30 INPUT "ENTER A POSITIVE NUMBER";N
40 PRINT
50 PRINT "THE NUMBER IS";N
60 PRINT "THE NATURAL LOG IS";LOG(N)
70 PRINT "THE COMMON LOG IS";
```

```
80 PRINT LOG(N)*.434294482
90 PRINT : GOTO 30
```

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.)

2.71828183 raised to the EXP power = the answer.

Type in this NEW program:

```
10 INPUT "ENTER A NUMBER";N
20 A = EXP(N)
30 PRINT "2.71828183 RAISED TO THE"
40 PRINT N;"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:

```
4.605170185
```

The BASE of the natural log system raised to this power should equal 100.

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 25-2: (For math fans only) Convince yourself that LOG and EXP functions are inverses of each other (hint: $\text{LOG}(\text{EXP}(N)) = N$). 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 25

Functions

INT
SQR
ABS
LOG
EXP

Miscellaneous

Natural Logs
Common Logs

The Trigonometric Functions

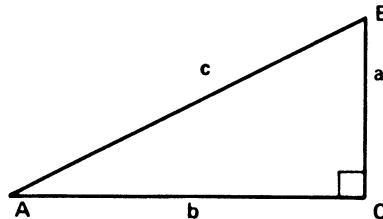
Since this is about as deep as we'll get into mathematics, I have to assume you know something about elementary trig.

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:

$$\text{SIN } A = a/c$$

$$\text{COS } A = b/c$$

$$\text{TAN } A = a/b$$



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"  
20 INPUT "BETWEEN 0 AND 90 DEGREES";A  
30 S = SIN(A*PI/180)  
40 PRINT "THE SIN OF A";A;"DEGREES IS";S
```

```
50 PRINT : GOTO 10
```

...and RUN, responding with any number between 0 and 90.

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 30 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 in Line 20 to radians. The SIN function will not work correctly without this conversion.

To convert angles from degrees to radians, multiply the degrees by $\pi/180$.

To convert angles from radians to degrees, multiply the radians by $180/\pi$.

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:

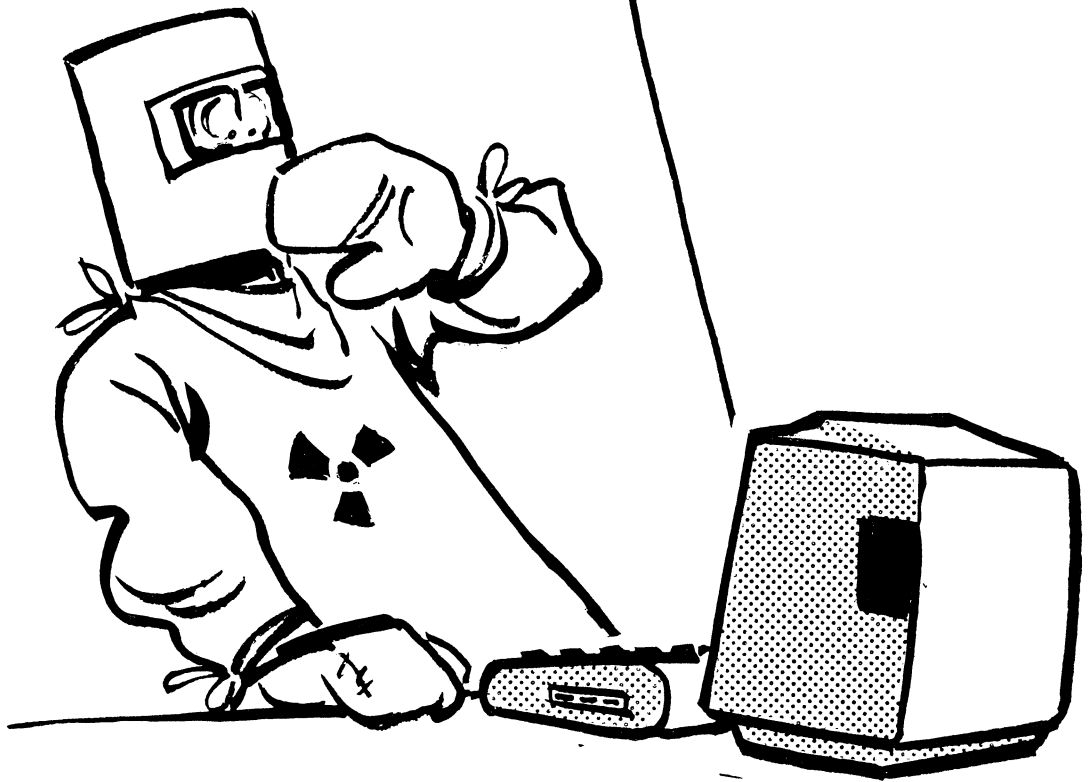
```
10 PRINT "ENTER AN ANGLE"
20 INPUT "BETWEEN 0 AND 90 DEGREES";A
30 C = COS(A* $\pi$ /180)
40 PRINT "THE COS OF A";A;"DEGREES IS";C
50 PRINT : GOTO 10
```

...and RUN.

Most answers are pretty close, very good for such a small computer.

We know that $\text{COS}(90^\circ)$ 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!

NO, NO, NO! RADIANS
HASN'T GOT ANYTHING
TO DO WITH ATOMIC
ENERGY!



For TANGent, RUN this program:

```

10 PRINT "ENTER AN ANGLE"
20 INPUT "BETWEEN 0 AND 90 DEGREES";A
30 T = TAN(A*π/180)
40 PRINT "TAN OF A";A;"DEGREES IS";T
50 PRINT : GOTO 10

```

The TAN function is not even defined for 90°, tho the COMMODORE 64 will *try* to calculate it.

This next NEW program displays all three major trig functions at the same time.

Note in Line 30 we divide our incoming angle by $(180/\pi)$ instead of multiplying it by $(\pi/180)$. The results are the same.

```

10 PRINT "SHIFT CLR/HOME"
20 PRINT "ENTER AN ANGLE"
30 INPUT "BETWEEN 0 AND 90 DEGREES";A
40 A = A/(180/π)
50 PRINT
60 PRINT "ANGLE =" ;A*180/π
70 PRINT "SIN =" ;SIN(A)
80 PRINT "COS =" ;COS(A)
90 PRINT "TAN =" ;TAN(A)

```

When we try to find the TANGent of 90 degrees, a “division by zero” error occurs because the Computer first finds the SIN of 90, which is 1, then divides into that the COS of 90, which is 0. 1/0 is a no-no as far as the Computer is concerned.

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 provision 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”, 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.

```

1 PRINT "SHIFT CLR/HOME" : PRINT
10 REM * ATN DEMO *
20 PRINT "ENTER ANY ANGLE"
30 INPUT "BETWEEN 0 AND 90 DEGREES":A
40 T = TAN(A*π/180) : PRINT
50 PRINT "TANGENT =" ; T : PRINT
60 I = ATN(T) * 180/π
70 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:

```

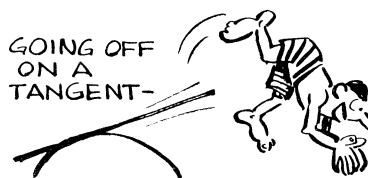
5 PRINT "SHIFT CLR/HOME"
10 REM INV FUNCTION DEMO PROGRAM
20 PRINT "ENTER A NUMBER - THE RATIO"
30 INPUT "OF 2 SIDES OF A TRIANGLE":R
40 AS=2*ATN(R/(1+SQR(ABS(1-R*R))))*180/π
50 AC = 90-AS : PRINT
60 PRINT "RATIO =" ; R

```

```
70 IF ABS(R)>1 THEN 110
80 PRINT "ARCSIN=";AS:PRINT"ARCCOS=";AC
90 PRINT "ARCTAN=";ATN(R)*180/π
100 PRINT : GOTO 20
110 PRINT "ARCSIN= U":PRINT"ARCCOS= U"
120 PRINT "ARCTAN="ATN(R)*180/π
130 PRINT : GOTO 20
```

Remember, when our ratio moves outside the range of -1 to 1 , ARCSIN and ARCCOS are both mathematically “Undefined.” Also, ARCSIN and ARCCOS produce angle measures between -90 and 90 degrees, but ARCCOS has a range between 0 and 180 degrees.

Other trig function routines can be found in *The BASIC Handbook*, available from CompuSoft Publishing and all good book and computer stores.



Learned In Chapter 26

Functions

SIN
COS
TAN
ATN

Miscellaneous

Degrees
Radians

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, COMMODORE 64 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 other factory built-in Functions, but, like subroutines, they greatly simplify BASIC programming.

The Format for defining our own Function is:

DEF FN name (variable) = formula

where:

name is the Function name, and

variable is a dummy variable that represents the value the Function will act on. *Name* and *variable* can be any valid variable names.

Formula is the expression where the calculations are carried out.

Let's say we want to write a Function to return RaNDom numbers, from 1

I DUNNO, YOU
MIGHT GIVE THE
CHAPTER A SHOT...



to the upper bound. Try this function program:

```
5 REM * UB = UPPER BOUND *
10 DEF FNR(UB) = INT(RND(0)*UB)+1
100 INPUT "GENERATE #'S UP TO: ";X
110 PRINT:FOR N=1 TO 50 : PRINT FNR(X);
115 NEXT
120 PRINT : PRINT : GOTO 100
```

and that's how easy that is ...

Learned In Chapter 27

Statement

DEF FN

PART 5
ARRAYS

Arrays

We know we can use combinations of the 26 letters of the alphabet and digits 0-9 to create variable names. 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. 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

COMMODORE 64 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're not sure you believe a single letter such as "A" can designate three different storage places? OK, in immediate mode 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.

```
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 type in:

```
30 FOR L = 1 TO 10
40 READ A(L)
50 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, type:

```
105 FOR N=1 TO 10
110 PRINT A(N)
120 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 can even do some simple arithmetic 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 that you know, just in case.

Let's work some more on the program.

Type:

```

10 PRINT "SHIFT CLR/HOME"
90 PRINT
100 PRINT "CAR#","ENGINE SIZE"
110 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, Delete Lines 105 and 120, and type:

```

20 INPUT "WHICH CAR'S ENGINE SIZE";W
110 PRINT W,A(W)
990 PRINT : LIST

```

...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,

HEH, HEH. ALL
I ASKED FOR
WAS ARRAYS.



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

In the language of professional computer types, this is called a *matrix*. A *matrix* is just an array that has more than one dimension. (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 inefficient 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 simpler 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:

```
60 FOR S = 101 TO 110
70 READ A(S)
80 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 "BAD SUBSCRIPT" 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.

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:

```
10 DIM A(110) : PRINT "SHIFT CLR/HOME"
```

...and RUN again. That's better, but it's not PRINTing the color code.

To display all the information, change these Lines:

```
20 INPUT "WHICH CAR'S ENGINE & COLOR";W
100 PRINT "CAR#","ENG. SIZE","COLOR"
110 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 28-1: 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.

```
20 DIM A(52) : PRINT "SHIFT CLR/HOME"  
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 READY 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(0)*52)+1  
60 PRINT A(V);  
70 NEXT N  
990 PRINT : LIST
```

...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 THEN 50
70 A(V) = 0 : P = P - 1
80 IF P <> 0 THEN 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?

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 28-2: 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 16
20 READ A$(N)
30 PRINT A$(N),
40 NEXT N
90 PRINT : LIST
100 DATA ALPHA,BRAVO,CHARLIE,DELTA
110 DATA ECHO,FOXTROT,GOLF,HOTEL
120 DATA INDIA,JULIETTE,KILO,LIMA
130 DATA MIKE,NOVEMBER,OSCAR,PAPA
```

...and RUN.

Oops. There's that same problem. "BAD SUBSCRIPT" 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$(16)
```

...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 CLR. Oh.

All CLR

The CLR 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 type:

```
CLR
```

and then:

```
PRINT A$(3)
```

Nothing. RUN the program again to reload the array, then PRINT A\$(3).

Array Names

```
A(N)
```

```
BC(N)
```

```
D3(N)
```

```
E4$(N)
```

```
XY$(N)
```

are examples of legal array names. The last 2 are for “string arrays.”

EXERCISE 28-3: Study the User programs in Section C to better understand the use of arrays for storage and access purposes. Time spent studying programs written by others is wisely invested.

Learned In Chapter 28

Statements

DIM
CLR

Miscellaneous

Arrays

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. Type in:

```
1000 DATA BRAVO ,XRAY ,ALPHA ,ZULU ,FOXTROT
1010 DATA TANGO ,HOTEL ,SIERRA ,MIKE ,DELTA
```

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 STRING SORT FROM DATA *
20 PRINT "SHIFT CLR/HOME"
30 FOR D=1 TO 10:READ A$(D):N=N+1:NEXT D
```

Line 10 is, of course, just the title.

Line 20 clears the screen, then:

Line 30 “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:

```

40 FOR F = 1 TO N-1 (F=First word to be compared)
50 FOR S = F+1 TO N (S=Second word to be compared)
100 NEXT S (Makes 9 passes)
110 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:

```

120 FOR D = 1 TO N:PRINT A$(D),:NEXT D

```

When the *sorting* 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.

```

60 IF A$(F) <= A$(S) THEN 100 (Tests for smaller ASCII#)
70 T$ = A$(F) (First word to Temp storage)
80 A$(F) = A$(S) (Copy Second word to First place)
90 A$(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 60 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 100, which will end this pass and READ another word to compare against F. If it is larger, drop to the next Line."

Line 70 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 80 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 90 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	DELTA	FOXTROT
HOTEL	MIKE	SIERRA	TANGO
XRAY	ZULU		

Printing will be in standard 10 space tab zone format.

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.

HEY, DOC, WHAT'S
2 PENS WITH BULLS
IN 'EM GOT TO DO
WITH "SEARCH and
SORT" ?



Add these temporaries:

```
55 PRINT F;A$(F),,S;A$(S)
57 FOR Z = 1 TO 500 : NEXT Z
65 PRINT "      <<--<< SWITCHEROO"
95 PRINT F;A$(F),,S;A$(S)
```

...and RUN.

Aw c'mon horse -- Whoa!

If that wasn't slow enough, change Line 57 so there is time for you to completely think it through. Pretend you're the Computer, and make the decision that Line 60 has to make. Take it from the top -- very slowly!

The Diagnosis

```
1 BRAVO                                2 XRAY
```

Means "in cell #1 is the word BRAVO. 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 *sorting* is going on is interesting. Add and change these Lines so they read:

```
56 PRINT TAB(30);"T$=";T$
57 PRINT TAB(30);"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, disk or tape. 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$
23 IF N$ = "END" THEN 40
25 N=N+1 : A$(N) = N$ : GOTO 20
```

Delete Lines 1000 and 1010.

...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 29-1: Change Line 60 of the *sort* program to list the names in reverse alphabetical order.

Learned In Chapter 29

Miscellaneous

Sorting

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 straightforward 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:

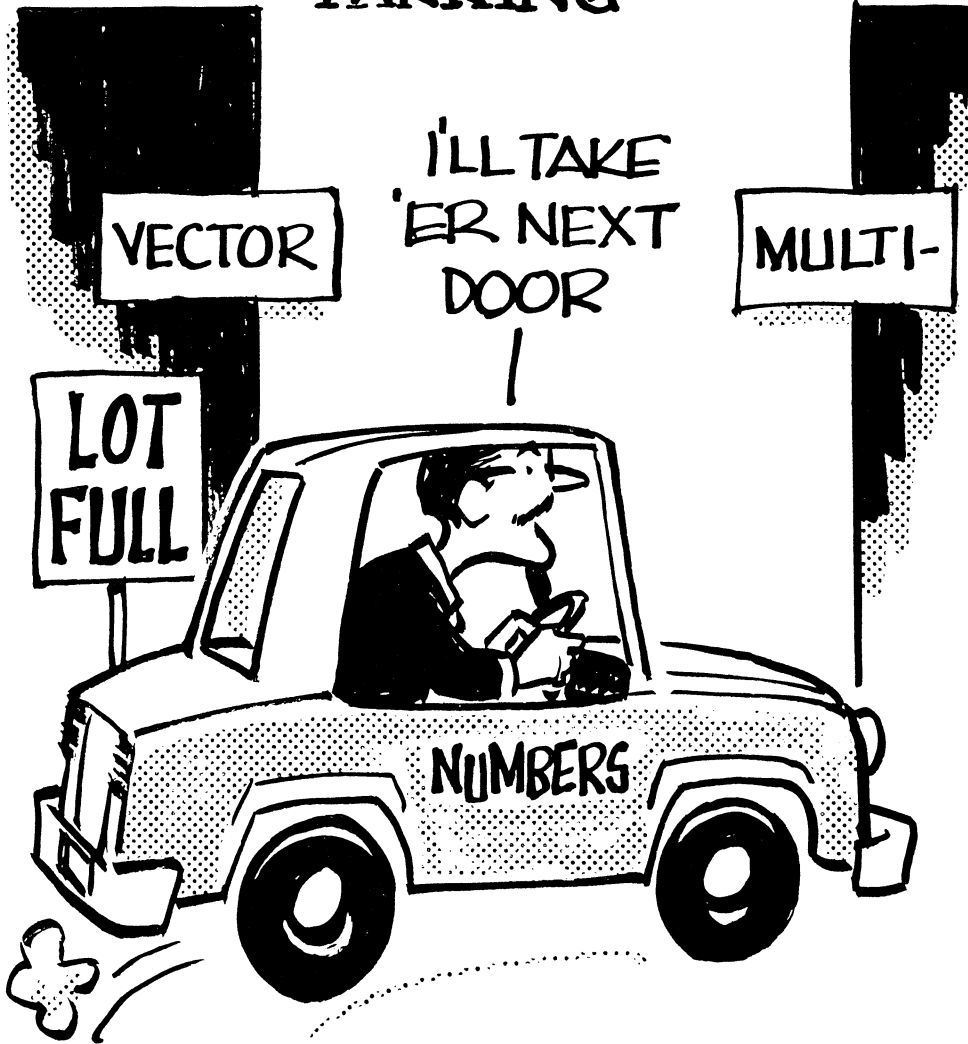
```
10 DIM M(40)
20 FOR V=1 TO 40
30 PRINT V,M(V)
40 NEXT V
```

Remember, any array with more than 11 elements (counting 0) must be DIMensioned.

...and RUN.

ARRAY'S

PARKING



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

```

...and RUN.

The *addresses* of all 40 cells are 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),
```

...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 1,2,3,4,5,6,7,8,9,10,11,12,13
105 DATA 14,15,16,17,18,19,20,21,22,23
110 DATA 24,25,26,27,28,29,30,31,32,33
115 DATA 34,35,36,37,38,39,40
```

and this Line to READ the DATA into matrix cells:

```
35 READ M(R,C)
```

...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 stay in the command mode for a minute and "poll" or "interrogate" several matrix positions and 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
```

Bad Subscript Error? 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
```

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.

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 on the screen.

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.

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 PRINT "SHIFT CLR/HOME" : DIM M$(6,4)
20 FOR R=1 TO 6
90 PRINT
100 DATA ALPHA,BRAVO,CHARLIE,DELTA,ECHO
110 DATA FOXTROT,GOLF,HOTEL,INDIA
120 DATA JULIETTE,KILO,LIMA,MIKE
130 DATA NOVEMBER,OSCAR,PAPA,QUEBEC
140 DATA ROMEO,SIERRA,TANGO,UNIFORM
150 DATA VICTOR,WHISKEY,XRAY
```

...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 tape or 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 07.0185,JONES,15
1020 DATA 07.0185,SMITH,87
1030 DATA 07.0185,BROWN,24
1040 DATA 07.0185,JOHNSON,53
1050 DATA 07.0185,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, Date 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, we 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). Add:

```
10 PRINT "SHIFT CLR/HOME" : PRINT
15 PRINT : PRINT "ENTRY #","DATE",
17 PRINT "NAME","AMT $" : PRINT
20 FOR E=1 TO 5 : PRINT E,
25 REM LOAD 5 ENTRIES
30 FOR D=1 TO 3
35 REM LOAD DATE,NAME,AMT
40 READ R$(E,D)
50 PRINT R$(E,D), :REM TEMP ARRAY PRINT
60 NEXT D
70 PRINT
80 NEXT E
```

...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, 1985.

Now that we know it loads OK, we can remove some of the test software. Change this Line:

```
20 FOR E=1 TO 5
```

Delete Lines 50 and 70

...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:

```
13 INPUT "WHOSE RECORD DO YOU NEED";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:

```
18 IF E > 0 THEN 110
110 FOR E=1 TO 5
120 IF R$(E,2) = N$ THEN 160
130 NEXT E
140 PRINT,N$; " IS NOT IN THE FILE."
150 PRINT : GOTO 13
160 PRINT E, R$(E,1),R$(E,2),R$(E,3)
170 PRINT : GOTO 13
```

...and RUN.

Answer with names that are in the DATA Lines, and those that are not. Lines 150 and 170 have built-in defaults back to the question.

The key Line is #160. 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

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 30-1: 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 30-2: Sort the names of the array in Exercise 30-1 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 30-3: If you survived Exercise 30-2, try sorting the array in increasing order by the numbers in Column 3.

Learned In Chapter 30

Miscellaneous

Multi-Dimension Arrays

String Arrays

Data Compression

PART 6

POKEING
AND COLOR
GRAPHICS

PEEK & 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 an expert 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).

COMMODORE 64 Memory Map

Decimal Address	Function
65535 - 57344	CBM 8K KERNAL Operating System
57343 - 56320	I/O Registers
56296 - 55296	Color RAM
55295 - 54272	SID Registers
53294 - 53248	VIC-II

Decimal Address	Function
53247 - 49152	Reserved RAM
49151 - 40960	8K CBM BASIC Interpreter
40959 - 2048	User RAM
2047 - 2040	SPRITE pointers
2039 - 1024	Screen memory
1023 - 2	Operating System scratchpad
0 & 1	6510 Registers

Figure 31-1

The Memory Map in Figure 31-1 shows that large chunks of the Computer's memory are reserved or "mapped" for very specific uses.

Turn the Computer off to clear out memory; wait a minute; turn it back on, 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. As Figure 31-1 shows, 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, the contents of which 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.)

HEE HEE HEE

HO HO HA

HEE HEE HEE
THAT TICKLES!

JES' POKIN',
BUDDY!



Okay, now RUN the program, being ready to slow it with **CTRL** if you see something interesting. It can also be stopped at any time with the **RUN/STOP** key, and restarted with CONT without having to start all over again with N at 0.

Change N to start at different places in memory and PEEK to your heart's delight (try starting at 40960 to see some of the innards of BASIC). If the color gets changed to the same as the background so you can't see anything, **RUN/STOP**, change the color back, and CONT. You can't goof up anything by just PEEKing. It's indiscriminate POKEing that gets you into trouble.

The command level is very handy for resetting the starting address. Change the value of N by just typing:

```
N=5000
```

for example, then:

```
CONT
```

instead of RUN.

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));
```

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 **RUN/STOP**, then:

```
PRINT N
```

in immediate mode 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.

Obviously, there is no use POKEing the ROM area since ROM stands for Read Only Memory. It's not changeable. Much of the "Memory mapped" area is reserved for specific things, so best not to POKE in there while we're just bungling around. Anything from 2048 to 40959 should be available memory, unless taken up with our BASIC program or required for processing. With such a short program as ours we surely can't goof anything up...can we?

Let's PEEK around 550 and see if anything is going on there. Change these two program lines to:

```
10 N = 550
20 PRINT N, PEEK(N)
```

...and RUN.

What we see are the address numbers and their contents, in easy-to-read parallel rows.

Great! Write a NEW program, POKE in some information and do something with it. Make it read:

```
10 REM * POKE PROGRAM *
20 N = 550
30 READ D
40 POKE N,D
50 N = N + 1
60 IF N = 560 THEN END
70 GOTO 30
100 DATA 80,69,69,75,45,65,45,66,79
110 DATA 79,33
```

Before RUNNING, let's analyze it.

Line 20 initializes the starting address at 550.

Line 30 READs a number from the DATA Line.

Line 40 POKEs the DATA "D" into address "N".

Line 50 increments the address number by one.

Line 60 ENDS execution when we have POKEd in all 11 pieces of DATA.

Line 70 sends us back for more DATA.

Lines 100 and 110 store 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=550 TO 560
220 PRINT N, PEEK(N)
230 NEXT N
```

...and RUN 200.

```
550 80
551 69
552 69
553 75
554 45
555 65
556 45
557 66
558 79
559 79
560 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?

Add:

```
205 PRINT "SHIFT CLR/HOME"  
220 PRINT CHR$(PEEK(N));
```

...and RUN 200.

That's how PEEK and POKE work.

Learned in Chapter 31

Statements

POKE

Functions

PEEK

Miscellaneous

Memory Map

Color with Commodore

The Commodore 64 has a wide variety of colors to add contrast and style to the display. To see these colors, it's necessary to connect the **64** to a color TV. However, we *can* use color statements on a black and white TV or a green monitor for some nice effects. You *do not* need a color TV or any other special hardware to use this Chapter. We can get different "shades" of black and white, or green, but not different colors.

With the TV hooked up and ready to go, type in the following program and RUN:

```
10 FOR I = 0 TO 15
20 POKE 53280, I
30 FOR N = 1 TO 500 : NEXT N
40 POKE 53281, I
50 FOR N = 1 TO 500 : NEXT N
60 NEXT I
```

Pretty neat. Press **RUN/STOP RESTORE** to return the screen to the normal display.

Line 20 POKEd the 16(0-15) border colors into the display. The format is:

```
POKE 53280, n
```

with n representing the color number.

Line 40 POKEd the same 16 colors into the background.

These two formats can also be used in the immediate mode. For example, type:

```
POKE 53280,14      RETURN
POKE 53281,0      RETURN
```

for a light blue border with a black background. The border and background colors are listed in Figure 32-1.

0	BLACK	8	ORANGE
1	WHITE	9	BROWN
2	RED	10	LIGHT RED
3	CYAN	11	GRAY 1
4	PURPLE	12	GRAY 2
5	GREEN	13	LIGHT GREEN
6	BLUE	14	LIGHT BLUE
7	YELLOW	15	GRAY 3

Coloring with **CTRL** and **C** Keys

Colors can also be displayed using the **CTRL** and **C** keys. Notice the colors listed under the number keys on the keyboard. These are the color combinations using the **CTRL** key (top left on the keyboard). Try pressing **CTRL** and the **3** key. Did the cursor change to red? Try other colors with **CTRL**.

Try the same color keys with the **C** key. Different colors, right? The **CTRL** key took us from Black to Yellow (keys 1-8) and the **C** key continued thru the colors from orange thru Gray 3 (keys 1-8 again). Note that these key numbers are not the same as the POKE color numbers. See the chart below for the different color alternatives.

Color with CTRL	Color with C
1 BLACK	1 ORANGE
2 WHITE	2 BROWN
3 RED	3 LT. RED

Color with CTRL	Color with C
4 CYAN	4 GRAY 1
5 PURPLE	5 GRAY 2
6 GREEN	6 LT. GREEN
7 BLUE	7 LT. BLUE
8 YELLOW	8 GRAY 3

Both the **CTRL** and **C** keys can also be used in a BASIC program to create color. Type in the following short program and RUN:

```

10 PRINT "DEFAULT COLOR IS LT. BLUE"
20 PRINT "<CTRL><2>WHITE <C><3>LT. RED";
30 PRINT "<C><7>BACK TO NORMAL"
40 LIST

```

Remember, **RUN/STOP RESTORE** will always return the normal display.

Reverse Mode with **CTRL**

To switch the background and cursor colors, in the immediate mode, type **CTRL 9** and a few characters (don't press **RETURN**). See the difference? Now press **CTRL 7** and a few more characters. We're back to the old standard display.

Add this line to our program:

```

5 PRINT "<CTRL><9>" ;

```

...and RUN.

Line 5 put the **64** in Reverse Mode and held it there with the trailing semi-colon. Line 10 PRINTed in reverse but the display went back to normal when the PRINT was encountered in Line 20.

Now let's have some real fun with color. The entire screen can be programmed to display any color. There are 1000 different positions on the screen and each of these can contain a character and a color.

WITHOUT NUMBERS,
..AMAZING!

EAT YOUR HEART
OUT, PICASSO!



Screen Memory Map

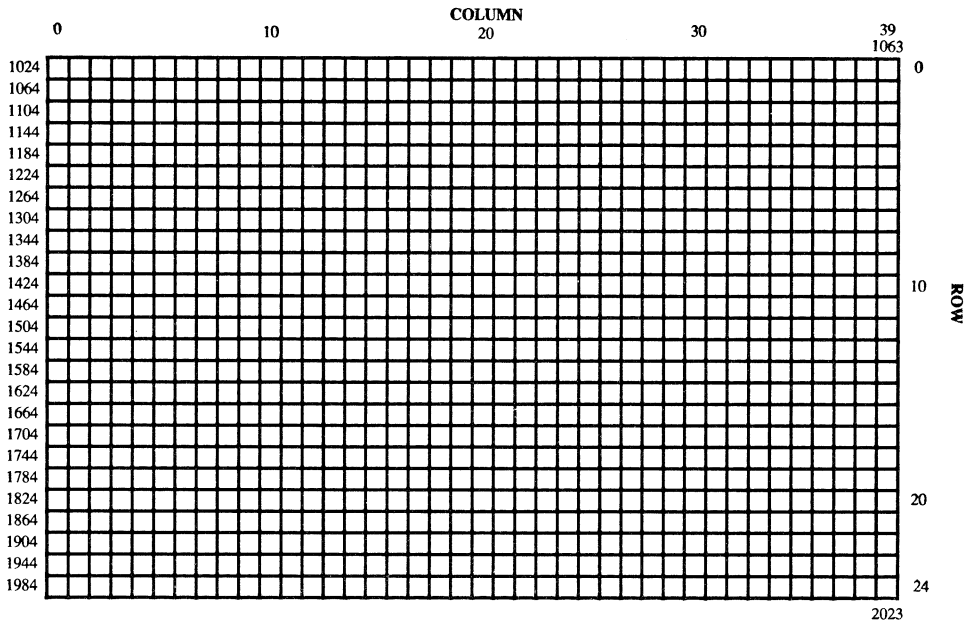


Figure 32-2

The screen memory map in Figure 32-2 outlines the different memory addresses for POKEing information to the screen. The top left position is 1024, and the top right position is 1063. This makes sense since we have 40 columns and 25 rows. Row numbers then will increment by 40. Row 2, column 0, will be position 1064.

Let's print a phrase down the middle of the screen, inserting characters at column 20 in each row. The first screen memory address is $1024 + 20 = 1044$. For each subsequent address we simply add 40 to this number. Sounds like a loop. Type in this NEW program:

```

10 PRINT "SHIFT CLR/HOME"
20 FOR I = 0 TO 920 STEP 40
30 READ A
40 POKE 1044 + I, A
60 NEXT I
70 DATA 3,15,12,15,18,19,32,23,9,20,8,32
80 DATA 3,15,13,13,15,4,15,18,5,32,54,52

```

The value READ in Line 20 is the character code that is POKEd in the screen in Line 30. These numbers are called screen codes and a complete listing can be found in Appendix E.

RUN the program. Users with color TV sets may not see anything appear on the screen. The characters are displayed in the same color as the background.

To change the color of each character from the background color, we need to look at another Memory Map.

Color Memory Map

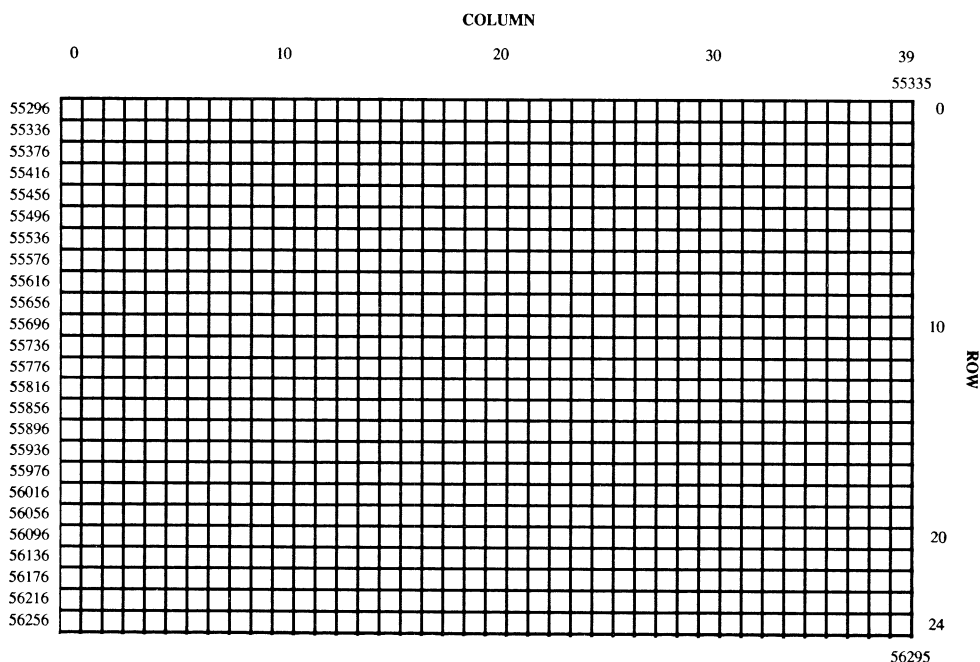


Figure 32-3

This map corresponds with the screen memory map. The top left position is 55296, which corresponds with position 1024 on the screen memory map. Therefore, if we POKE a color into address 55296, and a character into address 1024, the character will appear in that color. Add the following line to our resident program:

```
50 POKE 55316 + I, 1
```

Now type **SHIFT CLR/HOME** and RUN. Our statement should be displayed in all white letters.

Color via CHR\$

A third way to access color is with the CHR\$ function. This concept was introduced earlier when we printed a word in black and white. Now we can do it in color. Try the following NEW program:

```
10 PRINT "THIS IS ";
20 PRINT CHR$(28); "RED "; CHR$(30);
30 PRINT "GREEN "; CHR$(31); "BLUE"
40 PRINT CHR$(156); "PURPLE ";
50 PRINT CHR$(158); "YELLOW ";
60 PRINT CHR$(159); "AND CYAN,"
```

Play around with these different techniques until you've mastered the art of coloring with the Computer.

Learned in Chapter 32

Miscellaneous

CTRL key

C key

RESTORE

Screen Memory Map

Color Memory Map

Graphics with the 64

In the last Chapter we had a chance to use the Computer's color feature. We will expand on it in this chapter plus provide an introduction to graphic symbols. Graphic symbols on the **64** can be done in immediate mode, or included in BASIC programs for special effects. If you're ready, onward!

Keyboard Graphic Symbols

Look at the graphic symbols on the front of various keys. These are the graphic symbols which can be accessed in immediate mode in conjunction with the **G** and **SHIFT** keys. When the **G** key is used, the symbol on the left side of the key is displayed. When the **SHIFT** key is used, the symbol on the right is displayed.

Try **G** + . Looks like a checkerboard, doesn't it? Now try **SHIFT Z** for a diamond. Try other combinations, using both **G** and **SHIFT**. By putting several of these symbols together we could develop a pretty nice graphics design. Hmmm?

Graphic Symbols in BASIC

Not only are these graphic symbols on the keyboard, the **64** can also access them with **CHR\$**. Try typing:

```
CHR$(166)
```

```
RETURN
```


and

```
CHR$(122)      RETURN
```

The same graphic symbols are displayed as found on the keyboard. For a complete listing, see the CHR\$ chart in Appendix A.

Keep the CHR\$ chart handy because we are going to use it extensively. Let's design a card hand of four aces to illustrate the 64's graphic capabilities. Type in the following program:

```
5 REM * GRAPHICS PROGRAM *
10 PRINT "SHIFT CLR/HOME"
20 FOR A = 3 TO 30 STEP 9
30 PRINT TAB(A); CHR$(111);
40 PRINT CHR$(183);CHR$(183);CHR$(183);
50 PRINT CHR$(183);CHR$(183);CHR$(112);
60 NEXT A : Z = 1 : PRINT
```

...and RUN.

The CHR\$ symbols in Line 30 thru 50 build the top part of the cards. The 4 TAB positions are established with the FOR-NEXT Loop in Lines 20 and 60. Line 60 also sets the value of Z at 1 for use later in the program. Compare the CHR\$ codes in the program with those in the chart. Add the following Lines to the resident program:

```
70 FOR B = 1 TO 7
80 FOR C = 3 TO 30 STEP 9
100 X = 160
110 Y = 160
130 PRINT TAB(C); CHR$(180);
140 PRINT CHR$(Y);CHR$(160);CHR$(X);
150 PRINT CHR$(160);CHR$(160);
160 PRINT CHR$(167);
```

```
170 NEXT C : PRINT : NEXT B
```

...and RUN.

Notice that two CHR\$ statements in Line 140 use variables. These variables will be used to insert an "A" and the card suit symbol on each card. But first we need to complete the border of the cards. Add the following loop to the resident program:

```
200 FOR D = 3 TO 30 : STEP 9
210 PRINT TAB(D); CHR$(108);
220 PRINT CHR$(175); CHR$(175);
230 PRINT CHR$(175); CHR$(175);
240 PRINT CHR$(175); CHR$(186);
250 NEXT D : PRINT : END
```

It's getting better. Now we need to insert the different suits and the letter A on each card. Enter the next Lines to get our four-of-a-kind in aces.

```
90 IF B=4 THEN ON Z GOTO 300,320,340,360
120 IF B = 1 THEN Y = 65
300 REM FIRST CARD IS A SPADE
310 X = 97 : Z = Z + 1 : GOTO 130
320 REM SECOND CARD IS A HEART
330 X = 115 : Z = Z + 1 : GOTO 130
340 REM THIRD CARD IS A CLUB
350 X = 120 : Z = Z + 1 : GOTO 130
360 REM FOURTH CARD IS A DIAMOND
370 X = 122 : Z = Z + 1 : GOTO 130
```

SAVE this program as "CARDS". This is just the start of an interesting program.



WHAT SAY
WE GIVE IT
ANOTHER SHOT,
LEONARDO?



For those of you that would like to include the upside down “A” at the bottom right corner of each card, try changing the last CHR\$(175) in the D loop to a variable. Then set up a subroutine using the bit mapping to create your own upside-down A. Good luck!

Adding Color to the Cards

Let’s color the “A” in the upper left corner of each card. Later we will color the suit symbols, too.

Type in the following Line to the program and RUN:

```
135 IF Y = 65 THEN 400
145 PRINT "♣ 7" ;
400 IF C = 3 THEN 440
410 IF C = 12 THEN 450
420 IF C = 21 THEN 460
430 IF C = 30 THEN 470
440 PRINT CHR$(144) ;: GOTO 140
450 PRINT CHR$(28) ;: GOTO 140
460 PRINT CHR$(144) ;: GOTO 140
470 PRINT CHR$(28) ;: GOTO 140
```

Lines 440 to 470 alternately switch the color to Black(144) and Red(28) before the Letter “A” is displayed. Now that wasn’t too difficult. The final part is a piece of cake. We can use the same color statements that were used for the letter “A” to make each suit their respective color. Add the following Line and our program will be complete:

```
137 ON Z - 1 GOTO 440,450,460,470
```

...and RUN.

SAVE this program as “ACES”.

The program is designed so card suits, colors and the card values can be changed with relative ease. Have fun with the program!

Learned in Chapter 33

Miscellaneous

Graphics symbols

What's That Sound?

Most computers have a simple SOUND command but the **64** has a full range synthesizer. It toots like a trumpet and plays like a piano. The next time you need a one-man band, hire a programmer with a Commodore 64.

The **64** uses a small microprocessor chip to generate the sounds. This chip is controlled by POKEing various values into memory addresses that communicate with this chip. To gain full control of this chip requires a greater level of microprocessor theory than is intended by this book. What we can do is write a short program that illustrates the type of control we have over this sound generator.

Let's start by writing a program to play a middle C note. Type in this program:

```
10 REM LOOP CLEARINGS SOUND ADDRESSES
20 FOR C=54272 TO 54296:POKE C,0:NEXT C
```

Line 20 POKEs the value 0 in all addresses that are used to generate sound (54272 - 54296). This ensures that all stray values have been removed.

Next, we need to set the variables for controlling the note's volume, attack/decay, sustain/release, and waveform. Since these functions are to remain constant, their associated variables need to be set once. However, they can be changed later as needed within the program. Each of these variables will be explained later.

A single note is produced by POKEing a combination of two values into addresses:

```
54273   HI
54272   LOW
```

Look in Appendix F for the values that are POKEd into these HI and LOW addresses. Keep in mind that we must first establish the notes volume, attack/decay, sustain/release and waveform before we can actually hear something.

Add these Lines to the program:

```
30 IF L = -1 THEN END
40 REM V = VOLUME, A = ATTACK/DECAY
50 REM S = SUSTAIN/RELEASE, W = WAVEFORM
60 REM HF = HIGH FREQ., LF = LOW FREQ.
70 V = 54296 : A = 54277 : S = 54278
80 W = 54276 : HF = 54273 : LF = 54272
90 POKE V,15 : POKE A,136 : POKE S,136
200 POKE HF,8 : POKE LF,97
210 POKE W,17 : FOR T = 1 TO 500 : NEXT
220 POKE W,16
230 L = -1 : GOTO 20
```

...and RUN.

All that for just one note? Look at two values that are POKEd into the High and Low Frequency addresses in Line 200. Now find these values under the HI and LOW columns in Appendix F. These are the values for note C-3.

Now let's add more notes to the program. We'll use the combination of a loop and data statements to play a middle C scale. Make these changes and RUN:

```
100 FOR H = 8 TO 17
110 READ L
```

```

120 IF L = -1 THEN 20
200 POKE HF,H : POKE LF,L
230 NEXT H
500 DATA 97,104,143,48,143,24,210,195,-1

```

It sounds terrible because we're going up the musical scale without regard to musical "intervals". We must choose the notes more "chromatically". Look at the notes in the DATA Line and match them up with the numbers 8 - 17 used in the H loop. During the first loop, H is set at 8 and 97 is READ into L, so the first note matches up with C-3. During the second loop H is set at 9 and 104 is READ into L to give us D-3. Get the drift of all this?

Notice the only variables we changed were H and L. The duration is controlled by the loop in Line 210. A half note is about 250 loops. This value may vary depending on the length of the program. The chart below lists approximate values for notes.

Note	Duration
sixteenth note	31
eighth note	63
quarter note	125
half note	250
whole note	500

Try other combinations of high and low frequency values to create a little musical piece.

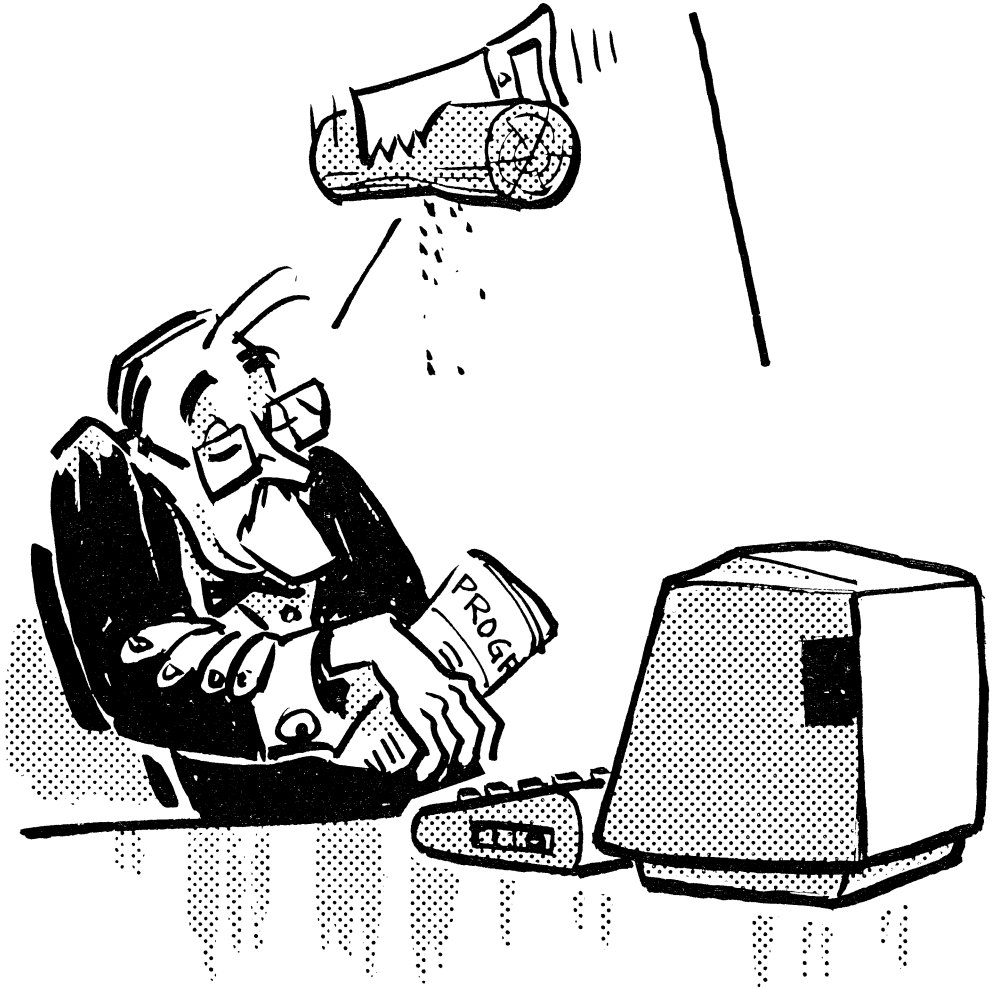
Attack/Decay

The Attack/Decay setting is similar to setting the crescendo and decrescendo in music. Attack is the rate at which a note or sound rises to its maximum volume. Decay is the rate at which it falls from the maximum volume to the sustain level. There are 4 levels of attack and decay.

Attack/Decay Rate Settings

	Attacks				Decays			
Values:	High	Med.	Low	Lowest	High	Med.	Low	Lowest
	128	64	32	16	8	4	2	1

NOW, FOR MY
NEXT NUMBER...



By adding any combination of these values we can arrive at each level. In our scale program we added a high attack value (128) with a high decay value (8) to get the setting (136). To achieve a maximum decay rate (decrecendo) we would add all the decay values ($8 + 4 + 2 + 1 = 15$).

Sustain/Release

The Sustain level establishes the duration of the note at a certain level while the Release level is the rate at which the note or sound falls from the sustain level to no volume. The Sustain/Release feature is very similar to Attack/Decay and is achieved in the same manner. The Sustain/Release chart below is the same as the Attack/Decay chart except for the POKE values.

Try changing the Attack/Release and Sustain/Release values in Line 90 to hear the effects of the different values.

Sustain/Release Rate Settings

	Sustain				Release			
	High	Med.	Low	Lowest	High	Med.	Low	Lowest
Values:	128	64	32	16	8	4	2	1

Waveform

The waveform value controls the type of sound the **64** generates. There are 4 different waveforms: triangle, sawtooth, pulse, and noise. The waveform must be activated before any note can be played.

Notice that there are two values for each type of wave. The first value starts that waveform and the second value turns it off.

Waveform Settings

	Triangle		Sawtooth		Pulse		Noise	
Values:	17	16	33	32	65	64	129	128

In the Waveform start value in Line 130 and the stop value in Line 220 to hear the various waveform shapes listed above.

Different Voices

The POKE values used to this point have been for voice 1. If harmony is needed, the same notes can be generated using voices 2 and 3. All three voices can be played together or alone. For a different sound in voice 2, select the POKE addresses for voice 2 and the appropriate sounds.

With this feature it is possible, for example, to let voice 1 play ROW, ROW, ROW YOUR BOAT like a piano, and then voice 2 could come in later and play the same tune sounding like a harpsichord.

The only difference between the three voices is the memory addresses that are POKEd. Look at the charts in Appendix G for the various combinations. Experiment with the various settings. After all, it takes practice to become a Commodore 64 synthesizer expert.

Learned in Chapter 34

Miscellaneous

Sound Registers

Volume Setting

Attack/Decay Setting

Sustain/Release Setting

Waveform Setting

High and Low Frequency Values

Note Duration

Voice 1, Voice 2, Voice 3



PART 7
MISCELLANEOUS

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 35-1, 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.

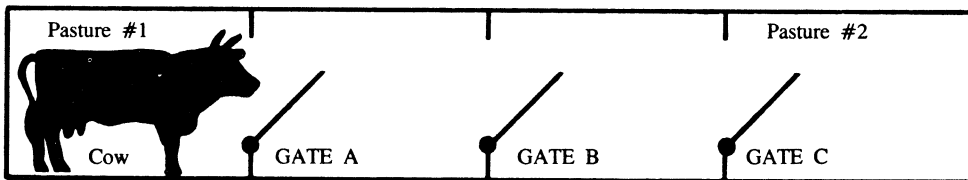


Figure 35-1

The principle is called “logical AND”.

In Figure 35-2, 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!

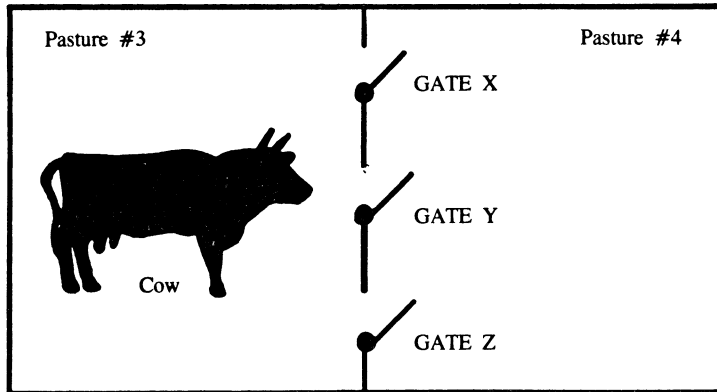


Figure 35-2

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
60 PRINT "OLD BESSIE IS SECURE."
70 END
80 PRINT "ALL GATES ARE OPEN."
90 PRINT "OLD BESSIE IS FREE TO ROAM."

```

...and RUN.

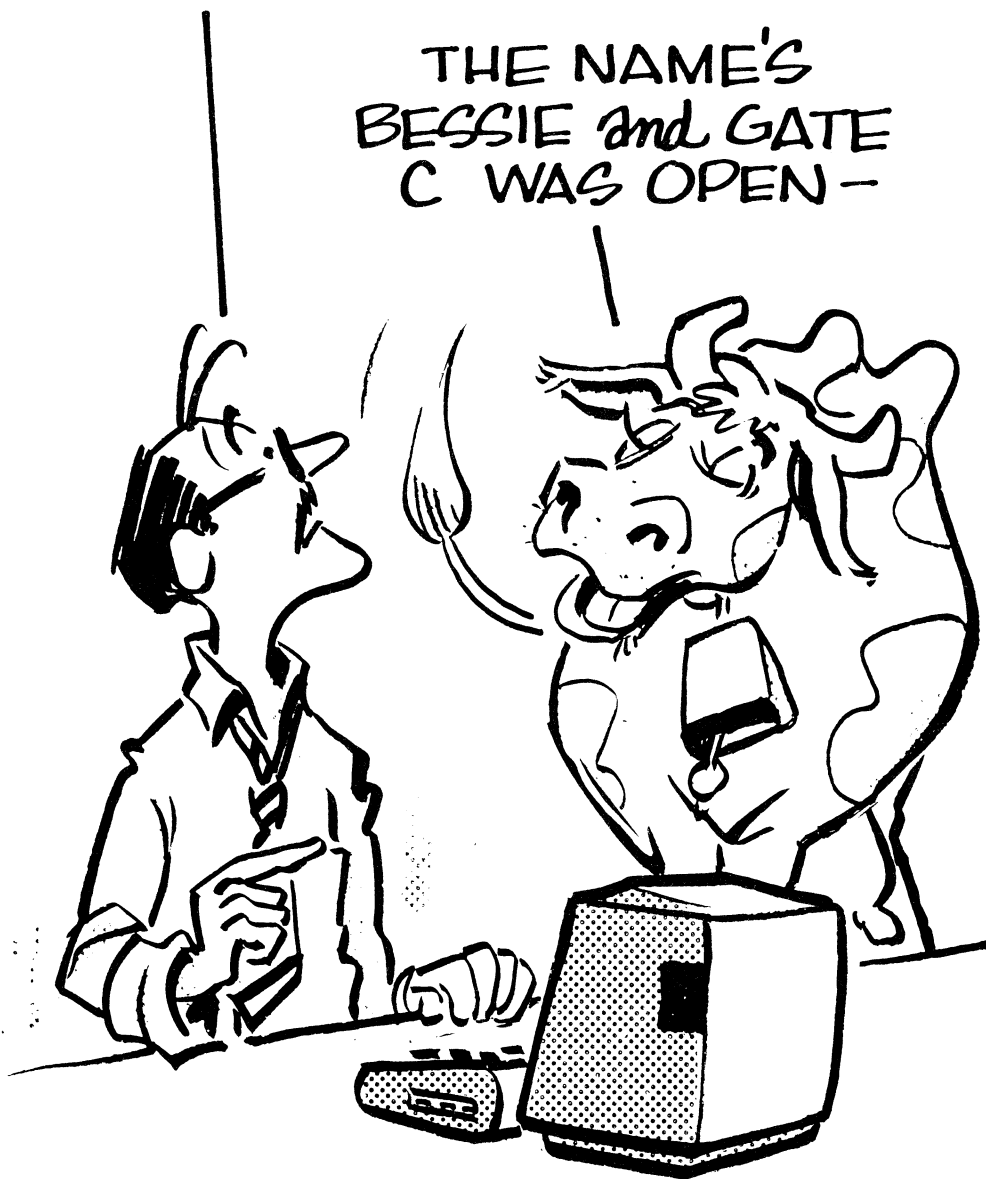
Answer (Y/N) the questions differently during different RUNs to see how the logical AND works in Line 50.

Where Is the Logic in All This?

You should by now understand every part in the program, except perhaps Line 50.

JEEPERS!
HOW DID YOU
GET IN HERE?

THE NAME'S
BESSIE and GATE
C WAS OPEN -



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 "computer" 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 COMMODORE 64 ... but would be awfully hard to play *Invaders* with.

EXERCISE 35-1: Using the above program as a model, and the "OR logic" seen in Figure 35-2, write a program which will report Bess' 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"
```

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 THEN 60
40 PRINT "N WAS NOT GREATER THAN 5"
50 END
60 PRINT "N WAS GREATER THAN 5"
```

...and RUN.

Line 20 is obviously the key one, containing NOT. 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...)

Order of Operations

When trying to figure out which gets calculated first in the thick of a “humongous” equation, here’s the 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. Exponentiation -- 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. Addition and subtraction -- from left to right.
5. Less than, greater than, equals, less or equal to, greater or equal to, not equal to -- from left to right.
6. The logical NOT.
7. The logical AND.
8. The logical OR.

And In Conclusion

Logical math is worth the hassle. As one last fun program, enter and RUN this “Midnight Inspection.” Line 100 checks each response for a NO answer (instead of a YES). Using logical OR, it branches to the “no-go” statement (Line 120) if any one of the tests is negative (“N”).

```
10 PRINT "SHIFT CLR/HOME"
20 PRINT "ANSWER WITH 'Y' OR 'N'."
30 PRINT
40 INPUT "HAS THE CAT BEEN PUT OUT";A$
50 INPUT "PORCH LIGHT TURNED OFF";B$
60 INPUT "ALL DOORS/WINDOWS LOCKED";C$
70 INPUT "IS THE T.V. TURNED OFF";D$
80 INPUT "THERMOSTAT TURNED DOWN";E$
90 PRINT:PRINT
100 IF A$="N" OR B$="N" OR C$="N" OR D$="N" OR E$="N" THEN 120
110 PRINT "          GOODNIGHT":END
120 PRINT "SOMETHING HAS NOT BEEN DONE."
130 PRINT "DO NOT GO TO BED"
140 PRINT "UNTIL YOU FIND THE PROBLEM!"
150 GOTO 30
```

In most cases, AND and OR statements are interchangeable if other parts of a program are rewritten to accommodate the switch.

Learned in Chapter 35

Miscellaneous

Logical AND
Logical OR
Logical NOT
Order of Operations

A Study Of Obscurities

COMMODORE 64 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.

USR

The USR 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 USR you’ll need ASSEMBLY language skills -- a whole book in itself. Since USR is a BASIC word, we will mention it here. However, since its use is beyond the scope of our book, we suggest using an assembly language manual.

SYS

The SYS statement is an even easier way to jump to an assembly language program ... We simply say:

```
SYS <memory-location>
```

and control will jump to whatever memory location we gave it (now he tells us). When the Computer hits an RTS (return to subroutine) instruction at the

THAT'S
PRETTY
OBSCURE.

USER
SYS



end of our assembly language program, control will jump right back to the Line after our SYS call in BASIC.

That's as far as we're going to press our luck on this one right now. We don't want to leave so terror-stricken that we won't continue.

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.

Learned In Chapter 36

Statements

SYS

Functions

USR

GET

The GET 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 PRINT "SHIFT CLR/HOME"  
20 GET A$ : IF A$ = "T" THEN 40  
30 GOTO 20  
40 PRINT "YOU HIT THE LETTER 'T'"  
50 GOTO 20
```

...and RUN.

Press a variety of individual alpha and numeric keys, but not **RUN/STOP**.

The keyboard seems to be dead, until we hit the "T" key. Why?

Aha! If the test in Line 20 passes, execution moves to Line 40 and a message is PRINTed. Then the process starts over. Hit T again. Hold it down. Press **RUN/STOP** to exit program.

The way GET works is very simple. The keyboard is constantly scanned by the Computer, checking to see if any key is pressed. When a key is depressed, it is assigned to the string variable associated with the GET statement. The

string variable in our example is A\$. To see if this is really true, add the following line and RUN again. Try many different keys, both alpha and numeric.

```
25 PRINT A$
```

It should be noted here that the **64** keyboard has a ten character buffer. If we had a more elongated program whereby the Computer was performing some other operation than GET, then any character we enter, up to 10, would be stored in the keyboard buffer. Any characters entered after the 10th character, would be lost. When the Computer reaches the GET statement, then one by one each character is called from the buffer.

To get the hang of this, add the following Line to the resident program:

```
22 FOR Z = 1 TO 4000 : NEXT Z
```

Now type RUN and then COMMODORE.

Slowly, but surely, COMMODORE appears on the screen.

Rapid Scanner

If GET scans the buffer and does not find a key pressed (the usual case), it is said to read a "null string". A null string is represented by two quote marks with nothing between them, thus:

```
" "
```

The ASCII code for null is 0.

We can GET numeric data, using K instead of K\$, but if we accidentally type a character other than a number, we'll get a ?SYNTAX ERROR ... it's better to GET input as strings, using K\$ and convert them to numbers afterwards.

To see how fast we can scan for input with GET, try this NEW program:

```
10 GET K$
```

```
20 IF K$ = "" THEN PRINT "NO KEYBOARD ";
30 PRINT "INPUT"
40 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 GET? So simple, yet the possibilities are enormous. Only a lot of experimenting will make you comfortable with it, but GET 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 the car. It seemed a bit officious, but elections were far away...

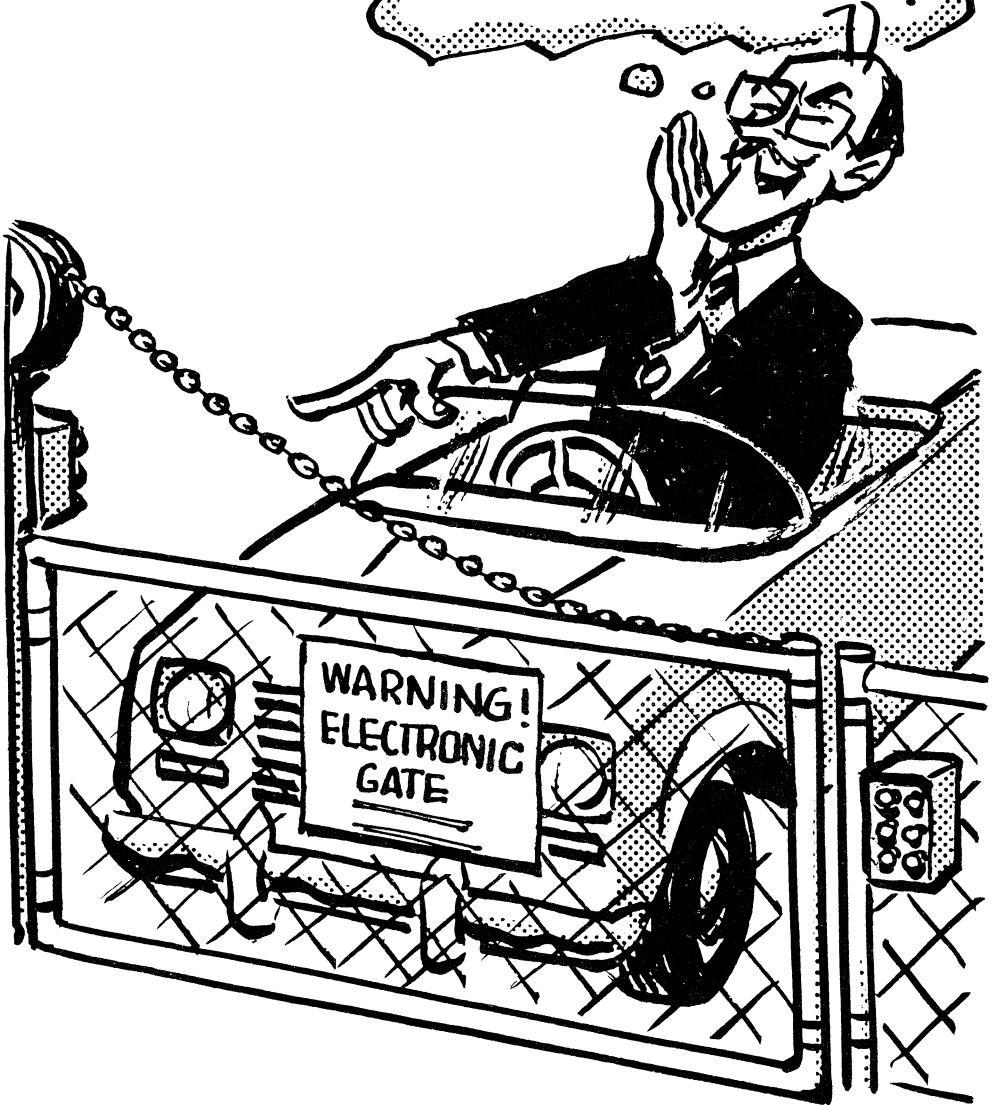
Anyway, to get my car thru the security fence it was necessary to drive to an electrically-operated gate. A secret code was punched into a numeric keypad for some sort of computer to analyze, and it controlled the motorized gate. *The secret code number was 1930.*

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 GET, and is offered here for your amusement, amazement and careful study.

```
10 PRINT "SHIFT CLR/HOME"
20 PRINT TAB(11);"THE GATE IS CLOSED"
30 PRINT : PRINT : PRINT : PRINT : PRINT
```

**OPEN,
SAYS ME!**

**OR I'LL WRITE A
PROGRAM THAT'LL
REALLY GET YA!**



```
40 PRINT TAB(10); "TYPE THE COMBINATION"  
50 PRINT TAB(10); "FOLLOWED BY A PERIOD"  
60 GET K$ : IF K$ = "" GOTO 60  
70 READ D$ : IF D$ = "." GOTO 100  
80 IF D$ = K$ GOTO 60  
90 RESTORE : GOTO 60  
100 PRINT "SHIFT CLR/HOME"  
110 PRINT TAB(11); "YOU MAY ENTER NOW"  
120 PRINT : PRINT : PRINT : PRINT  
130 PRINT TAB(9); "WAIT FOR GATE TO OPEN"  
140 FOR T=1 TO 3000 : NEXT T  
150 RESTORE : GOTO 10  
1000 DATA 1,9,3,0.
```

The password (1930 followed by a period) is imbedded a character at a time in DATA Line 90. The commas only separate the characters and should be typed in to open the gate.

Line 60 holds the magic. It checks the input buffer looking for something besides a null string. If it finds a key pressed, execution drops to Line 70.

Line 70 READs a piece of DATA. If it happens to be a period (which can only be READ from DATA after each of the other code characters have been READ), execution moves to Line 100 which tells us to enter the premises.

If, however, the test in Line 70 does not find a period, execution defaults to the next test, in Line 80.

Line 80 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 60 for GET to await another keyboard character. If the keyboard and DATA characters don't match, the test fails and execution drops to Line 90.

Line 90 RESTOREs the DATA pointer back to its beginning, and returns execution to Line 60 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 140 merely allows the gate a brief time to open and close (and us to read the screen), then RESTOREs the DATA and starts the program over again from the beginning.

The password can be changed to any combination of characters by changing Line 1000.

If you wanted it to be "COMMODORE 64" for example:

```
1000 DATA C , O , M , M , O , D , O , R , E , " " 6 , 4 , ,
```

Or "OPENSESAME"

```
1000 DATA O , P , E , N , S , E , S , A , M , E , ,
```

Don't forget that last piece of DATA, the period. By changing Line 70, of course, we could change that period to anything we wanted.

Happy gate crashing!

Learned In Chapter 37

Functions

GET

Miscellaneous

Input Buffer

PART 8

PROGRAM
CONTROL

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 would 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*.

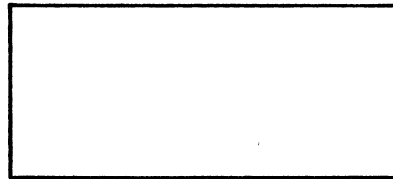
Flowcharts are most helpful when kept simple. A cluttered flowchart is hard to read and usually isn't much more helpful than an ordinary program LIST-ing. A good flowchart is also helpful for “documentation” to give us (or others) a picture of how the program works -- for later on, when we've forgotten.

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.

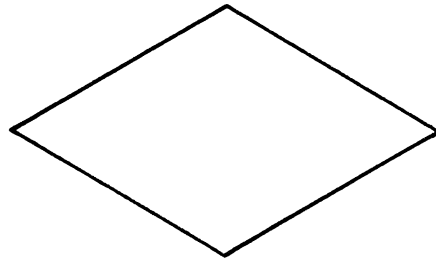
BEGIN or END



PROCESSING BLOCK
(something the
Computer does without
making any decisions)



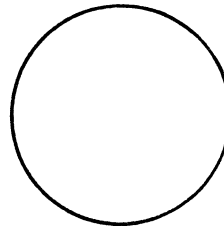
DECISION DIAMOND
(branches off in different
directions, depending on the
decision it makes.)



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.

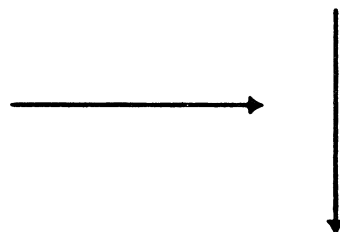
Another useful symbol is:

CONTINUATION



The circle usually has a number inside it which corresponds to a number on another page if the flowchart is too large for a single sheet.

CONNECTOR ARROWS

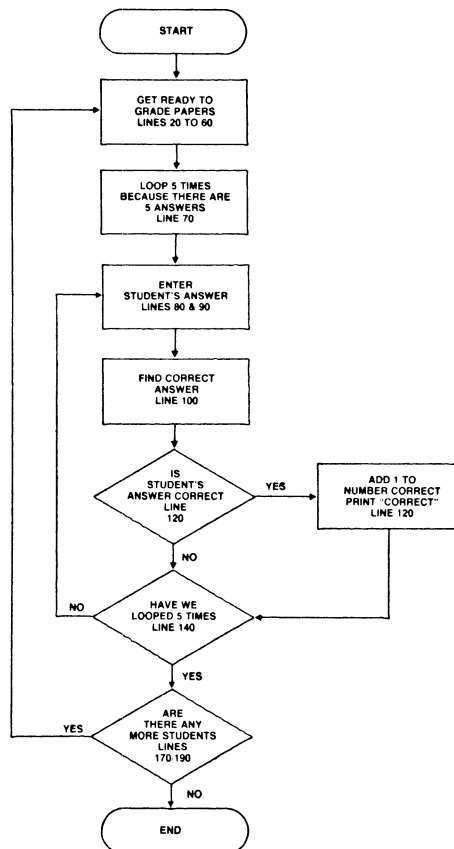


Arrows indicate the direction in which program execution proceeds.

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 38-1: 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:

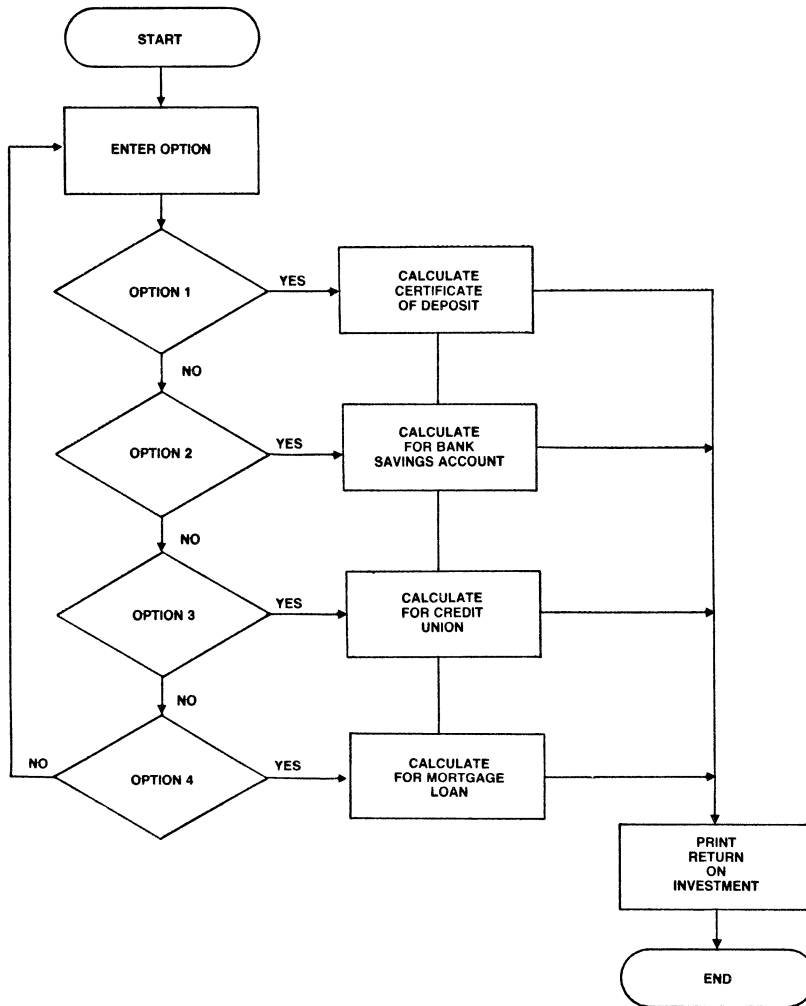
- 1 - CERTIFICATE OF DEPOSIT
- 2 - BANK SAVINGS ACCOUNT
- 3 - CREDIT UNION
- 4 - MONEY MARKET FUND

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 38-2: Write the master program as flowcharted, with a branch to a program to calculate the return on a Bank Savings Account paying simple interest.

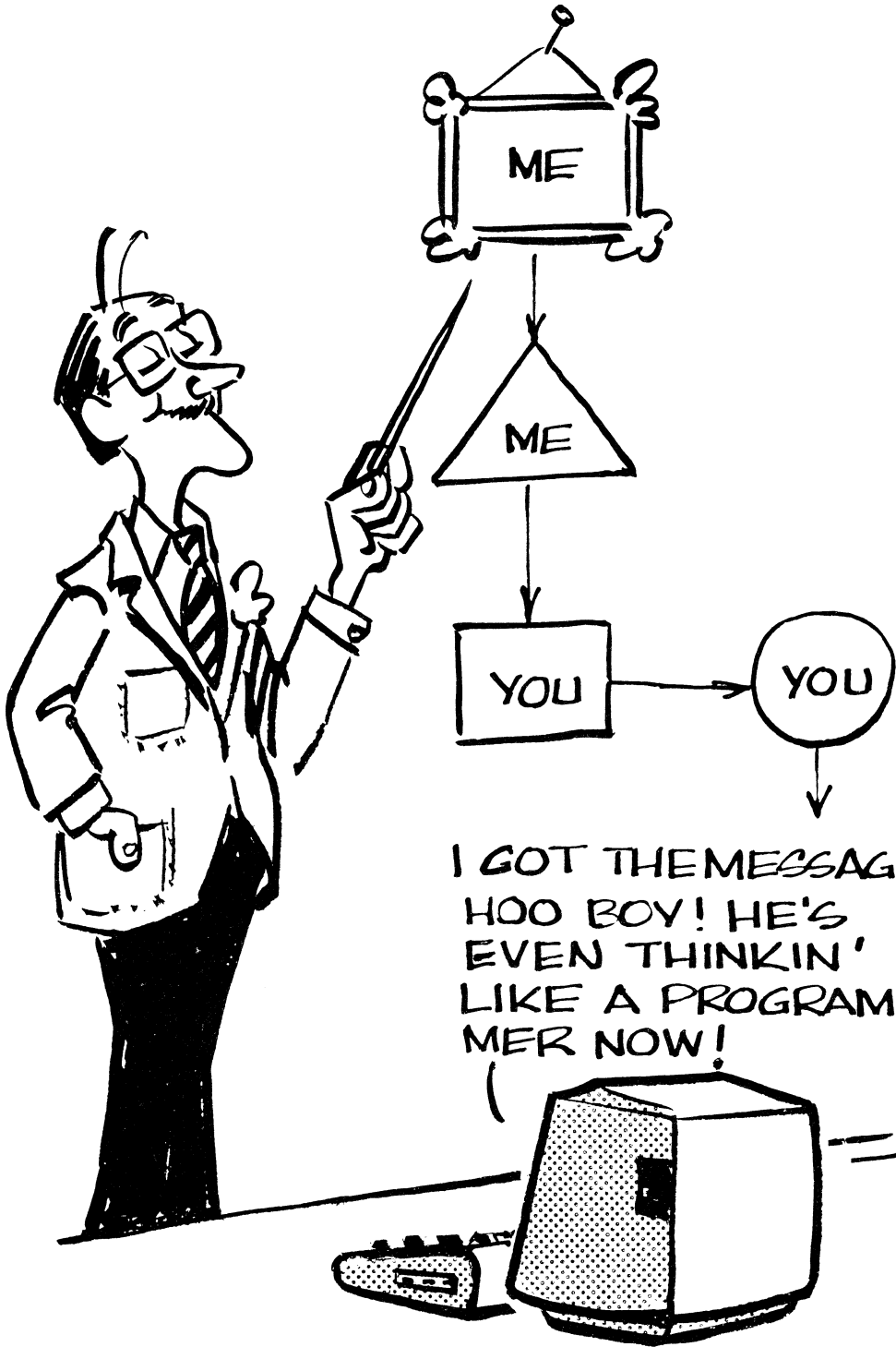
EXERCISE 38-3: Choose a program from an early Chapter and design your own flowchart.



Learned In Chapter 38

Miscellaneous

Flowcharting



I GOT THE MESSAGE!
HOO BOY! HE'S
EVEN THINKIN'
LIKE A PROGRAM-
MER NOW!

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 Ft. Worth 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:

```
? FRE(0) - (FRE(0) < 0) * 65536
```

If there is no program loaded into memory, the answer should be:

```
38909
```

If the answer is too large, there may be trouble.

Possible Solution

If this seems to be a problem, 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, type 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 PRINT "SHIFT CLR/HOME"
20 DIM A(MM)
30 PRINT "TESTING RAM, PLEASE STAND BY..."
40 FOR N=1 TO MM
50 A(X)=X
60 NEXT
70 PRINT "TEST COMPLETE. ERROR = ";ERR
```

...and RUN. The test should take about 30 seconds.

If errors show up, you may have found the problem.

You will probably want to enter this test program into your Computer, try it out before you need it, then SAVE it on disk or tape and hope that you won't ... (need it, that is).

Video Display Problems?

If adjusting the brightness, contrast, color, horizontal and vertical sync, etc., fail to give the desired display, the problem could be in the Computer.

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. Are you using correct syntax?

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 GG
ERROR: Must be a variable recognizable by the
Computer.
```

- c. Forget the Line number, accidentally mix a letter in with the number, or use a Line number larger than 65529?

EXAMPLE:

```
72B3 PRINT "BAD LINE NUMBER."
|
└─ ERROR
```

- d. Accidentally have a double quotation mark in the text?

EXAMPLE:

```
10 PRINT "HE SAID "HELLO THERE."
```

- e. Type a Line more than 70 characters long?
- f. Misspell PRINT or INPUT (*It happens!*)?
- g. Accidentally type a stray character in the Line, especially an extra comma or semicolon?
- h. Accidentally enter invisible characters?
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.

"PHYSICIAN - HEAL
THYSELF!"



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, or proceed without looping.

- 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
```

4. If the goofed-up statement is an IF-THEN or GOTO.
- a. Does the Line number 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 “BAD SUBSCRIPT ERROR”.
- a. Did you exceed the limits of one of the built-in functions?

7. 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 calculator mode (no Line number).

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"
398 IF X<0 THEN PRINT "X OUT OF"
399 PRINT "RANGE AT #398" : STOP
```

Line 299 checks whether the Line immediately following Line 299 is executed. Line 398 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 calculator mode, 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 type:

```
GOTO (line number)
```

This is a useful tool for working back through a program. Give the variables acceptable values using calculator-mode statements, 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 that it’s not necessary to LIST the entire program just to look at the one section. Type:

```
LIST (line number)-(line number)
```

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 39

Miscellaneous

Defensive programming
Computer-detected errors
Flags
Hardware checkout procedures

CONGRATULATIONS,
PAL, WE DID IT!!
MIND IF I RIPOFF
A FEW NOTES OF
POMP *and* CIRCUMSTANCE?





SECTION B

**ANSWERS
TO
EXERCISES**

SAMPLE ANSWER FOR EXERCISE 4-1:

```
50 PRINT D
```

SAMPLE RUN FOR EXERCISE 4-1:

```
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 4 and continue.

SAMPLE ANSWER FOR EXERCISE 4-2:

```
10 REM * TIME SOLUTION *
20 D = 6000
30 R = 500
40 T = D / R
50 PRINT "IT TAKES";T;"HOURS."
```

Note: Remember to **RETURN** each Line.

SAMPLE RUN FOR EXERCISE 4-2:

```
IT TAKES 12 HOURS.
```

Note: In order to arrive at the formula in Line 40 it is necessary to transpose $D = R * T$ and express in terms of T.

SAMPLE ANSWER FOR EXERCISE 4-3:

```
10 REM * CIRCUMFERENCE SOLUTION *
20 D = 35
30 C = π * D
40 PRINT "CIRCUMFERENCE =" ;C;"FEET."
```

SAMPLE RUN FOR EXERCISE 4-3:

```
CIRCUMFERENCE = 109.955743 FEET.
```

SAMPLE ANSWER FOR EXERCISE 4-4:

```

10 REM * CIRCULAR AREA SOLUTION *
20 R = 5
30 A = π * R * R
40 PRINT "AREA =" ;A;" SQUARE INCHES."

```

SAMPLE RUN FOR EXERCISE 4-4:

```

AREA = 78.5398164 SQUARE INCHES.

```

Note: Some BASICs do not have a function which means “raise to the power” to handle R^2 (**64** BASIC does.) In easy cases like this one, we can simply use R times R EXPONENTIATION function as we proceed.

SAMPLE ANSWER FOR EXERCISE 4-5:

```

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 4-5:

```

YOUR NEW BALANCE IS $ 188

```

SAMPLE ANSWER FOR EXERCISE 5-1:

```

10 REM * CAR MILES SOLUTION PROGRAM *
20 N = 10000000
30 D = 10000
40 T = N * D
50 PRINT "THE TOTAL MILES DRIVEN IS" ;T

```

SAMPLE RUN FOR EXERCISE 5-1:

```

THE TOTAL MILES DRIVEN IS 1E+10

```

Note: As discussed earlier, the answer is the number 1 followed by ten zeros. 10,000,000,000. Ten Billion. The Computer will not print any numbers over 999,999,999 or under .01 without converting them to exponential notation.

SAMPLE ANSWER FOR EXERCISE 5-2:

```
20 N = 1E+6
30 D = 1E+4
```

SAMPLE RUN FOR EXERCISE 5-2:

```
THE TOTAL MILES DRIVEN IS 1E+10
```

Note: The answer came out exactly the same as before, meaning we not only receive answers in SSN, but can also use it in our programs.

SAMPLE ANSWER FOR EXERCISE 6-1:

```
10 REM * FAHRENHEIT TO CELSIUS *
20 F = 65
30 C = (F-32) * (5/9)
40 PRINT F;"DEG. FAHR. =" ;C;"DEG. CEL."
```

SAMPLE RUN FOR EXERCISE 6-1:

```
65 DEG. FAHR. = 18.3333333 DEG. CEL.
```

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 6-2:

```
30 C = F-32 * (5/9)
```

SAMPLE RUN FOR EXERCISE 6-2:

```
65 DEG. FAHR. = 47.2222222 DEG. CEL.
```

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 6-3:

$$30 \text{ C} = (F-32) * 5/9$$

SAMPLE RUN FOR EXERCISE 6-3:

$$65 \text{ DEG. FAHR.} = 18.3333333 \text{ DEG. CEL.}$$

SAMPLE ANSWER FOR EXERCISE 6-4:

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

$$10 \text{ A} = 30 - (9 - 8) - (7 - 6)$$

Try a few on your own.

SAMPLE ANSWER FOR EXERCISE 7-1:

```
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 7-1:

A EQUALS 5.

SAMPLE ANSWER FOR EXERCISE 7-2:

```
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 7-2:

```
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 on to Line 90 after PRINTing Line 70.

SAMPLE ANSWER FOR EXERCISE 12-1:

```
1 PRINT "SHIFT CLR/HOME"
2 INPUT "DELAY (IN SECONDS) YOU WISH";S
3 P = 700
4 D = S * P
5 FOR X = 1 TO D
6 NEXT X
7 PRINT "DELAY 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 12-2:

```
60 PRINT "RATE      ", "TIME"
70 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 12-3:

```
5 PRINT "SHIFT CLR/HOME" : PRINT
10 PRINT "*** SALARY RATE CHART ***"
20 PRINT
30 PRINT " YEAR", " MONTH"
40 PRINT
50 FOR Y = 5000 TO 20000 STEP 1000
60 REM CONVERT YEARLY TO MONTHLY
70 M = Y / 12
80 PRINT Y, M
90 NEXT Y
```

SAMPLE RUN FOR EXERCISE 12-3:

```
*** SALARY RATE CHART ***

YEAR                MONTH

5000                416.666667
6000                500
7000                583.333333
                   etc.
```

SAMPLE ANSWER FOR EXERCISE 12-4:

```

10 R = .01
20 D = 1
30 T = .01
35 PRINT "SHIFT CLR/HOME"
40 PRINT "DAY","DAILY","TOTAL"
50 PRINT " #","RATE","EARNED"
60 PRINT
70 PRINT D,R,T
80 IF R > 1E+6 THEN END
90 R = R * 2
100 D = D + 1
110 T = T + R
120 GOTO 70

```

SAMPLE RUN FOR EXERCISE 12-4:

DAY #	DAILY RATE	TOTAL EARNED
1	.01	.01
2	.02	.03
3	.04	.07
4	.08	.15
5	.16	.31
6	.32	.63

SAMPLE ANSWER FOR EXERCISE 12-5:

```

5 PRINT "SHIFT CLR/HOME"
10 PRINT "LENGTH","WIDTH","AREA"
20 FOR L = 0 TO 500 STEP 50
30 W = (1000 - 2 * L) / 2
40 A = L * W
50 PRINT L,W,A
60 NEXT L

```

SAMPLE RUN FOR EXERCISE 12-5:

LENGTH	WIDTH	AREA
0	500	0
50	450	22500
100	400	40000
150	350	52500
200	300	60000
etc.		

ADDENDUM TO EXERCISE 12-5:

Here's a program that lets the Computer do the comparing:

```

1 PRINT "SHIFT CLR/HOME"
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 AVAIL.
30 F = 1000
35 REM * L IS LENGTH OF ONE SIDE *
40 FOR L=0 TO 500 STEP 50
45 REM * W IS WIDTH OF ONE SIDE *
50 W = (F - 2 * L) / 2
60 A = W * L
65 REM COMPARE WITH A CURRENT MAXIMUM
67 REM REPLACE IF NECESSARY
70 IF A <= M THEN 100
80 M = A
85 REM UPDATE CURRENT DESIRED LENGTH
90 N = L
100 NEXT L
110 PRINT "FOR LARGEST AREA "
112 PRINT "USE THESE DIMENSIONS:"
115 PRINT
120 PRINT ,N;"FT. BY";500-N;"FT."
125 PRINT
130 PRINT "FOR A TOTAL AREA OF";M;
140 PRINT "SQ. FEET."

```

SAMPLE ANSWER FOR EXERCISE 13-1:

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

SAMPLE ANSWER FOR EXERCISE 13-2:

```
5 PRINT "SHIFT CLR/HOME" : PRINT  
10 PRINT "*** SALARY RATE CHART ***"  
20 PRINT  
30 PRINT " YEAR", " MONTH", " WEEKLY"  
40 PRINT  
50 FOR Y = 5000 TO 20000 STEP 1000  
60 REM * CONVERT YEARLY TO MONTHLY  
70 M = Y / 12  
73 REM * CONVERT YEARLY TO WEEKLY  
77 W = Y / 52  
80 PRINT Y, M  
85 PRINT TAB(25); W  
90 NEXT Y
```

SAMPLE ANSWER FOR EXERCISE 14-1:

```
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 14-2:

The program will be the same as the answer to Exercise 14-1 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 15-1:

Addition of the following single Line gives a nice clean PRINTout with all the values "rounded" to their integer value:

```
45 A = INT(A)
```

Worth all the effort to learn it, wasn't it?

SAMPLE ANSWER FOR EXERCISE 15-2:

```
45 A = INT(10 * A) / 10
```

When 3.14159265 was multiplied times 10 it became 31.4159265. The INTEger value of 31.4159265 is 31. 31 divided by 10 is 3.1, etc.

SAMPLE ANSWER FOR EXERCISE 15-3:

This was almost too easy.

```
45 A = INT(100 * A) / 100
```

SAMPLE ANSWER FOR EXERCISE 16-1:

```
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 20-1:

```
10 PRINT CHR$(67);CHR$(79);CHR$(77);
15 PRINT CHR$(77);CHR$(79);CHR$(68);
17 PRINT CHR$(79);CHR$(82);CHR$(69);
20 PRINT CHR$(32);CHR$(54);CHR$(52)
```

SAMPLE ANSWER FOR EXERCISE 20-2:

```
10 INPUT "ENTER A NUMBER";A$
20 A = ASC(A$)
30 IF A<48 THEN 10
40 IF A>57 THEN 10
50 PRINT "ASCII VALUE OF ";A$;" IS";A
```

SAMPLE ANSWER FOR EXERCISE 21-1:

```
10 PRINT "SHIFT CLR/HOME" : PRINT
20 INPUT "FIRST STRING";A$
30 INPUT "SECOND STRING";B$
40 PRINT : PRINT "ALPHABETICAL ORDER:"
50 IF A$ < B$ THEN PRINT A$,B$ : END
60 PRINT B$,A$
```

SAMPLE ANSWER FOR EXERCISE 22-1:

```
10 PRINT "SHIFT CLR/HOME"
20 INPUT "INPUT STRING";A$
30 IF LEN(A$)>10 THEN GOTO 50
40 STOP
50 PRINT "THE 10 CHARACTER LIMIT WAS ";
60 PRINT "EXCEEDED"
```

SAMPLE ANSWER FOR EXERCISE 22-2:

```

10 PRINT "SHIFT CLR/HOME"
20 INPUT "ENTER PASSWORD";A$
30 FOR X=1 TO 11
40 READ N
50 P$ = P$ + CHR$(N)
60 NEXT X
70 IF A$ = P$ THEN 100
80 PRINT "WRONG PASSWORD, GET LOST!"
90 END
100 PRINT "CORRECT PASSWORD-ENTER"
110 DATA 79,80,69,78,32,83
120 DATA 69,83,65,77,69

```

SAMPLE ANSWER FOR EXERCISE 23-1:

```

1 PRINT "SHIFT CLR/HOME"
10 PRINT "INPUT YOUR STREET ADDRESS:"
20 INPUT A$
30 A = VAL(A$)
40 PRINT: PRINT "YOUR NEIGHBOR'S "
45 PRINT "STREET NUMBER IS ";A+4
50 PRINT : LIST

```

SAMPLE ANSWER FOR EXERCISE 23-2:

```

10 PRINT "SHIFT CLR/HOME"
20 FOR X = 101 TO 120
30 A$ = STR$(X)
40 PRINT A$+"WT",
50 NEXT X
60 PRINT : LIST

```

SAMPLE RUN FOR EXERCISE 23-2:

```

101WT      102WT      103WT      104WT
105WT      106WT      107WT      108WT
109WT      110WT      111WT      112WT
113WT      114WT      115WT      116WT
117WT      118WT      119WT      120WT

```

SAMPLE ANSWER FOR EXERCISE 24-1:

```
10 PRINT "SHIFT CLR/HOME" : PRINT
20 INPUT "IS THIS A SMART COMPUTER";A$
30 B$ = LEFT$(A$,1)
40 IF B$="Y" THEN PRINT "FOR SURE": END
50 IF B$="N" THEN PRINT "NEGATIVE": END
60 PRINT "THIS IS A YES OR NO QUESTION"
70 GOTO 20
```

SAMPLE ANSWER FOR EXERCISE 24-2:

```
10 PRINT "SHIFT CLR/HOME" : 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 ";
75 PRINT "LARGEST NUMBER IS ";P$
80 PRINT : LIST
90 DATA N106WT,A208FM,Z154DX
```

SAMPLE ANSWER FOR EXERCISE 24-3:

Choice C. P-

SAMPLE ANSWER FOR EXERCISE 25-1:

```
10 PRINT "SHIFT CLR/HOME"
20 A = 5 : B = 12
30 C = SQR(A^2 + B^2)
40 PRINT "THE SQUARE ROOT OF";A;
50 PRINT "SQUARED "
60 PRINT "PLUS";B;"SQUARED IS";C
70 PRINT : LIST
```

SAMPLE ANSWER FOR EXERCISE 25-2:

```
10 INPUT "ENTER A NUMBER";N
20 PRINT "LOG(EXP(";N;"))=";LOG(EXP(N))
30 PRINT "EXP(LOG(";N;"))=";EXP(LOG(N))
40 PRINT
50 GOTO 10
```


SAMPLE ANSWER FOR EXERCISE 28-1:

Add or change the following Lines:

```

10 DIM A(210) : PRINT "SHIFT CLR/HOME"
85 FOR B=201 TO 210
90 READ A(B)
95 NEXT B
97 PRINT
100 PRINT "CAR#","ENG. SIZE",
105 PRINT "COLOR","BODY STYLE"
110 PRINT W,A(W),A(W+100),A(W+200)
530 DATA 20,20,10,20,30,20,30,10,20,20

```

SAMPLE ANSWER FOR EXERCISE 28-2:

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 29-1:

Change Line 60 to:

```

60 IF A$(S) <= A$(F) THEN 100

```

Another 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 30-1:

```

10 PRINT "SHIFT CLR/HOME"
20 FOR E=1 TO 4
30 FOR D=1 TO 3
35 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 30-2:

Add:

```
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 30-3:

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 34-1:

```
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 90
60 PRINT "OLD BESSIE IS SECURE ";
70 PRINT "IN PASTURE #1"
80 END
90 PRINT "A GATE IS OPEN. "
100 PRINT "OLD BESSIE IS FREE TO ROAM."
```

SAMPLE ANSWER FOR EXERCISE 37-1:

```

10 REM * TEST GRADER *
20 PRINT "ENTER FIVE ANSWERS"
30 N = 0 : PRINT "SHIFT CLR/HOME"
40 DIM B(5)
50 FOR J=1 TO 5 : READ B(J) : NEXT
100 FOR I = 1 TO 5
110 PRINT "ANSWER NUMBER";I;";";
120 INPUT A
130 PRINT A,B(I)
140 IF A=B(I) THEN 170
150 PRINT "WRONG"
160 GOTO 190
170 PRINT "CORRECT"
180 N=N+1
190 PRINT
200 NEXT I
210 PRINT N;"RIGHT OUT OF 5 IS";
220 PRINT N/5 * 100;"PERCENT."
230 DATA 12,45,38,26,39

```

SAMPLE ANSWER FOR EXERCISE 37-2:

```

100 PRINT "SHIFT CLR/HOME"
110 PRINT : PRINT
120 PRINT "INVESTMENT SELECTIONS:"
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 NUMBER";F
190 ON F GOTO 1000,2000,3000,4000
200 GOTO 100 : REM * F NOT BTWN 1 & 4 *
1000 REM * CERTIFICATE OF DEPOSIT *
1010 PRINT : PRINT "THE C.D. PROGRAM ";
1015 PRINT "HAS YET TO BE WRITTEN."
1020 GOSUB 10000 : GOTO 100
2000 REM * BANK SAVINGS ACCOUNT *
2010 PRINT "SHIFT CLR/HOME" : PRINT
2015 PRINT "THIS ROUTINE CALCULATES ";
2017 PRINT "SIMPLE INTEREST ON DOLLARS";
2020 PRINT "HELD IN DEPOSIT FOR A ";

```

```
2030 PRINT "SPECIFIED PERIOD USING A ";
2035 PRINT "SPECIFIED PERCENTAGE OF ";
2037 PRINT "INTEREST." : PRINT
2040 PRINT : PRINT "ENTER DEPOSIT ";
2045 INPUT "(IN DOLLARS):";P
2050 PRINT "HOW LONG TO LEAVE ";
2055 INPUT "IT IN (IN DAYS):";D
2060 PRINT "EXPECTED INTEREST RATE ";
2065 INPUT (IN %):";R
2070 PRINT "SHIFT CLR/HOME" : PRINT : PRINT
2075 PRINT "A STARTING PRINCIPAL OF $";P
2080 PRINT "AT A RATE OF";R;"% FOR";
2085 PRINT D;"DAYS,"
2090 PRINT "YIELDS INTEREST OF:"
2100 REM INTEREST = (%/YR) / (DAYS/YR) *
2110 REM DAYS * PRINCIPAL
2200 I = R / 100 / 365 * D * P
2300 PRINT : PRINT ,"$";I
2400 END
3000 REM * CREDIT UNION PROGRAM *
3010 PRINT : PRINT "THE C.U. PROGRAM"
3015 PRINT "HAS YET TO BE WRITTEN"
3020 GOSUB 10000 : GOTO 100
4000 REM * MORTGAGE LOAN PROGRAM *
4010 PRINT : PRINT "THE M.L. PROGRAM"
4015 PRINT "HAS YET TO BE WRITTEN"
4020 GOSUB 10000 : GOTO 100
10000 FOR I=1 TO 1500 : NEXT I : RETURN
```




SECTION C

**PREPARED
USER
PROGRAMS**

***** 12-Hour Clock *****

```
1 REM * COPYRIGHT (C) 1984 BY COMPUSOFT
2 REM ALL RIGHTS RESERVED. *
3 REM *   <<< 12-HOUR CLOCK   >>>   *
10 INPUT "THE HOUR IS";E
20 F = INT(E / 10) : E = E - (F * 10)
30 INPUT "THE MINUTES ARE";C
40 D = INT(C / 10) : C = C - (D * 10)
50 INPUT "THE SECONDS ARE";A
60 B = INT(A / 10) : A = A - (B * 10)
70 PRINT "SHIFT CLR/HOME"
80 PRINT : FOR N=1 TO 915 : NEXT
90 A = A + 1
100 IF A > 9 THEN 120
110 GOTO 310
120 A = 0
130 B = B + 1
140 IF B > 5 THEN 160
150 GOTO 310
160 B = 0
170 C = C + 1
180 IF C > 9 THEN 200
190 GOTO 310
200 C = 0
210 D = D + 1
220 IF D > 5 THEN 240
230 GOTO 310
240 D = 0
250 E = E + 1
260 IF E > 9 THEN 280
270 GOTO 300
280 E = 0
290 F = F + 1
300 IF (F=1) AND (E=3) THEN 330
310 PRINT ,F;E": "D;C": "B;A
320 GOTO 80
330 A=0 : B=0 : C=0 : D=0 : E=0 : F=0
340 GOTO 310
```


Checksum For Business

For those responsible for inventory numbers or check clearing and balancing in business, a checksum is a most useful testing "code". This simple program calculates error-free checksums almost instantly. It is designed for 6-digit numbers and so can be used for stock number verification or other applications.

```

1 REM * COPYRIGHT (C) 1984 BY COMPUSOFT.
2 REM ALL RIGHTS RESERVED. *
3 REM * <<< CHECKSUM FOR BUSINESS >>> *
10 PRINT
20 INPUT "THE FIRST DIGIT IS ";A
30 INPUT "THE SECOND DIGIT IS ";B
40 INPUT "THE THIRD DIGIT IS ";C
50 INPUT "THE FOURTH DIGIT IS ";D
60 INPUT "THE FIFTH DIGIT IS ";E
70 INPUT "THE SIXTH DIGIT IS ";F
80 PRINT
90 PRINT "THE NUMBER IS ";A;B;C;D;E;F
100 S = A+2*B+C+2*D+E+2*F
110 T = INT(S / 10)
120 U = S - T * 10
130 S = T + U
140 IF S > 9 THEN 110
150 PRINT:PRINT " THE CHECKDIGIT IS ";S

```

Speed Reading

Your Computer is your own personal Tachistoscope, a device used to practice speed reading. Study this sample program carefully to see how easy it is for you to substitute your own reading material at whatever reading level you want. The variable time loop lets you input the desired reading speed in words-per-minute.

```

1 REM * COPYRIGHT (C) 1984 BY COMPUSOFT.
2 REM ALL RIGHTS RESERVED. *
3 REM * <<< SPEED READING PROGRAM >>> *
10 GOTO 30
20 FOR I=1 TO B : NEXT : PRINT "SHIFT CLR/HOME"
25 PRINT : RETURN
30 PRINT "HOW MANY WORDS/MINUTE"

```

```
35 INPUT "DO YOU READ";W
40 B = (12 * 60 / W) * 915
60 REM * 915 IS APPROX. LOOPS/SECOND *
70 PRINT"SHIFT CLR/HOME" : PRINT
90 PRINT " FIRST PARAGRAPH FROM 'GONE WITH THE
    WIND'." : GOSUB 20
100 PRINT " SCARLETT O'HARA WAS NOT BEAUTIFUL, BUT MEN
    SELDOM      " : GOSUB 20
110 PRINT "REALIZED IT WHEN CAUGHT BY HER OWN CHARM AS
    THE TARLETON" : GOSUB 20
120 PRINT "TWINS WERE. IN HER FACE WERE TOO SHARPLY
    BLENDED THE  " : GOSUB 20
130 PRINT "DELICATE FEATURES OF HER MOTHER, A COAST
    ARISTOCRAT OF " : GOSUB 20
140 PRINT " FRENCH DESCENT, AND THE HEAVY ONES OF HER
    FLORID IRISH" :GOSUB 20
150 PRINT " FATHER. BUT IT WAS AN ARRESTING FACE,
    POINTED OF CHIN," : GOSUB 20
160 PRINT "SQUARE OF JAW. HER EYES WERE PALE GREEN
    WITHOUT A TOUCH " : GOSUB 20
170 PRINT " OF HAZEL, STARRED WITH BRISTLY BLACK LASHES
    AND SLIGHTLY" : GOSUB 20
180 PRINT" TITLED AT THE ENDS. ABOVE THEM, HER THICK
    BLACK BROWS" : GOSUB 20
190 PRINT" SLANTED UPWARDS, CUTTING A STARTLING OBLIQUE
    LINE IN HER" : GOSUB 20
200 PRINT" MAGNOLIA-WHITE SKIN--THAT SKIN SO PRIZED BY
    SOUTHERN" : GOSUB 20
210 PRINT" WOMEN AND SO CAREFULLY GUARDED WITH BONNETS,
    VEILS, AND" : GOSUB 20
220 PRINT " MITTENS AGAINST HOT GEORGIA SUNS
    " : GOSUB 20
230 PRINT "SHIFT CLR/HOME" : PRINT
```

Loan Amortization

This program provides a fully developed installment plan for the repayment of small-to-moderate size loans, such as car or home improvement loans. It includes all instructions necessary. Use it with common sense; in the last payment period, amounts may be carried out to a fraction of a cent.

Challenge: modify the program to eliminate fractional-cent payments, without changing the total amount paid as interest or principal.

```

1 REM * COPYRIGHT (C) 1984 BY COMPUSOFT,
2 REM ALL RIGHTS RESERVED. *
3 REM *   <<< LOAN AMORTIZATION >>>   *
10 PRINT "SHIFT CLR/HOME" : PRINT
15 INPUT "PRINCIPAL";P
20 INPUT "NUMBER OF PERIODS";L
30 INPUT "INTEREST RATE";R
40 I = R / 12 : I = I / 100
50 T = 1 : FOR X=1 TO L
60 T = T * (1 + I) : NEXT X : T = 1 / T
70 T = 1 - T
80 M = P * I / T
90 M = INT(M * 100 + .5) / 100
100 PRINT "SHIFT CLR/HOME" : PRINT
110 FOR Z=1 TO L
120 IF Z/3 <> INT(Z/3) THEN 140
130 PRINT : PRINT "   PRESS <RETURN>";
132 INPUT "TO CONTINUE";A$
135 PRINT "SHIFT CLR/HOME" : PRINT
140 A = (INT(P * I * 100 + .5)) / 100
150 B = M - A : P = P - B
160 IF P < M THEN M = P
170 B = M - A
180 PRINT "PAYMENT NUMBER:"Z
190 PRINT "REMAINING PRINCIPAL:"P
200 PRINT "MONTHLY PAYMENT:"M
210 PRINT "PRINCIPAL PAYMENT:"B
220 PRINT "INTEREST PAYMENT:"A
230 PRINT
240 NEXT Z

```

Dow-Jones Industrial Average Forecaster

There is no guarantee that this program will make you instantly wealthy, but it is an example of converting a financial magazine article into a usable computer program. The article describing the market premises on which this program is built appeared in Forbes Magazine.

```

1 REM * COPYRIGHT (C) 1984 BY COMPUSOFT
2 REM   ALL RIGHTS RESERVED. *
3 REM * <DOW JONES AVERAGE FORECASTER> *
10 PRINT "SHIFT CLR/HOME" : PRINT
20 PRINT "* PROJECTS TARGET ";
25 PRINT "DOW-JONES INDUSTRIAL "
30 PRINT "AVERAGES AS A FUNCTION OF "
35 PRINT "YEARS DJI EARNINGS AND ";
40 PRINT "INFLATION RATE *" : PRINT
50 REM * K = COST OF MONEY, ASSUMES 3% * 60 K = .03
70 REM * P = RISK PREMIUM OF STOCKS
75 REM OVER BONDS, ASSUMES 1% *
80 P = .01
90 PRINT "DO YOU KNOW YEARS PROJECTED"
100 INPUT "EARNINGS OF 30 DJI (Y/N)"; A$
110 IF A$ = "Y" THEN 290
120 PRINT
130 PRINT "THIS METHOD GIVES EARNING ";
135 PRINT "ESTIMATES " : PRINT "USING ";
137 PRINT "NEWSPAPER PRICES AND P/E "
140 PRINT "RATIOS. BETTER FORECASTS ";
145 PRINT "OF EACH" : PRINT "COMPANY'S";
150 PRINT " EARNINGS MAY GIVE AN ";
155 PRINT "IMPROVED OVERALL FORECAST"
160 PRINT
170 D = 0
180 FOR N=1 TO 30
190 READ A$
200 PRINT "WHAT IS THE CURRENT PRICE ";
205 PRINT "OF":PRINT " --> ";A$;" <--";
210 INPUT P
220 PRINT "CURRENT P/E RATIO: "; :
230 INPUT R
240 E=P/R

```

```
250 D=E+D
260 NEXT N
270 PRINT
280 GOTO 330
290 PRINT : PRINT "WHAT IS THE TOTAL ";
295 PRINT "PROJECTED EARNINGS FOR"
300 PRINT "1 SHARE OF EACH";
310 INPUT D
320 REM * I = ESTIMATED INFLATION RATE *
330 PRINT "WHAT IS THE INFLATION RATE";
340 INPUT I
350 T = D / (K + P + I * .01)
360 R = T / D
370 PRINT
380 PRINT "INFLATION RATE:"I
385 PRINT "DJI EARNINGS:"D
390 PRINT "PROJECTED DJ AVERAGE:"T
395 PRINT "AVERAGE/EARNINGS RATIO:"R
400 REM *** DATA ***
410 DATA ALLIED CHEM, ALCOA
415 DATA AMER BRANDS, AMER CAN, A.T. & T
420 DATA BETH STEEL, CHRYSLER, DUPONT
425 DATA E. KODAK, ESMARK, EXXON
430 DATA G.E., GEN FOODS, GEN MOTORS
435 DATA GOODYEAR, INCO
440 DATA INT. HARV., INT. PAPER
445 DATA JOHNS-MAN, MINN MM, OWENS-ILLS
450 DATA PROCTER & G, SEARS,STD OIL CAL
455 DATA TEXACO, UNION CARBIDE
460 DATA U.S. STEEL,UNITED TECHNOL.
465 DATA WESTINGHOUSE, WOOLWORTH
```

Craps

The game is as old as history. A testimonial to the intelligence and ingenuity of our ancient ancestors. An excellent way to demonstrate the running of twin Random Number Generators.

You don't need to know how to play the game -- the Computer will quickly teach you. (...There's one born every minute...)

```

1 REM *   COPYRIGHT (C) 1984 BY COMPUSOFT.
2 REM ALL RIGHTS RESERVED. *
3 REM *   <<<<< CRAPS >>>>>   *
10 PRINT "SHIFT CLR/HOME" : PRINT
20 INPUT "PRESS <RETURN> TO CONTINUE";A$
30 PRINT "SHIFT CLR/HOME" : PRINT : GOSUB 150 : P = N
40 PRINT : PRINT "YOU ROLLED "; P ;" ",
50 ON P GOTO 60,90,90,70,70,70,80,70,70,70,80,90
70 PRINT "YOUR POINT IS "; N : GOTO 100
80 PRINT "YOU WIN!!" : PRINT : GOTO 20
90 PRINT "YOU LOSE." : PRINT : GOTO 20
100 GOSUB 150 : M = N
110 PRINT : PRINT "YOU ROLLED "; M,
120 IF P=M THEN 80
130 IF M=7 THEN 90
140 GOTO 100
150 A = INT(RND(0)*6)+1
160 B = INT(RND(0)*6)+1 : N = A + B : RETURN

```

Automatic Ticket Number Drawer

Like to make a big splash at the next Rotary Club, Country Fair, or other ticket drawing giveaway? This program uses the random number generator to pick the lucky number(s) and eliminate charges of stuffing the ticket box, besides giving the whole affair some pizzaz. If your own number comes up and you are charged with rigging the Computer, you're on your own.

```

1 REM *   COPYRIGHT (C) 1984 BY COMPUSOFT
2 REM ALL RIGHTS RESERVED. *
3 REM *   <<< AUTO TICKET # DRAWER >>> *
10 PRINT "SHIFT CLR/HOME" : PRINT
20 REM * PICKS WINNER(S) BY DRAWING
30 REM TICKET NUMBER *

```

```

50 INPUT "THE LOWEST NUMBER IS "; B
60 PRINT
70 INPUT "THE HIGHEST NUMBER IS "; H
80 PRINT
90 E = H-B+1
120 PRINT "HOW MANY WINNERS DO ";
125 INPUT "YOU WANT";W
130 PRINT "SHIFT CLR/HOME" : PRINT
140 IF W > E THEN GOTO 280
150 PRINT : PRINT : PRINT : PRINT
160 PRINT TAB(10)"* AND THE WINNING *"
170 IF W > 1 THEN GOTO 200
180 PRINT TAB(9)"* T I C K E T   I S *"
190 GOTO 210
200 PRINT TAB(9)"* T I C K E T S   A R E *"
210 PRINT
220 FOR N= 1 TO W
230 Z = INT(RND(0)*E)+1
240 IF N/11 = INT(N/11) THEN INPUT "PRESS <RETURN> TO
      CONTINUE";A$ : PRINT "SHIFT CLR/HOME"
250 PRINT TAB(12)">----->>>" ;Z+B-1:PRINT
260 NEXT N
270 END
280 PRINT "SHIFT CLR/HOME" : PRINT
290 PRINT "YOU HAVE MORE WINNERS ";
300 PRINT "THAN TICKETS--      DUMMY!"

```

Slowpoke

The kiddies will enjoy this one. It tests reaction time. When the Computer says "GO", you press **RUN/STOP** to stop it. Then it's the next player's turn to RUN it. The Player who stops it on the smallest number wins. Any player who gets a "SLOWPOKE" has to take the dog for a walk.

With a little easy rework of the PRINT statements it can be converted into a "drunkometer" reaction time tester.

To change the speed of the printing, you can change the FOR-NEXT loop in Line 120.

```

1 REM * COPYRIGHT (C) 1984 BY COMPUSOFT.
2 REM ALL RIGHTS RESERVED *

```

```

3 REM * <<<<< SLOW POKE >>>>> *
5 PRINT "SHIFT CLR/HOME" : PRINT
10 PRINT "    GET READY . . . . . "
20 FOR B=1 TO 915 : NEXT B
30 PRINT : PRINT : PRINT
40 PRINT "GET SET . . . . . "
50 X = INT(RND(0)*1830)+10
60 FOR N=1 TO X : NEXT N
70 PRINT "SHIFT CLR/HOME" : PRINT
80 PRINT : PRINT : PRINT
90 PRINT TAB(15)" G O ! ! !"
100 FOR Z=1 TO 10
110 PRINT Z
120 FOR J=1 TO 10 : NEXT J
130 NEXT Z
140 PRINT : PRINT : PRINT
150 PRINT "    S L O W P O K E "
160 FOR N=1 TO 1000 : NEXT N

```

Format

```

1 REM * FORMATS DISKS *
3 PRINT "SHIFT CLR/HOME" : PRINT
10 INPUT "FORMAT DISK #0 (Y/N):";A$
20 IF A$<>"Y" AND A$<>"N" THEN 10
30 IF A$="N" THEN PRINT "ABORTED" : END
40 INPUT "DISK NAME:";N$
50 PRINT "2 CHARACTER ID: ";
60 GET I$ : IF I$="" THEN 60
70 PRINT I$;
80 GET D$ : IF D$="" THEN 80
90 PRINT D$ : PRINT : PRINT
95 PRINT "PLACE BLANK DISK IN DRIVE"
100 A$="" : PRINT "FORMAT AS 'N$',";
105 PRINT I$D$' (Y/N): "; : INPUT A$
110 IF A$<>"Y" THEN 10
120 CLOSE15 : OPEN15,8,15
130 PRINT#15,"N0:"N$","I$D$
140 PRINT "--FORMAT COMMAND ISSUED--"
170 PRINT "PLEASE STAND BY...";
180 CLOSE15 : PRINT "DONE" : GOTO 10

```

Random Generators





```
1 DIM S(12)
3 REM * DEMONSTRATES SINGLE VS TWIN
5 REM   RANDOM NUMBER GENERATORS
10 PRINT "SHIFT CLR/HOME" : PRINT : PRINT "HOW ";
15 INPUT "MANY THROWS DO YOU WANT";HM
20 INPUT "1 OR 2 GENERATORS: ";NG
30 PRINT : PRINT : PRINT
35 PRINT "(STAND BY WHILE I ROLL ...)"
40 FOR N=1 TO HM
50 IF NG=1 THEN T=INT(RND(0)*11)+2 : GOTO 70
60 T=INT(RND(0)*6)+INT(RND(0)*6)+2
70 S(T)=S(T)+1 : NEXT
80 PRINT : PRINT TAB(13)"GENERATORS = ";
90 IF NG=1 THEN PRINT "1" : GOTO 110
100 PRINT "2"
110 PRINT
120 PRINT TAB(10)"ROLL","OCCURENCES"
130 PRINT : FOR N=2 TO 12
140 PRINT TAB(10)N,S(N) : NEXT
150 PRINT : PRINT "AGAIN? ";
160 GET A$ : IF A$="" THEN 160
170 PRINT A$;
180 IF A$<>"N" THEN 210
190 PRINT : PRINT : PRINT "BYE..."
200 END
210 FOR N=2 TO 12 : S(N) = 0
220 NEXT : GOTO 10
```







SECTION D
APPENDICES








Appendix A


























ASCII & CHR\$ Codes




















PRINT	CHR\$
	0
	1
	2
	3
	4
WHT	5
	6
	7
DISABLES SHIFT 	8
ENABLES SHIFT 	9
	10
	11
	12
RETURN	13
SWITCH TO LOWER CASE	14
	15
	16
CRSR 	17
RVS ON 	18
CLR HOME	19
INST DEL	20
	21
	22








PRINT	CHR\$
	23
	24
	25
	26
	27
RED 	28
CRSR 	29
GRN 	30
BLU 	31
SPACE	32
!	33
"	34
#	35
\$	36
%	37
&	38
•	39
(40
)	41
*	42
+	43
,	44
—	45








PRINT	CHR\$
.	46
/	47
0	48
1	49
2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57
:	58
;	59
<	60
=	61
>	62
?	63
@	64
A	65
B	66
C	67
D	68








PRINT	CHR\$
E	69
F	70
G	71
H	72
I	73
J	74
K	75
L	76
M	77
N	78
O	79
P	80
Q	81
R	82
S	83
T	84
U	85
V	86
W	87
X	88
Y	89
Z	90
[91
£	92
]	93
•	94
←	95
	96
	97
	98
	99
	100
	101
	102

PRINT	CHR\$
	103
	104
	105
	106
	107
	108
	109
	110
	111
	112
	113
	114
	115
	116
	117
	118
	119
	120
	121
	122
	123
	124
	125
	126
	127
	128
Orange	129
	130
	131
	132
f1	133
f3	134
f5	135
f7	136

PRINT	CHR\$
f2	137
f4	138
f6	139
f8	140
SHIFT/RETURN	141
SWITCH TO UPPER CASE	142
	143
	144
	145
	146
	147
	148
Brown	149
Lt. Red	150
Grey 1	151
Grey 2	152
Lt. Green	153
Lt. Blue	154
Grey 3	155
	156
	157
	158
	159
SPACE	160
	161
	162
	163
	164
	165
	166
	167
	168
	169
	170

PRINT	CHR\$
	171
	172
	173
	174
	175
	176
	177

PRINT	CHR\$
	178
	179
	180
	181
	182
	183
	184

PRINT	CHR\$
	185
	186
	187
	188
	189
	190
	191

CODES 192-223 SAME AS 96-127

CODES 224-254 SAME AS 160-190

CODE 255 SAME AS 126

Reserved Words

ABS	LIST	STEP
AND	LOAD	STOP
ASC	LOG	STR\$
ATN	MID\$	SYS
CHR\$	NEW	TAB
CLOSE	NEXT	TAN
CLR	NOT	THEN
CMD	ON	TIME
CONT	OPEN	TIMES\$
COS	OR	TO
DATA	OUT	USR
DEF	PEEK	VAL
DIM	POKE	VERIFY
END	POS	WAIT
EXP	PRINT	+
FN	PRINT#	-
FOR	READ	*
FRE	REM	/
GET	RESTORE	⇧
GET#	RETURN	>
GOSUB	RIGHT\$	=
GOTO	RND	<
IF	RUN	
INPUT	SAVE	
INPUT#	SGN	
INT	SIN	
LEFT\$	SPC	
LEN	SQR	
LET	STATUS	

Error Messages

BAD DATA -- The program received string data when numeric data was expected.

BAD SUBSCRIPT -- The array element that the program is trying to reference is beyond the range of values reserved in the DIM statement.

BREAK -- The **STOP** key was hit ending program execution.

CAN'T CONTINUE -- The command CONT was typed, and there is no program to continue. The program was just edited, or a line(s) was just added or deleted.

DEVICE NOT PRESENT -- The device specified by the OPEN, CLOSE, CMD, PRINT#, INPUT#, or GET# statement is not available. Either you do not have the hardware to support the device, or the device is disabled.

DIVISION BY ZERO -- The Computer is being asked to divide a number by 0. You may think of your Computer as the smartest thing going, but it is not capable of handling numbers of infinite value.

EXTRA IGNORED -- You typed in more items than called for by the INPUT statement so only the first few were used.

FILE NOT FOUND -- With files on tape, this means an END-OF-TAPE marker was found. With disk files, this means that a file by that name does not exist.

FILE NOT OPEN -- A file must be OPENed first before you can CLOSE, CMD, PRINT#, INPUT#, or GET#.

FILE OPEN -- You tried to OPEN a file using the number of a file that is already OPEN.

FORMULA TOO COMPLEX -- The manipulation of strings or formulas has gotten too complex or too long for the Computer.

ILLEGAL DIRECT -- You cannot use the INPUT statement in the direct mode, only within a program.

ILLEGAL QUANTITY -- The number used as the argument of either a function or a statement is not within the range allowed by the Computer.

LOAD -- A problem is preventing the LOADING of a program from tape.

NEXT WITHOUT FOR -- An attempt was made to RUN a program containing a FOR-NEXT loop, but the word "FOR" was missing. This error could also be caused by loops that were not correctly nested or by a variable name in a NEXT statement that does not correspond with one in a FOR statement.

NOT INPUT FILE -- You tried to INPUT or GET data from a file that was set up strictly as an OUTPUT file.

NOT OUTPUT FILE -- You tried to PRINT data to a file that was designated as an INPUT file only.

OUT OF DATA -- The Computer is being told to READ more items from the DATA statement than are available.

OUT OF MEMORY -- The amount of space left in the RAM is not enough for storing the program or the variables. This could also be caused by too many nested FOR-NEXT loops or too many GOSUBs.

OVERFLOW -- The Computer is unable to use a number because it's too large. The largest number allowed is 1.70141884E+38. An overflow condition can also be the result of mathematical calculations at either statement or command levels.

REDIM'D ARRAY -- The Computer is being told to DIMension a Matrix

after it has already been DIMensioned earlier in the same program. Using an array variable before DIMensioning the actual array causes that array to be automatically DIMensioned to ten elements. Trying to DIMension that array thereafter will cause this error.

REDO FROM THE START -- You typed in character data rather than numeric data during INPUT. To continue the program, type in the correct type of data.

RETURN WITHOUT GOSUB -- The Computer has read a RETURN statement and there is no corresponding GOSUB.

STRING TOO LONG -- An attempt was made to store more than 255 letters or characters in a string variable.

?SYNTAX ERROR -- The Commodore 64 does not recognize a statement. A command, statement, or function is misspelled or an operand is missing.

TYPE MISMATCH -- A number was used in place of a string or vice-versa.

UNDEF'D STATEMENT -- A branching statement such as GOTO or GOSUB called for a Line number that does not exist, or the Computer was told to RUN a non-existent Line number.

VERIFY -- The tape or disk program does not match the current program in memory.

Hex-to-Decimal Conversion Chart

HEX CODE	Most Significant Byte		Least Significant Byte	
	IV	III	II	I
0	0	0	0	0
1	4096	256	16	1
2	8192	512	32	2
3	12288	768	48	3
4	16384	1024	64	4
5	20480	1280	80	5
6	24576	1536	96	6
7	28672	1792	112	7
8	32768	2048	128	8
9	36864	2304	144	9
A	40960	2560	160	10
B	45056	2816	176	11
C	49152	3072	192	12
D	53248	3328	208	13
E	57344	3584	224	14
F	61440	3840	240	15

Hex-to-Decimal Conversion Chart
Decimal Value = IV + III + II + I

Appendix E

Screen Display Codes

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
@		0	U	u	21	*		42
A	a	1	V	v	22	+		43
B	b	2	W	w	23	,		44
C	c	3	X	x	24	—		45
D	d	4	Y	y	25	.		46
E	e	5	Z	z	26	/		47
F	f	6	[27	0		48
G	g	7	£		28	1		49
H	h	8]		29	2		50
I	i	9	↑		30	3		51
J	j	10	←		31	4		52
K	k	11	SPACE		32	5		53
L	l	12	!		33	6		54
M	m	13	“		34	7		55
N	n	14	#		35	8		56
O	o	15	\$		36	9		57
P	p	16	%		37	:		58
Q	q	17	&		38	;		59
R	r	18	,		39	<		60
S	s	19	(40	=		61
T	t	20)		41	>		62

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
?		63		U	85			107
		64		V	86			108
	A	65		W	87			109
	B	66		X	88			110
	C	67		Y	89			111
	D	68		Z	90			112
	E	69			91			113
	F	70			92			114
	G	71			93			115
	H	72			94			116
	I	73			95			117
	J	74	SPACE		96			118
	K	75			97			119
	L	76			98			120
	M	77			99			121
	N	78			100			122
	O	79			101			123
	P	80			102			124
	Q	81			103			125
	R	82			104			126
	S	83			105			127
	T	84			106			

Codes from 128-255 are reversed images of codes 0-127.

Appendix F

Musical Values

MUSICAL NOTE		OSCILLATOR FREQ		
NOTE	OCTAVE	DECIMAL	HI	LOW
0	C-0	268	1	12
1	C#-0	284	1	28
2	D-0	301	1	45
3	D#-0	318	1	62
4	E-0	337	1	81
5	F-0	358	1	102
6	F#-0	379	1	123
7	G-0	401	1	145
8	G#	425	1	169
9	A-0	451	1	195
10	A#-0	477	1	221
11	B-0	506	1	250
16	C-1	536	2	24
17	C#-1	568	2	56
18	D-1	602	2	90
19	D#-1	637	2	125
20	E-1	675	2	163
21	F-1	716	2	204
22	F#-1	758	2	246
23	G-1	803	3	35
24	G#-1	851	3	83
25	A-1	902	3	134
26	A#-1	955	3	187
27	B-1	1012	3	244
32	C-2	1072	4	48
33	C#-2	1136	4	112
34	D-2	1204	4	180

MUSICAL NOTE		OSCILLATOR FREQ		
NOTE	OCTAVE	DECIMAL	HI	LOW
35	D#-2	1275	4	251
36	E-2	1351	5	71
37	F-2	1432	5	152
38	F#-2	1517	5	237
39	G-2	1607	6	71
40	G#-2	1703	6	167
41	A-2	1804	7	12
42	A#-2	1911	7	119
43	B-2	2025	7	233
48	C-3	2145	8	97
49	C#-3	2273	8	225
50	D-3	2408	9	104
51	D#-3	2551	9	247
52	E-3	2703	10	143
53	F-3	2864	11	48
54	F#-3	3034	11	218
55	G-3	3215	12	143
56	G#-3	3406	13	78
57	A-3	3608	14	24
58	A#-3	3823	14	239
59	B-3	4050	15	210
64	C-4	4291	16	195
65	C#-4	4547	17	195
66	D-4	4817	18	209
67	D#-4	5103	19	239
68	E-4	5407	21	31
69	F-4	5728	22	96
70	F#-4	6069	23	181
71	G-4	6430	25	30
72	G#-4	6812	26	156
73	A-4	7217	28	49
74	A#-4	7647	29	223
75	B-4	8101	31	165
80	C-5	8583	33	135
81	C#-5	9094	35	134
82	D-5	9634	37	162
83	D#-5	10207	39	223
84	E-5	10814	42	62
85	F-5	11457	44	193
86	F#-5	12139	47	107

MUSICAL NOTE		OSCILLATOR FREQ		
NOTE	OCTAVE	DECIMAL	HI	LOW
87	G-5	12860	50	60
88	G#-5	13625	53	57
89	A-5	14435	56	99
90	A#	15294	59	190
91	B-5	16203	63	75
96	C-6	17167	67	15
97	C#-6	18188	71	12
98	D-6	19269	75	69
99	D#-6	20415	79	191
100	E-6	21629	84	125
101	F-6	22915	89	131
102	F#-6	24278	94	214
103	G-6	25721	100	121
104	G#-6	27251	106	115
105	A-6	28871	112	199
106	A#-6	30588	119	124
107	B-6	32407	126	151
112	C-7	34334	134	30
113	C#-7	36376	142	24
114	D-7	38539	150	139
115	D#-7	40830	159	126
116	E-7	43258	168	250
117	F-7	45830	179	6
118	F#-7	48556	189	172
119	G-7	51443	200	243
120	G#	54502	212	230
121	A-7	57743	225	143
122	A#-7	61176	238	248
123	B-7	64814	253	46

Sound Settings

SETTING VOLUME — SAME FOR ALL 3 VOICES														
VOLUME CONTROL	POKE 54296	Settings range from 0 (off) to 15 (loudest)												
VOICE NUMBER 1														
TO CONTROL THIS SETTING:	POKE THIS NUMBER:	FOLLOWED BY ONE OF THESE NUMBERS (0 to 15 ... or ... 0 to 255 depending on range)												
TO PLAY A NOTE	C	C#	D	D#	E	F	#	G	G#	A	A#	B	C	C#
HIGH FREQUENCY	54273 34	36	38	40	43	45	48	51	54	57	61	64	68	72
LOW FREQUENCY	54272 75	85	126	200	52	198	127	97	111	172	126	188	149	169
WAVEFORM	POKE 54276	TRIANGLE 17	SAWTOOTH 33		PULSE 65		NOISE 129							
PULSE RATE (Pulse Waveform)														
HI PULSE	54275	A value of 0 to 15 (for Pulse waveform only)												
LO PULSE	54274	A value of 0 to 255 (for Pulse waveform only)												
ATTACK/DECAY	POKE 54277	ATK4	ATK3	ATK2	ATK1	DEC4	DEC3	DEC2	DEC1					
		128	64	32	16	8	4	2	1					
SUSTAIN/RELEASE	POKE 54278	SUS4	SUS3	SUS2	SUS1	REL4	REL3	REL2	REL1					
		128	64	32	16	8	4	2	1					
VOICE NUMBER 2														
TO PLAY A NOTE	C	C#	D	D#	E	F	F#	G	G#	A	A#	B	C	C#
HIGH FREQUENCY	54280 34	36	38	40	43	45	48	51	54	57	61	64	68	72
LOW FREQUENCY	54279 75	85	126	200	52	198	127	97	111	172	126	188	149	169
WAVEFORM	POKE 54283	TRIANGLE 17	SAWTOOTH 17		PULSE 65		NOISE							

PULSE RATE															
HI PULSE	54282	A value of 0 to 15 (for Pulse waveform only)													
LO PULSE	54281	A value of 0 to 255 (for Pulse waveform only)													
ATTACK/DECAY		POKE	ATK4	ATK3	ATK2	ATK1	DEC4	DEC3	DEC2	DEC1					
	54284		128	64	32	16	8	4	2	1					
SUSTAIN/RELEASE		POKE	SUS4	SUS3	SUS2	SUS1	REL4	REL3	REL2	REL1					
	54285		128	64	32	16	8	4	2	1					
VOICE NUMBER 3															
TO PLAY A NOTE		C	C#	D	D#	E	F	F#	G	G#	A	A#	B	C	C#
HIGH FREQUENCY	54287	34	36	38	40	43	45	48	51	54	57	61	64	68	72
LOW FREQUENCY	54286	75	85	126	200	52	198	127	97	111	172	126	188	149	169
WAVEFORM		POKE	TRIANGLE		SAWTOOTH		PULSE		NOISE						
	54290		17		33		65		129						
PULSE RATE															
HI PULSE	54289	A value of 0 to 15 (for Pulse waveform only)													
LOW PULSE	54288	A value of 0 to 255 (for Pulse waveform only)													
ATTACK/DECAY		POKE	ATK4	ATK3	ATK2	ATK1	DEC4	DEC3	DEC2	DEC1					
	54291		128	64	32	16	8	4	2	1					
SUSTAIN/RELEASE		POKE	SUS4	SUS3	SUS2	SUS1	REL4	REL3	REL2	REL1					
	54292		128	64	32	16	8	4	2	1					



SECTION E

INDEX

A

Abbreviations 140
 ABS 178
 AND 256
 Arithmetic Functions 176
 Arrays 194
 ASCII 144
 Chart 326
 ASC 148
 ATN 188

B

Backup 63
 BASIC 47
BREAK Key 6
 Byte 55

C

C key 236
 Calculator Mode 51
 Caret 32,177
 Characters
 ASCII 145
 graphics 242
 CHR\$ 145
 CLR 205
CLR/HOME Key 89
 Clock
 setting 172
 TIMES\$ 172
 CLS 75
 Codes
 ASCII 144
 error 330
 graphics 242
 Comma 28
 Common Log 181
 Concatenate (+) 156
 Conditional Tests (IF-THEN) 42
 CONT 83

COS 185

Cursor 1

D

DATA 126
 Debugging 280
 Delete, **INST/DEL** Key 3
 DIM 204
 Directory 60,65
 Disk 2
 Disk Error 65
 Initialize 66
 Division 22

E

Editing 15
 END 12
 ERR 281
 Error Codes and Messages 281,282,330
 EXP 182
 Exponential notation 32
 Expressions
 logical 256
 numeric 51
 relational 40
 string 131

F

Flowcharting 274
 FOR-NEXT 68
 FORMAT 61
 FRE(0) 54
 Functions
 arithmetic 176
 defined 190
 integer 98
 intrinsic 176
 string 131
 trigonometric 184

G

GET 266
 GOSUB 113
 GOTO 40
 Graphics 242

H

Hex-to-Decimal Conversion Chart 333

I

IF-THEN 40
 INPUT 46,141
 Instring Routine 169
 INT 98,176
 Interpreter 48
 Inverse trigonometric functions 188

L

LEFT\$ 163
 LEN 155
 LET 26
 Line length 143
 Line Numbers 7
 LIST 11,81
 LOAD 58
 LOG 178
 Logical Operators 256

M

Matrix 199
 Memory Map 228
 MID\$ 163
 Modes
 Calculator 51
 Multi-Dimension Arrays 215
 Multiple Statement Lines 137

N

Natural Log 179
 Negation 22
 NEW 2
 NEXT 68
 optional 141
 NOT 260
 Numeric Variables 138

O

ON GOSUB 114
 ON GOTO 109
 Operators
 arithmetic 22
 logical 256
 relational 40
 OR 256
 Order of Operations 35,261

P

Parentheses 35
 PEEK 228
 POKE 228
 PRINT 3

 PRINT TAB 86
 Print Zones 28

Q

Quotation Marks 3

R

RANDOM 117
 READ 126
 Relational Operators 40
 REMark 11
 Reserved Words 329
 Resident Program 24

- RESTORE 147
RESTORE key 2
 RETURN 113
 RIGHTS\$ 163
 RND 117
 RUN 3,81
- S**
- SAVE 57
 Saving on Disk 61
 Scientific Notation
 (see Exponential Notation)
 Scratch 65
 Semicolon Key 29,47,88,140
 SGN 112
SHIFT Key 15
 SIN 184
 Sort 207
 Sound 248
 SPC 91
 SQR 176
 Statement
 conditional 42
 define 190
 program 6
 unconditional 44
 STEP 69
 STOP 82
 String
 arrays 220
 comparisons 150
 data 151
 variables 131,139
 STR\$ 162
 Subroutine 112
 Syntax Error 8
 SYS 263
- T**
- TAB 86
 TAN 187
- THEN 40
 TIMES\$ 172
 TO 68
 Trigonometric Functions 184
- U**
- Unconditional branching 44
 USR 263
- V**
- VAL 159
 Variables
 classifying 131
 names 25,138,139
 reserved words 329
 subscript 194
 Video Display
 clearing 75
 memory 229,239
 Verify 63
- + 22
 - 22
 * 22
 ^ 32,177
 / 22
 () 35
 > 40
 < 40
 = 40
 <> 40
 <= 40
 >= 40

Also Available From CompuSoft Publishing:

The BASIC Handbook <i>Encyclopedia of the BASIC language</i> ISBN 0-932760-05-8	\$19.95
Learning TRS-80 Model III BASIC <i>Also includes Model I</i> ISBN 0-932760-08-2	\$19.95
Learning IBM BASIC <i>for the Personal Computer</i> ISBN 0-932760-13-9	\$19.95
Learning TIMEX/Sinclair BASIC ISBN 0-932760-15-5	\$14.95
Learning TRS-80 Model 4/4P BASIC ISBN 0-932760-19-8	\$19.95
The TRS-80 Model 100 Portable Computer ISBN 0-932760-17-1	\$19.95
The IBM BASIC Handbook ISBN 0-932760-23-6	\$14.95
Learning IBM PC Disk BASIC ISBN 0-932760-25-2	\$17.95
Learning Apple II BASIC <i>Includes II Plus and IIe</i> ISBN 0-932760-24-4	\$14.95
Learning Microsoft BASIC <i>for the Macintosh</i> ISBN 0-932760-27-9	\$19.95

plus \$2.00 Postage and Handling — \$3.00 foreign orders

(California addresses add 6% sales tax)

COMPUSOFT® PUBLISHING
P.O. Box 19669, Dept. C1
San Diego, CA 92119

FREE

Update Information For

LEARNING COMMODORE 64 BASIC

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
P.O. Box 19669, Dept. C1
San Diego, CA 92119

\$14.95

COMPU^{SOFT}
LEARNING SERIES

LEARNING COMMODORE 64™ BASIC

The Commodore 64 has everything. It has color, sound, graphics - it even plays music. Often thought of as merely an inexpensive "game machine", the 64 is actually a powerful computer, capable of creating and running thousands of programs for business and education, as well as entertainment. The key to it all is BASIC, and learning the BASIC language from David Lien is not only easy, it's fun.

CompuSoft is proud to introduce this, the latest in Lien's best-selling series of BASIC books, a list that includes the undisputed standard reference encyclopedia of the language: *THE BASIC HANDBOOK*.

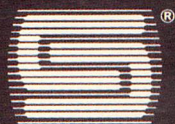
ABOUT THE AUTHOR

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 IBM BASIC*, *Learning Microsoft BASIC for the Macintosh*, *Learning Apple II BASIC*, *Learning TRS-80 Model 4/4P BASIC*, and the *Epson MX Series Printer Manuals*.

There are other books on BASIC for the Commodore 64; you may have picked some up. And you may have put them down again, more bewildered than before. If engineering jargon and "computerese" haven't taught you what you want to know, try the David Lien approach. It's a technique of "uncomplicating" technical subjects that he's been refining since 1976, when he wrote the official manual for the computer that started it all.

LEARNING COMMODORE 64 BASIC will turn you on to the real power of this surprisingly capable machine. There's a lot more to the Commodore 64 than games.



COMPU^{SOFT}
PUBLISHING



PRINTED IN USA

0-932760-22-8