

# EINSTEIN'S COMPUTER GUIDES

## EINSTEIN'S BEGINNERS' GUIDE TO THE COMMODORE 64<sup>TM</sup>

- 75 MODEL PROGRAMS
- ALL PROGRAMS EXPLAINED  
LINE BY LINE
- FULL DESCRIPTION OF SPECIAL  
FUNCTION KEYS

**JEFF EINSTEIN**  
HARCOURT BRACE JOVANOVICH

HBJ



---

**EINSTEIN'S**  
BEGINNERS' GUIDE TO  
THE

**COMMODORE 64™**

---



**EINSTEIN'S COMPUTER GUIDES**

---

**EINSTEIN'S**  
BEGINNERS' GUIDE TO THE  
**COMMODORE 64<sup>TM</sup>**

---

**JEFF EINSTEIN**

---

Books for Professionals  
Harcourt Brace Jovanovich, Publishers • San Diego • New York • London

## For Herm

Copyright © 1984 by Harcourt Brace Jovanovich, Inc.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy, recording, or any information and retrieval system without permission in writing from the publisher.

Requests for permission to make copies of any part of the work should be mailed to: Permissions, Harcourt Brace Jovanovich, Publishers, Orlando, Florida 32887.

Commodore 64™ is a trademark of Commodore Business Machines, Inc. This publication was prepared by Harcourt Brace Jovanovich, Inc., which is solely responsible for its content. It is not endorsed by any other organization.

Printed in the United States of America

Library of Congress Cataloging in Publication Data

Einstein, Jeff.  
Einstein's Beginners' guide to the Commodore 64.  
(Einstein's computer guides) (Books for professionals)  
Includes index.

I. Commodore 64 (Computer) I. Title. II. Title: Beginners' guide to the Commodore 64. III. Series: Einstein, Jeff. Einstein's computer guides. IV. Series: Books for professionals.

QA76.8.C64E35 1984 001.64'2 84-5752  
ISBN 0-15-600413-5

Designed by Karen Savary.

First Edition

A B C D E

## CONTENTS

---

Introduction	vii
1 Immediate, Program, and Edit Modes	1
2 Math Operators and Priorities	12
3 Variables	16
4 Conditional Statements and Relations	22
5 Operator Input	25
6 <b>FOR/NEXT</b> Loops	33
7 Random Number Generator	40
8 <b>READ, DATA, and RESTORE</b>	46
9 Subroutines	52
10 Logic	57
11 String Handling Functions	62
12 Arrays	72
13 <b>CHR\$, ASC, and PEEK and POKE</b>	84
14 Graphics	90
15 Sprite Graphics	104
16 Sound	112
17 Math and Trigonometric Functions	119
18 Memory, Report and Error Codes, and Applications	127
Appendix 1 Saving, Loading, and Printing	131
Appendix 2 Commodore 64 BASIC	142
Appendix 3 <b>CHR\$/ASC</b> Codes	147
Appendix 4 Error Messages	152
Appendix 5 <b>PEEK/POKE</b> Codes	154
Appendix 6 Musical Note Values	160
Appendix 7 Sample Exercise Answers	165
Glossary	181
Index	183





## INTRODUCTION

---

The good news: Computers don't confuse people.

The bad news: People confuse people. Most computer manuals and books contribute unwittingly to this confusion for one or both of the following reasons:

### 1. *Technical overkill*

- Some authors would rather showcase than share what they know, while others assume that readers have some prior knowledge of computers. Both types ignore and ultimately fail the reader.
- Einstein's *Beginners' Guide to the Commodore 64* delivers specifically what new Commodore owners most want and need—immediate on-screen results, with no excess technical jargon.

### 2. *Bad writing*

- Most programmers and computer scientists are not writers. Thus, even the noblest efforts to edify often end with confusion, leaving the reader adrift in a sea of split infinitives and dangling participles. Simply stated, bad writing is bad writing, no matter how much you know.
- Relax, I'm a writer. I never split infinitives or dangle my participle without just cause.

Einstein's *Beginners' Guide to the Commodore 64* will introduce you to the full range of your Commodore's capabilities, including the complete set of Commodore V2 BASIC language instructions. All commands, functions, and statements are demonstrated in easy, entertaining sample programs; each program is broken down and explained—line by line. Most chapters conclude with practice exercises that encourage you to apply what you've learned. A detailed appendix on saving and loading files on disk is included for those users struggling with the complexities of data storage and retrieval.

First consult your *Commodore 64 User's Guide* for proper hook-up procedures; then turn to Chapter 1, and please

- proceed at the pace most comfortable for you.

- don't be afraid to experiment (you won't hurt your Commodore). If you get an idea, go with it (you never know unless you try).
- don't ingest a chapter until you have digested the previous one.
- attempt to solve the chapter exercises.
- recommend this book to a friend.
- have fun!

JEFF EINSTEIN, President

Einstein's Automation Profiles, Inc.  
372 Fifth Avenue  
New York, NY 10018

WHAT YOU SEE IS WHAT YOU GET

# Immediate, Program, and Edit Modes

Power on! Your display should appear as follows:

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

On a color monitor the display will appear in light blue letters against a darker blue background with a light blue border. Consult Chapter 1 of your *Commodore 64 User's Guide* if you see anything else (on the screen, of course).

The blinking light blue square beneath the word **READY** is the *cursor*. It indicates where the next character or space will be **PRINTed** on the screen. *Press* the **CLR/HOME** key (near the upper-right corner of the keyboard), and the cursor jumps to the top-left corner of the screen—the “home” position. *Press* **SHIFT** and **CLR/HOME** at the same time and the entire display disappears, leaving the cursor back home by itself.

**note:** Keys to be operated in a specified order are separated by hyphens in this book. Thus, **SHIFT-CLR/HOME** tells you to press and hold the **SHIFT** key down, then press the **CLR/HOME** key, then release both.

- **CLR/HOME** positions the cursor in the home position.
- **SHIFT-CLR/HOME** clears the display and positions the cursor in the home position.

## GETTING THE JOB DONE FAST

---

### Immediate Mode

Type the following:

```
PRINT "HELLO (YOUR NAME) "
```

and press **RETURN**. Your display should appear as follows:

```
PRINT "HELLO (YOUR NAME) "  
HELLO (YOUR NAME)  
READY.
```

The computer **PRINT**ed HELLO (YOUR NAME) without the quotation marks.

- The Immediate Mode always executes the indicated command the moment you press **RETURN** (**PRINT**, in this instance).
- The computer will **PRINT** only those letters between quotation marks (*literal strings*). Try typing the same line without the quotation marks and see what happens.

Numbers do not require quotation marks in **PRINT** commands. Press **SHIFT-CLR/****HOME** to clear the display. Then *type*

```
PRINT 12345
```

Then press **RETURN**. Your display will appear as follows:

```
PRINT 12345  
12345  
READY.
```

### In Plain English . . .

The Immediate Mode executes any command once and then forgets the whole thing—perfect if you want to use your Commodore as a simple calculator. But most of us expect computers to remember.

**note:** Commodore BASIC provides abbreviated symbols for certain commands. It is not necessary to type the word **PRINT** each time it's required. You can use a question

mark symbol (?) instead; that is, ? 12345 yields the same results as **PRINT** 12345. A complete list of abbreviated command symbols can be found in your *Commodore 64 User's Guide*.

## DEFERRED GRATIFICATION

---

### Program Mode

In the Program Mode your Commodore 64 stores (remembers) your commands and relevant data in memory for retrieval at any time. First, *press* **SHIFT-CLR/HOME** to clear the display. Now *type* the following program lines (*press* **SHIFT-CLR/HOME** whenever **CLR/HOME** appears in this book), and be certain to *press* **RETURN** after completing each line. (**RETURN** enters the program line into the computer's memory.)

```
10 PRINT " (CLR/HOME) "
20 PRINT "HELLO (YOUR NAME) "
```

A little heart (a rather endearing symbol for a computer) should have appeared as you *pressed* **SHIFT-CLR/HOME** after the first quotation mark in line 10. If it didn't, *type* program line 10 again. (Don't forget the quotation marks.)

Now *type* the word **RUN**, then *press* **RETURN**.

Voila! Your display should appear as follows:

```
HELLO (YOUR NAME)
READY.
```

Here's what happened.

1. The command **RUN** followed by **RETURN** *executed* (or "ran") your program beginning at the first numbered program line.
2. Program line 10 cleared the display—**PRINT** "(CLR/HOME)".
3. Program line 20 **PRINTed** all characters between the quotation marks—HELLO (YOUR NAME).

In the Program Mode, it is important that you proceed as follows:

1. *Always* begin each program line with a line number in the range from 0 to 63,999. The computer executes your program, line by line, in the ascending order of the line numbers.
2. *Always* number your program lines in minimum increments of 10 (for example, 10, 20, 30, etc.) to allow future insertion of additional program lines between existing lines.

The command **RUN** followed by **RETURN** executes your program, starting with the very first program line.

3. *Always* follow every command and completed program line by *pressing* **RETURN**.

## **GREAT CAESAR'S GHOST!**

---

### **Edit Mode**

There's an editor in your Commodore 64 to accommodate the remote possibility that sooner or later you will want to amend or change a program line. *Type* **LIST** and *press* **RETURN**. Program lines 10 and 20 should appear on the display, followed by the word **READY**.

- **LIST** followed by **RETURN** does just that: **LISTs** your program.

The **LIST** commands are as follows (ln stands for line number):

<b>LIST</b>	<b>LISTs</b> the entire program.
<b>LIST ln</b>	<b>LISTs</b> only the requested line ln.
<b>LIST ln-ln</b>	<b>LISTs</b> all program lines from the first to the last line number inclusively.
<b>LIST ln-</b>	<b>LISTs</b> all program lines from the specified line number to the end of the program.
<b>LIST -ln</b>	<b>LISTs</b> all program lines up to and including the line number specified.

Now *retype* line 20, without your name, as follows:

```
20 PRINT "HELLO"
```

Next *press* **RETURN**. Has anything changed in your program listing? *Type* **LIST**, then *press* **RETURN**. The new line 20 has replaced the old one. Unfortunately, retyping

entire program lines can sometimes strain your patience (not to mention your fingers). There's another, sometimes quicker, way to edit. Find the cursor control keys located in the lower-right corner of the keyboard. Their functions are as follows:

<b>UP/DOWN</b> cursor	moves the cursor down.
<b>SHIFT-UP/DOWN</b> cursor	moves the cursor up.
<b>LEFT/RIGHT</b> cursor	moves the cursor right.
<b>SHIFT-LEFT/RIGHT</b> cursor	moves the cursor left.

1. Press **SHIFT-UP/DOWN** twice to position the cursor over the 2 in program line 20.
2. Now depress the **LEFT/RIGHT** cursor key until the cursor is positioned directly over the H in Hello.
3. Press **INST/DEL** (upper-right corner of the keyboard) to **DELETE** the quotation mark.
  - **INST/DEL** behaves as a destructive backspace and **DELETES** any character to its immediate left.
4. Now press **RETURN** to re-enter the amended line 20 into the computer's memory. The cursor should be located directly over the R in the word READY. Type **RUN** (directly over READY), then press **RETURN**. The following prompt will appear:

```
?UNDEF'D STATEMENT ERROR
```

This indicates that your computer has encountered an undefined command. READY has become RUNDY, and not even Webster's can define a RUNDY. Now type **RUN** again and press **RETURN**. The computer will return the number 0 because you **DELETED** the first quotation mark of the literal string.

5. Type **LIST** and press **RETURN** to restore your program **LISTing**. Now press **SHIFT-UP/DOWN** twice to position the cursor over the 2 in program line 20.
6. Depress the **LEFT/RIGHT** cursor key until the cursor is positioned again over the H in HELLO. Press **SHIFT-INST/DEL** to insert a blank space. Now retype the missing quotation mark.
  - **SHIFT-INST/DEL** inserts a space anywhere in a program line.

In fact, you might as well go ahead and insert your name after the word HELLO. Oh no! Each subsequent **LEFT/RIGHT** cursor keystroke leaves a horrid little inverse bracket, while the **SHIFT-LEFT/RIGHT** cursor leaves an inverse vertical

line. The **UP/DOWN** and **SHIFT-UP/DOWN** cursors leave inverse Q's and dots, respectively. **STOP THE PRESSES!**

- The cursor keys react as graphics keys while editing literal strings (characters within quotation marks). *Press RETURN* once; then leap right back up again with the **SHIFT-UP/DOWN** cursor to edit line 20 as before:

```
20 PRINT "HELLO (YOUR NAME) "
```

**RETURN** allows you to use cursor keys within literal strings.

Finished? Now *press RETURN* twice, once to re-enter amended program line 20 back into memory, and again to move the cursor past the word **READY**. The second **RETURN** will generate another prompt:

```
?OUT OF DATA ERROR
```

Nothing to worry about for now. You will doubtless encounter several different error messages in the course of this book. A complete list of error messages can be found in Appendix 4.

Now let's add another program line to enliven things a bit. *Type*

```
30 GOTO 20
```

Then *press RETURN*. *Type LIST* and *press RETURN*. Your entire program should appear as follows:

```
10 PRINT " (CLR/HOME) "
20 PRINT "HELLO (YOUR NAME) "
30 GOTO 20
READY.
```

Now *type RUN* and *press RETURN*.

Stand back! **HELLO (YOUR NAME)** shoots down the left border of the screen. *Press RUN/STOP* (far-left key, middle row). Your program stops (or **BREAKS**) with the following prompt:

```
BREAK IN 20
READY.
```

- In most instances, **RUN/STOP** interrupts (or **BREAKS**) your program.
- The **BREAK IN 20** prompt indicates that your program was interrupted while executing line 20.



Now *type* **CONT** and *press* **RETURN**. Your program **CONT**inues where it left off.

- **CONT** instructs the computer to **CONT**inue with the program execution.

This time, *press* **RUN/STOP** and **RESTORE** (just below the **INST/DEL** key) simultaneously to interrupt the program. Not only does the program stop, but the screen clears with the word **READY** in the upper-left corner just above the cursor.

- In the event that **RUN/STOP** fails to **BREAK** your program, the more authoritative **RUN/STOP-RESTORE** usually will do it—without injuring your program.
- **RUN/STOP-RESTORE** always clears (**RESTORE**s) the display and positions the cursor in the top-left corner.

You have only one legal, although drastic, alternative if **RUN/STOP-RESTORE** fails to halt your program: Turn the computer off. Doing so, however, erases your entire program and will most likely ruin your day.

*Type* **LIST**, then *press* **RETURN**. Your program **LIST**ing reappears unscathed.

The **GOTO 20** statement in line 30 causes the program to jump back to program line 20. To put it simply: the **GOTO** command in program line 20 creates a *loop*. The program loops back to line 20 each time line 30 is encountered.

To **DELETE** program line 30 (or any other program line) simply *type* 30 and *press* **RETURN**. Skeptical? *Type* **LIST**, then *press* **RETURN**. Notice that program line 30 no longer exists.

**COPY**ing a program line is just as simple. Use the **UP/DOWN** cursor key to position the cursor over the 2 in line 20. *Type* the number 3 directly over the 2, then *press* **RETURN** until the cursor is past the word **READY** (remember **RUNDY**). Now *type* **LIST** and *press* **RETURN**. Line 30 is an exact duplicate of line 20. Hence, no need to bend your fingers or your brain.

*Retype* line 30 as before:

```
30  GOTO 20
```

Then *press* **RETURN** to enter it into memory. *Type* **LIST** and *press* **RETURN**. The most recent line 30 has replaced the old, and your program appears as before:

```
10  PRINT " (CLR/HOME) "  
20  PRINT "HELLO (YOUR NAME) "  
30  GOTO 20
```

Likewise, you'll have no problem inserting a new line into your program. *Type* line 15 as follows:

```
15  END
```

Then *press RETURN*. Now *type LIST* and *press RETURN*. Line 15 jumps right into the program between lines 10 and 20—exactly where it belongs. *Type RUN* and *press RETURN*. The display clears and the word **READY** appears.

- **END** stops the program execution by indicating where a BASIC program is to **END**.

*Type* 15 and *press RETURN* to **DELETE** program line 15. Now *type LIST* and *press RETURN*.

**note:** The preceding hints and suggestions are simply that. They are not intended to be a comprehensive guide to editing. As with anything else, your ability will expand as you become more experienced. Experiment with different keystroke combinations until you develop your own technique. There is no single right or wrong method. What works is right; what doesn't, isn't.

## **FORGET MY GRAMMAR, HOW DO I LOOK?**

---

### **Punctuation**

Punctuation improves the sound and readability of written communication by helping us determine the placement and duration of pauses. But your Commodore 64 is a vain creature, employing punctuation in a slightly different capacity: to determine the amount of blank space between **PRINT** statements. Stated otherwise—to look better.

#### **Comma**

Use the appropriate cursor keys to *type* a comma directly after the final quotation mark in line 20:

```
20 PRINT "HELLO (YOUR NAME) ",
```

*Press RETURN* as necessary to enter amended line 20 into memory and move the cursor past **READY**. Now *type RUN* and *press RETURN*.

Your display should appear as two columns of **HELLO (YOUR NAME)**—unless your name consists of fewer than four letters, in which case you will see four columns. *Press RUN/STOP* to **BREAK** the program.

- Each horizontal line of the Commodore 64 display is divided into 4 zones of 10 columns (characters) each, and each comma advances the **PRINT** statement to the beginning of the next available zone.

## Semicolon

**LIST** your program and change the comma at the end of line 20 to a semicolon:

```
20 PRINT "HELLO (YOUR NAME)";
```

Press **RETURN**, *type* **RUN**, and *press* **RETURN** again.

This time, each **HELLO (YOUR NAME) PRINT** statement follows directly on the heels of the previous one. *Press* **RUN/STOP** to **BREAK** the program.

- A semicolon advances each **PRINT** statement by one space.

*Type* the following in Immediate Mode (i.e., without a line number):

```
PRINT "A"; "B"; "C"
```

and *press* **RETURN**. ABC will appear. Each semicolon now performs two functions:

- It advances each subsequent **PRINT** statement one space, as before.
- It allows you to combine separate **PRINT** statements within a single program line.

**LIST** your program again. Add a space between the last letter of your name and the final quotation mark in line 20:

```
20 PRINT "HELLO (YOUR NAME) sp";
```

**note:** Each user-inserted space next to quotation marks is indicated by an “sp” throughout this book.

*Press* **RETURN** to enter amended line 20 into memory, *type* **RUN**, then *press* **RETURN** again. Note how much better it looks with the space. *Press* **RUN/STOP** to **BREAK** your program; then *type* **LIST** and *press* **RETURN**.

## Colon

*Type* 30 and *press* **RETURN** to **DELETE** line 30. Now add a colon, followed by **GOTO** 20, to the end of line 20:

```
20 PRINT "HELLO (YOUR NAME) sp"; : GOTO 20
```

*Press* **RETURN**, *type* **RUN**, and *press* **RETURN** again. Your program **RUNs** exactly as before, even though it has one less program line. Now line 20 **PRINTs** **HELLO (YOUR NAME)** and loops itself with **GOTO 20**. *Press* **RUN/STOP** to **BREAK** your program.

- A colon combines separate command statements in the same program line.

**note:** Don't lose your head. Too many command statements in one program line can make subsequent editing a nasty chore. The limit is 239 characters.

## TAB

**LIST** your program and edit line 20 to appear as follows:

```
20 PRINT TAB(15); "HELLO (YOUR NAME)": GOTO 20
```

Press **RETURN**, type **RUN**, and press **RETURN** again. Now HELLO (YOUR NAME) runs in a single column down the middle of the screen. Press **RUN/STOP** to **BREAK**.

Keep the following in mind when using **TAB**:

1. As on a typewriter, **TAB** positions the **PRINT** display at the column indicated by the adjacent number in parentheses (column 15 in program line 20).
2. **TAB** statements are always separated by a semicolon from the characters to be **PRINTed** (numbers or literal strings).
3. The Commodore 64 display is 40 columns wide (0-39).

## SPC

Now **LIST** your program and amend line 20 again as follows:

```
20 PRINT SPC(15); "HELLO (YOUR NAME)": GOTO 20
```

Press **RETURN** to enter line 20 into memory. Now type **RUN** and press **RETURN** again. Your screen should appear exactly as before, with a single column of HELLO (YOUR NAME) running down the middle. Press **RUN/STOP** to **BREAK**.

Note these special features of **SPC**:

1. **SPC** starts at the current **PRINT** position and spaces over the number of columns indicated in parentheses.
2. **TAB** always **PRINTs** at the absolute column number indicated, while **SPC** merely spaces over the number of columns indicated from the current **PRINT** position.

**note:** I'm through telling you to press **RETURN** after everything. From now on, you're on your own.

## CHAPTER 1 REVIEW

---

CLR/HOME	Line numbers
Colon	<b>LIST</b>
Comma	Literal strings
<b>CONT</b>	<b>PRINT</b>
Copy program line	Program Mode
Cursor	Punctuation
Cursor keys	<b>RESTORE</b>
Delete program line	<b>RETURN</b>
Edit Mode	<b>RUN</b>
Error messages	<b>RUN/STOP—BREAK</b>
<b>GOTO</b> loops	Semicolon
Immediate Mode	<b>SPC</b>
Insert copy line	<b>TAB</b>
<b>INST/DEL</b> and <b>SHIFT-INST/DEL</b>	

---

## PRACTICE EXERCISES

---

*Sample answers to all exercises appear in Appendix 7.*

- 1-1. Write a program to **PRINT** your name, address, and phone number in the middle of the screen (as on a business card).
  - 1-2. Write a program to **PRINT HELLO** in three separate columns down the screen. (*Hint:* Use a **GOTO** loop.)
  - 1-3. Write a program to **PRINT** your name once at line 1, column 10; again at line 11, column 10; and once more at line 21, column 10.
-

---

## Math Operators and Priorities

Your Commodore 64 takes to numbers like a duck takes to water. In fact, Immediate Mode allows you to use your computer as a calculator. Try the following:

**PRINT** 999

and **RETURN**. (Remember, don't use line numbers in Immediate Mode or quotation marks for numbers.) Your computer will **PRINT** the number 999 for you.

You can **PRINT** math expressions as well. Try this:

**PRINT** 999 + 1

and **RETURN**. Bingo, 1000.

### SUMMING UP

---

## Standard Math Operators

The *standard math operators* are

Symbol	Definition
-	Subtraction
+	Addition
*	Multiplication
/	Division
↑	Raised to the power of
-	Negation (negative number)

Try these simple expressions in Immediate Mode:

```
PRINT 25-10000
PRINT 25+10000
PRINT 10*10
PRINT 25/4
PRINT 10↑2
PRINT 10-(-2) (include the parentheses)
```

## FIRST THINGS FIRST

### Priority

Consider the compound expression  $5 + 5 * 8$ . Is the answer 45 or 80? Which operation do we perform first, addition or multiplication? There's no way to know unless we first set our priorities. Your Commodore 64, however, always knows which operation to perform first and never confuses priorities. Highest to lowest (first to last), the priorities are

-	Negation
↑	Raised to the power of
* or /	Multiplication or division (done in order from left to right)
+ or -	Addition or subtraction (done in order from left to right)

Thus in the example  $5 + 5 * 8$ ,  $5 * 8 = 40$  and  $5 + 40 = 45$ .

Parentheses can alter the operator priority. Try this:

```
PRINT 10+10*5
```

Your Commodore returns the number 60 (multiplying first). Now try this:

```
PRINT (10+10)*5
```

The computer returns 100. The difference is that the addition was done first this time.

- Expressions within parentheses are always worked out first.

- Parentheses can even be nested within parentheses (expressions within nested parentheses are worked out before expressions within outer parentheses). For example, `PRINT 10 * ((10 + 10)/5)` is evaluated as  $10 * (20/5)$  which equals 40.

**note:** Remember that parentheses, like quotation marks, come in pairs. Odd numbers of parentheses will result in an error message.

## **ALL NUMBERS ARE NOT CREATED EQUAL**

### **Numeric Notation**

Your Commodore uses a variety of notational formats to specify numeric values. First, the computer recognizes two types of numbers, “integers” and “real numbers.” *Integers*, or whole numbers, are used in counting. They have no fractional parts and no decimal point; they may be positive or negative. *Real numbers* are used in measurement. They are expressed in a decimal format: 2.34 is a real number, as is  $-2.0$ .

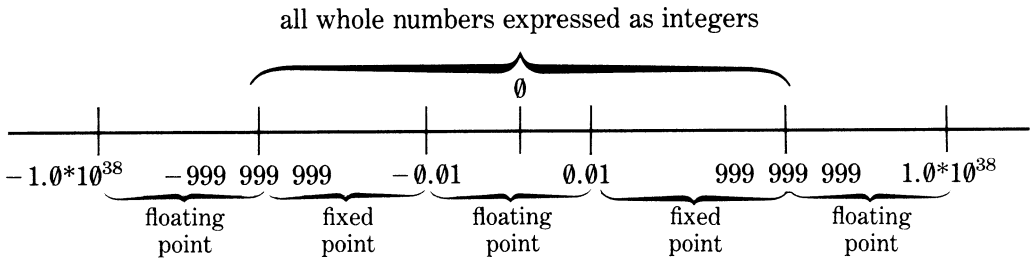
Everybody uses *fixed point* or *standard notation*: It’s good for ordinary use in most everyday applications and is the form of numbers you see in most magazines, newspapers, and books. Scientists, engineers, and computer operators who deal with very large and very small numbers—and thus eschew the ordinary—use *floating point* or *scientific notation*. This notational system may appear somewhat complicated at first glance—but don’t panic. You’ll get the hang of it in no time. Follow—

Any number can be expressed as a multiple of some power of 10: 100 is  $10^2$ , 1000 is  $10^3$ , 0.1 is  $10^{-1}$ , and 0.001 is  $10^{-3}$ , and so on. For example, 1234 can be written as  $1.234 \times 1000$ , or  $1.234 \times 10^3$ . To use this number on your Commodore, you replace the “ $\times 10$ ” part of this expression with E (exponent) and enter 1.234E+3 (or simply 1.234E3). Similarly, you could enter 0.001234 ( $1.234 \times 10^{-3}$ ) as 1.234E-3 (negative exponents must be specified with the minus sign).

Your Commodore can use all three systems—so how does it tell which one to employ? Easy! It makes the decision by gauging the size of the number as follows:

<u>Notational System</u>	<u>Number Range</u>
Integer	-999 999 999 ↔ 999 999 999
Fixed Point	-999 999 999 ↔ -0.01 0.01 ↔ 999 999 999
Floating Point	-1.0*10 ↑ 38 ↔ -999 999 999 -0.01 ↔ 0.01 999 999 999 ↔ 1.0*10 ↑ 38





- The largest number your Commodore can store is  $\pm 1.0 \times 10 \uparrow 38$ . If you try to enter a larger number, you'll generate the prompt: ?OVERFLOW ERROR.
- The smallest number your Commodore can store is  $\pm 1.0 \times 10 \uparrow -38$ . Anything smaller will **PRINT** the number 0.
- Your Commodore can accommodate numbers of up to 10 digits, but it will automatically round off all numbers at the ninth digit, based on the value of the tenth: If the tenth digit is greater than or equal to 5, the ninth digit is rounded upward; if less than 5, downward.

The following are examples of **PRINT** commands and the output display they generate:

<u>COMMAND</u>	<u>DISPLAY</u>	<u>REASON</u>
<b>PRINT</b> +5	5	plus signs (+) are optional
<b>PRINT</b> -5	-5	negative signs (-) are required
<b>PRINT</b> 10.490	10.49	trailing zeros are deleted
<b>PRINT</b> 1 * 10 $\uparrow$ 54	?OVERFLOW ERROR	number too large
<b>PRINT</b> -1 * 10 $\uparrow$ -54	0	number less than $\pm 1 \times 10 \uparrow -38$
<b>PRINT</b> 1.234567876543	1.23456788	ninth digit rounded off
<b>PRINT</b> 1400000000	1.4E + 09	number larger than 999 999 999
<b>PRINT</b> -0.0004	-4E - 04	number between -0.01 and +0.01

You should try **PRINT** commands with your own numbers until you are comfortable with these number formats.

## CHAPTER 2 REVIEW

Addition	? OVERFLOW ERROR
Division	Parentheses
Fixed point	Priority
Floating point	Raised to the power of
Integers	Real numbers
Multiplication	Scientific notation
Negation	Subtraction
Operators	

## Variables

Let's consider for a moment how your Commodore 64 stores numeric information. First *type* **NEW** and *press* **RETURN** to clear your computer's memory. Then *type* the following program:

```
10 REM VARIABLES #1
20 A = 1
30 B = 2
40 C = 3
50 PRINT " (CLR/HOME) "; "A = "; A; TAB (10); "spB = "; B;
    TAB (20); "spC = "; C
```

and **RUN**.

Your **PRINT** display should appear as follows:

```
A = 1    B = 2    C = 3
```

Now **LIST** your program.

### In Plain English . . .

```
10 REM VARIABLES #1
```

**REM** is short for **REMark**—a label or reminder. A **REM** statement means something to you only and is totally ignored by the computer. The **REM** statement in line 10 simply helps you remember the program name.

```
20 A = 1
30 B = 2
40 C = 3
```

These lines assign numeric values of 1, 2, and 3 to variables A, B, and C, respectively. A *variable* is a memory cell that stores a numeric value, either a floating point decimal

value or an integer value (e.g., 15.5 or 15). In your Commodore, recognized variable names have only one or two characters. The first character must be a letter (A to Z), but the second character can be either a letter or a number (alphanumeric). But in fact, variables may have more than two alphabetic characters, although your Commodore recognizes only the first two letters of any variable name. Hence, the variable AB and the variable ABNORMAL are both interpreted as the same variable—AB.

**note:** You can work exclusively with integers by using a percent symbol (%) after the variable name. For example,  $X\% = 4$  (not 4.0). The range of permissible integer values is from -32,768 to 32,767.

```
50 PRINT " (CLR/HOME) "; "A = "; A; TAB (10) ; "spB = "; B;
    TAB (20) ; "spC = "; C
```

Line 50 first clears the display (CLR/HOME), then PRINTs the assigned names and values of variables A, B, and C. Note the blank spaces within the literal strings ("spB = " and "spC = ") to tidy the PRINT display. Try the same line without the spaces to see the difference. Note also how line 50 combines several PRINT statements into one.

**note:** The variables themselves do not require quotation marks in PRINT statements. PRINT "A =" returns A = . PRINT A (without the quotation marks) returns the value of variable A.

Variables can be used to store mathematic expressions. Add the following line to VARIABLES #1:

```
60 PRINT: PRINT "A + B + C = "; A + B + C; TAB (15) ; "A * B * C = "; A * B * C
```

and RUN. Your display should now appear as follows:

```
A =    1    B =    2    C =    3
A + B + C = 6    A * B * C = 6
```

### In Plain English . . .

Line 60 PRINTs one empty line to tidy the display, PRINTs the literal string "A + B + C =", PRINTs the result of the expression A + B + C (that is, 6), PRINTs the literal string "A \* B \* C =" at TAB 15, and finally PRINTs the result of the expression A \* B \* C (that is, 6).

Now let's check the value of your variables in Immediate Mode. *Type*

```
PRINT A and press RETURN. PRINT A PRINTs the value 1.
PRINT B and press RETURN. PRINT B PRINTs the value 2.
PRINT C and press RETURN. PRINT C PRINTs the value 3.
```

Your variables are still intact in the computer's memory and will remain so unless or until you

1. type **NEW** and press **RETURN**. (Don't do this now!)
  - **NEW** erases the entire program and all variables from memory.
2. assign different values to your variables; or
3. turn off the power.

Now add the following lines to **VARIABLES #1**:

```
70 A=B
80 C=A
90 PRINT:PRINT "A=";A;TAB(10);"spB=";B;TAB(20);"spC=";C
100 PRINT:PRINT "A+B+C=";A+B+C;TAB(15);"A*B*C=";A*B*C
```

and **RUN**. The following display should result:

```
A= 1      B= 2      C= 3
A+B+C= 6      A*B*C= 6
A= 2      B= 2      C= 2
A+B+C= 8      A*B*C= 8
```

### In Plain English . . .

```
70 A=B
80 C=A
```

Line 70 reassigns variable A to equal the value of variable B. Line 80 then reassigns variable C to equal variable A. So now A=2, B=2, and C=2.

```
90 PRINT:PRINT "A=";A;TAB(10);"spB=";B;TAB(20);"spC=";C
100 PRINT:PRINT "A+B+C=";A+B+C;TAB(15);"A*B*C=";A*B*C
```

Line 90 now **PRINTs** the newly assigned values of variables A, B, and C.

Line 100 is an exact copy of line 60, except the expression A\*B\*C now equals 8—to reflect the newly assigned values of each variable.

Now let's check your variables again in Immediate Mode. *Type*

**PRINT** A and *press* **RETURN**. **PRINT A** **PRINTs** the value 2.

**PRINT B** and *press RETURN*. **PRINT B PRINTS** the value 2.  
**PRINT C** and *press RETURN*. **PRINT C PRINTS** the value 2.

Variables A, B, and C now reflect their newly assigned values.

**note:** **RUN** sets the values of all variables to zero and starts program execution from the first line, but **GOTO** executes from any line and leaves the variables intact. For example, *type GOTO 90* in Immediate Mode, then *press RETURN*. **VARIABLES #1** now begins at line 90.

## STRUNG OUT

---

### String Variables

*String variables* are similar to numeric variables with one major difference. Numeric variables store numeric values, whereas string variables store literal string values (characters between quotation marks). For an example of this, *type* and **RUN** the following program:

```
NEW
10 REM INEVITABLE
20 PRINT " (CLR/HOME) "
30 A$ = "DEATH"
40 B$ = "spAND TAXES"
50 PRINT "A$ = "; A$, "B$ = "; B$
60 PRINT: PRINT "A$ + B$ = "; A$ + B$
```

And **RUN**. Your display should appear as follows:

```
A$ = DEATH   B$ =  AND TAXES
A$ + B$ = DEATH AND TAXES
```

**note:** All string variables end in a dollar sign (\$) so that the computer can distinguish them from numeric values.

#### In Plain English . . .

```
10 REM INEVITABLE
20 PRINT " (CLR/HOME) "
```

The **REM** statement in line 10 merely states the obvious.

Line 20 clears the display.

```
30 A$ = "DEATH"
40 B$ = "spAND TAXES"
```

Line 30 sets string variable A\$ to equal the literal string "DEATH".

Line 40 sets string variable B\$ to equal the literal string "spAND TAXES".

```
50 PRINT "A$ = "; A$, "B$ = "; B$
60 PRINT: PRINT "A$ + B$ = "; A$ + B$
```

Line 50 **PRINTs** A\$ = and the value of string variable A\$, then **PRINTs** B\$ = followed by the value of string variable B\$.

Line 60 **PRINTs** one empty line to tidy the display, then it **PRINTs** A\$ + B\$ = followed by the sum of string variable A\$ and string variable B\$ (DEATH AND TAXES). Adding string variables is called *concatenation*. Other math operators will not work with string variables for obvious reasons (you simply cannot multiply or divide DEATH by TAXES).

Now add the following lines to your program:

```
70 B$ = A$
80 PRINT: PRINT "WHEN B$ = A$ IT FOLLOWS THATsp"; A$; " = "; B$
```

and **RUN**. Your display should now appear as follows:

```
A$ = DEATH   B$ =  AND TAXES
A$ + B$ = DEATH AND TAXES
WHEN B$ = A$ IT FOLLOWS THAT DEATH = DEATH
```

### In Plain English . . .

```
70 B$ = A$
80 PRINT: PRINT "WHEN B$ = A$ IT FOLLOWS THATsp"; A$; " = "; B$
```

Line 70 sets string variable B\$ to equal string variable A\$.

Line 80 **PRINTs** WHEN B\$ = A\$ IT FOLLOWS THAT—the literal string—followed by DEATH = DEATH, which is the value of string variable A\$ followed by = and the newly assigned value of string variable B\$. True enough, I suppose.

## CHAPTER 3 REVIEW

---

Concatenation

Floating-point decimal variable

Integer variable

**NEW**

**REM**

String variables

Variable

---

## PRACTICE EXERCISES

---

- 3-1. Write a program to determine the area (in square inches) of rectangle X if side A is 20 inches long and side B is 25 times longer than side A. Assign variables to sides A and B.
  - 3-2. Assume that your car gets 25 miles per gallon of gasoline (nice car!). Now assume that you must drive 555 miles to pick up your wife at her mother's house. (No, you can't leave her there, and no, she can't take a bus.) The gasoline will cost you \$1.15 per gallon. Write a program to determine how many gallons of gas the round trip would require and how much it would cost. Assign variables to the values 25, 555, and 1.15.
  - 3-3. Write a program to **PRINT YOUR NAME + YOUR LOVER'S NAME** by assigning each name to a different string variable and then concatenating them.
-

IF WISHES WERE PORSCHEs, THEN THE  
DISADVANTAGED WOULD DRIVE

## Conditional Statements and Relations

Let's take a look at how your Commodore 64 makes decisions. For example, you've just sent your teenage son to college. Each of his monthly letters includes a request for \$75. Naturally, you have no money and must borrow from your savings (\$1500). How many months of relative serenity will you enjoy before you go broke and your son returns home? *Type* in the following program to find out, but don't **RUN** it yet.

```

0  REM HOMEWARD BOUND
10  PRINT " (CLR/HOME) "
20  M=0
30  P=75
40  S=1500
50  M=M+1
60  S=S-P
70  PRINT "AFTER";M;" spMONTHS, YOUR SAVINGS=";S
80  GOTO 50

```

**In Plain English . . .**

```

0  REM HOMEWARD BOUND
10  PRINT " (CLR/HOME) "

```

Catchy title, eh? Line 10 clears the display.

```

20  M=0
30  P=75
40  S=1500

```

Line 20 sets variable **M** equal to 0. Using *mnemonics* (e.g., **M** for months, **P** for payments, **S** for savings, etc.) helps minimize confusion in your own mind when many variables are assigned.

Line 30 sets **P** (for payments) equal to 75.



Line 40 sets S (for savings) equal to 1500.

```
50 M=M+1
60 S=S-P
```

Line 50 adds 1 to the old value of M with each loop and stores the new total in M. So, in essence, it's counting for you.

Line 60 subtracts 75 (P) from your savings (S) with each loop and each time stores the new total in S.

```
70 PRINT "AFTER"; M; "spMONTHS, YOUR SAVINGS = "; S
80 GOTO 50
```

Line 70 **PRINTs** AFTER (the current value of M) MONTHS, YOUR SAVINGS = (the current value of S).

Line 80 is the loop that jumps back to line 50. Each loop adds 1 to the value of M (line 50), subtracts 75 from the total savings (line 60), and **PRINTs** the display (line 70).

Now **RUN** the program. The months add up while your savings dwindle. But wait! Your savings suddenly dip below 0! Not even your teenage son can spend that much money! Press **RUN/STOP** to **BREAK** the program! What happened? **LIST** your program and let's take a look. There must be a way to keep the creditors away from your door.

Lines 50–80 form an *endless loop*, meaning that the program could go on forever. To remedy this, each loop requires a *conditional statement* for the computer to know when to stop or exit and move on to something else. The symbols used to express *conditional relations* are

<u>Symbol</u>	<u>Definition</u>
=	Equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
<>	Not equal to

A conditional statement for **HOMEWARD BOUND** might read “if my savings dip below 0, please send my son home and notify me.” This could be programmed by first amending line 80 as follows:

```
80 IF S>0 THEN 50
```

and then adding these two lines

```

90 PRINT:PRINT:PRINT "AFTER";M;"spMONTHS,
   YOUR SON IS HOME... "
100 PRINT:PRINT "AND YOU ARE BROKE"

```

Now **RUN**.

**In Plain English . . .**

```
80 IF S>0 THEN 50
```

Line 80 now tests the value of S with each loop (a *conditional loop*) and will jump back to line 50 as long as your savings (S) are greater than 0. In other words, **IF** S is greater than 0, **THEN** jump back to program line 50 (**GOTO** is implied). Or, more simply, **IF** the test is true, **THEN** loop back to line 50.

**IF** the test is *not* true (S is not greater than 0), **THEN** line 80 is ignored and lines 90 and 100 **PRINT** the bad news.

## CHAPTER 4 REVIEW

---

Conditional loop	Endless loop
Conditional relation	<b>IF/THEN</b>
Conditional statement	

---

## PRACTICE EXERCISES

---

- 4-1. Suppose you opened a savings account in 1980 with \$500 dollars at 11% annual interest. Write a program to **PRINT** your yearly balance (savings plus interest) for the next 15 years. Use a conditional statement to end the program at the right year.
  - 4-2. Now amend the above program to **PRINT** only the year in which your total savings surpass \$2000.
-

## Operator Input

You may at times want to talk back to your computer (quite frequently if you're anything like me). But how do you make it listen? Try the following program to find out how to **INPUT** both numbers and literal strings:

```

0 REM DON'T GET PERSONAL
10 PRINT " (CLR/HOME) "; "WHAT'S YOUR NAME? "
20 INPUT N$
40 PRINT " (CLR/HOME) "; "HELLO, sp"; N$
50 PRINT:PRINT "HOW OLD ARE YOU, sp"; N$; "? "
60 INPUT A
80 PRINT " (CLR/HOME) "; A; "?!?! "
90 PRINT:PRINT "YOU DON'T LOOK A DAY OVER"; A-1

```

Now **RUN** the program and follow the instructions below.

### In Plain English . . .

```

0 REM DON'T GET PERSONAL
10 PRINT " (CLR/HOME) "; "WHAT'S YOUR NAME? "

```

Line 0 names the program.

Line 10 clears the screen, then asks WHAT'S YOUR NAME?

```

20 INPUT N$

```

Line 20 interrupts the program to wait for your **INPUT**, then stores your **INPUT** in string variable N\$. Your Commodore 64 has infinite patience and will not continue the program until you answer the question. *Type* in your name to the right of the prompt (?) and *press* **RETURN**.

**note:** No mere **RUN/STOP** key will suffice to **BREAK** a program awaiting operator **INPUT**. To **BREAK** any **INPUT**, *press* **RUN/STOP-RESTORE**.

```

40 PRINT " (CLR/HOME) "; "HELLO, sp"; N$
50 PRINT: PRINT "HOW OLD ARE YOU, sp"; N$; "? "

```

Line 40 clears the display, then **PRINTs** HELLO, followed by the current value of string variable N\$ (your name).

Line 50 **PRINTs** a blank line to tidy the display, then **PRINTs** the question HOW OLD ARE YOU, followed by the current value of string variable N\$ (your name).

```

60 INPUT A

```

Line 60 interrupts the program to wait for your **INPUT**, then stores your **INPUT** in variable A.

**note:** Any alphabetic **INPUT** in response to a request for numeric **INPUT** (**INPUT** A above) will result in the following prompt:

```
?REDO FROM START
```

The prompt is requesting you to re-**INPUT** a number. Just *type* in your age and *press* **RETURN**.

```

80 PRINT " (CLR/HOME) "; A; "?!?! "
90 PRINT: PRINT "YOU DON'T LOOK A DAY OVER": A-1

```

Line 80 first clears the display, then **PRINTs** the current value of variable A (your age—unless of course you lied).

Line 90 **PRINTs** a blank line followed by a wisecrack remark.

Now make the following changes:

Add line 5:

```

5 PRINT " (CLR/HOME) "

```

Amend line 10:

```

10 INPUT "WHAT'S YOUR NAME"; N$

```

*Type* 20 and *press* **RETURN** to **DELETE** line 20. Now add line 30:

```

30 PRINT " (CLR/HOME) "

```

Amend lines 40 and 50:

```

40 PRINT "HELLO, sp";N$; ", sp";
50 INPUT "HOW OLD ARE YOU";A

```

DELETE line 60 and add line 70:

```

70 PRINT " (CLR/HOME) "

```

Amend line 80:

```

80 PRINT A; "?!!!"

```

DON'T GET PERSONAL should now LIST as follows:

```

0 REM DON'T GET PERSONAL
5 PRINT " (CLR/HOME) "
10 INPUT "WHAT'S YOUR NAME";N$
30 PRINT " (CLR/HOME) "
40 PRINT "HELLO, sp";N$; ", sp";
50 INPUT "HOW OLD ARE YOU";A
70 PRINT " (CLR/HOME) "
80 PRINT A; "?!!!"
90 PRINT:PRINT "YOU DON'T LOOK A DAY OVER";A-1

```

Now RUN DON'T GET PERSONAL again, and follow along.

### In Plain English . . .

```

0 REM DON'T GET PERSONAL
5 PRINT " (CLR/HOME) "
10 INPUT "WHAT'S YOUR NAME";N$

```

Line 0 sets the mood, and line 5 clears the display.

Line 10 now incorporates the **INPUT** prompt (WHAT'S YOUR NAME) into the **INPUT** statement, storing your **INPUT** in string variable N\$.

**note:** The **INPUT** prompt and the relevant variable (string variable N\$ in line 10) are always separated by a semicolon. The **INPUT** prompt must not exceed 38 characters.

Type in your name and press **RETURN**.

```

30 PRINT " (CLR/HOME) "
40 PRINT "HELLO, sp";N$; ", sp";
50 INPUT "HOW OLD ARE YOU";A

```

Line 30 clears the display, and line 40 **PRINT**s HELLO, followed by the current value of string variable N\$ (your name). The semicolon at the end of line 40 guarantees that the next **PRINT** statement (line 50) will follow immediately on the same line.

As in line 10, line 50 now incorporates the **INPUT** prompt (HOW OLD ARE YOU) into the **INPUT** statement, storing your **INPUT** in variable A. *Type* in your age and *press* **RETURN**.

```

70 PRINT " (CLR/HOME) "
80 PRINT A; "?!!! "
90 PRINT:PRINT "YOU DON'T LOOK A DAY OVER"; A-1

```

Line 70 clears the display, line 80 **PRINT**s the current value of variable A (your age), and line 90 remarks appropriately.

## MORE AND MORE

---

### Multiple Input

You can also **INPUT** more than one piece of data on any given line. As an example, *type* the following program. (First *type* **NEW** and *press* **RETURN** to clear your Commodore's memory.)

```

NEW
0 REM FRIENDS
5 PRINT " (CLR/HOME) "
10 INPUT "PLEASE LIST FOUR FRIENDS"; A$, B$, C$, D$
20 PRINT:INPUT "NOW LIST THEIR RESPECTIVE AGES"; A, B, C, D
30 PRINT " (CLR/HOME) "
40 PRINT "THE AVERAGE AGE OF"
50 PRINT A$; ", sp"; B$; ", sp"; C$; ", spANDsp"; D$; "spISSp";
   (A + B + C + D) / 4

```

Now **RUN** your new program called FRIENDS.

### In Plain English . . .

```

0 REM FRIENDS
5 PRINT " (CLR/HOME) "

```

```

10 INPUT "PLEASE LIST FOUR FRIENDS"; A$, B$, C$, D$
20 PRINT:PRINT "NOW LIST THEIR RESPECTIVE AGES"; A, B, C, D

```

Line 0 names the program, and line 5 clears the display.

Line 10 requests four separate bits of **INPUT**, storing each subsequent **INPUT** in string variables A\$, B\$, C\$, and D\$. So, list four friends as prompted, *pressing RETURN* after each name.

**note:** **INPUT** string variables must always be separated by commas.

Line 20 first **PRINTs** a blank line, then prompts you to list their respective ages, storing your numeric **INPUT** in variables A, B, C, and D. Enter their respective ages, *pressing RETURN* after each entry.

```

30 PRINT " (CLR/HOME) "
40 PRINT "THE AVERAGE AGE OF"
50 PRINT A$; ", sp"; B$; ", sp"; C$; ", spANDsp"; D$; "spISsp";
  (A + B + C + D) / 4

```

Line 30 clears the display, and line 40 **PRINTs** THE AVERAGE AGE OF.

Line 50 **PRINTs** the current values of string variables A\$, B\$, C\$, and D\$ (your friends' names), then averages their ages (A + B + C + D)/4.

## **2 + 2 = 4, OR DOES IT?**

---

### **Testing Input**

The following program demonstrates how to test your **INPUT**.

```

NEW
10 PRINT " (CLR/HOME) "
20 INPUT "YES OR NO (Y OR N)"; A$
30 IF A$ = "Y" THEN 70
40 IF A$ <> "N" THEN 10
50 PRINT " (CLR/HOME) "; "OKAY, BE THAT WAY!!"
60 STOP
70 PRINT " (CLR/HOME) "
80 PRINT "YES WHAT? "

```

and **RUN**. Maddening, eh?

**In Plain English . . .**

```

10 PRINT " (CLR/HOME) "
20 INPUT "YES OR NO (Y OR N) "; A$

```

Line 10 clears the display, and line 20 requests your **INPUT** (Y OR N), storing the **INPUT** in string variable A\$.

```

30 IF A$="Y" THEN 70
40 IF A$<>"N" THEN 10

```

Line 30 tests string variable A\$ and will cause the program to jump to line 70 only **IF** A\$="Y".

Line 40 tests string variable A\$ only **IF** line 30 is false (i.e., if A\$ does *not* equal "Y"). If A\$ does *not* equal "N" (A\$<>"N"), then the program jumps back to line 10 to request additional **INPUT**.

```

50 PRINT " (CLR/HOME) "; "OKAY, BE THAT WAY!! "
60 STOP

```

Line 50 clears the display and **PRINTs** OKAY, BE THAT WAY only **IF** line 40 proves false (i.e., if A\$ equals "N").

Line 60 **STOPs** the program with a BREAK IN 60 prompt. **STOP** does just that.

```

70 PRINT " (CLR/HOME) "
80 PRINT "YES WHAT? "

```

Line 70 clears the display only if line 30 tests as true (A\$="Y").

Line 80 **PRINTs** YES WHAT?

**GETTING AHEAD****The GET Statement**

There's a faster, more direct method of talking back to your Commodore. *Type* the following:

```

NEW
10 PRINT " (CLR/HOME) "
20 GET A
30 PRINT A

```



and **RUN**.

All that to get a 0. What happened?

### In Plain English . . .

Line 10 clears the display.

**GET** in line 20 reads the keyboard and stores the value of whichever key is pressed in variable A. Unlike **INPUT**, **GET** does not await your **INPUT** and does not require you to press **RETURN**. Since no key was pressed, variable A stored the number 0 and line 30 **PRINT**ed the current value of variable A (0).

So what good is **GET**? Amend line 20 to read

```
20 GET A: IF A=0 THEN 20
```

and **RUN**. This time, the screen clears and nothing at all—not even a zero—appears. *Type* in any number from 0 to 9. Your number appears instantly with no need to press **RETURN**.

### In Plain English . . .

Now line 20 includes a conditional statement that tests **GET A: IF A=0 THEN 20**. Simply, **IF** no key is pressed (0), **THEN** loop back to line 20. More simply, **IF** no key is pressed, **THEN** stick around and wait until one is.

**note:** Any attempt to enter an alphabetic letter or a graphic character in response to a **GET** statement with a numeric variable will result in the following prompt:

```
?SYNTAX ERROR
```

**GET** statements work equally well with string variables. Consider the following:

```
10 PRINT " (CLR/HOME) "
20 GET A$: IF A$ = " " THEN 20
30 PRINT A$
```

and **RUN**.

### In Plain English . . .

Line 10 clears the display. **GET A\$** in line 20 scans the keyboard and stores the value of any pressed key in string variable A\$. **IF A\$ = " "** **THEN 20** is a conditional statement to test string variable A\$. If string variable A\$ is empty (" "), **THEN** line 20 loops itself until any key is pressed.

- `A$ = " "` indicates an *empty string*.

Line 30 **PRINTs** the current value of string variable `A$`.

Now place a semicolon at the end of line 30:

```
30 PRINT A$;
```

and add line 40:

```
40 GOTO 20
```

and **RUN**. You have just programmed a primitive word processor. Amazing! (Go ahead, you can now write the Great American Novel.)

**note:** Unlike **INPUT**, **GET** will **BREAK** simply by pressing the **RUN/STOP** key at any time.

**GET** it? Got it! Good.

## CHAPTER 5 REVIEW

---

Empty string	<b>INPUT</b>
<b>GET</b>	<b>STOP</b>

---

## PRACTICE EXERCISES

---

- 5-1. Amend the program you wrote for Exercise 4-1 to allow you to **INPUT** both your initial deposit and the interest rate.
  - 5-2. Write a program to calculate the area of circle X in square inches. (*Hint:* **INPUT** allows you to alter radius length, and the equation  $\text{Radius} \times \text{Radius} \times \text{PI}$ , where **PI** (or  $\pi$ ) equals 3.141 592 653, will calculate the area of circle X in square inches.)
-

BACK AGAIN

---

## FOR/NEXT Loops

Let's set up a simple table to square each number from 1 to 10:

```
10 PRINT " (CLR/HOME) "  
20 A = 1  
30 PRINT A; "*" ; A; " = "; A*A  
40 A = A + 1  
50 IF A <= 10 THEN 30
```

and **RUN**. Your display should appear as follows:

```
1 * 1 = 1  
2 * 2 = 4  
3 * 3 = 9  
4 * 4 = 16  
5 * 5 = 25  
6 * 6 = 36  
7 * 7 = 49  
8 * 8 = 64  
9 * 9 = 81  
10 * 10 = 100
```

**In Plain English . . .**

```
10 PRINT " (CLR/HOME) "  
20 A = 1  
30 PRINT A; "*" ; A; " = "; A*A
```

Line 10 clears the display, and line 20 sets variable A equal to 1.

Line 30 **PRINTs** the current value of A \* A, or A “squared.”

```

40  A = A + 1
50  IF A <= 10 THEN 30

```

Line 40 adds 1 to A with each loop.

Line 50 is a conditional statement that tests the current value of A. **IF** A is less than or equal to 10, **THEN** the program loops back to program line 30.

Now make the following changes.

First, amend lines 20 and 40:

```

20  FOR A = 1 TO 10
40  NEXT A

```

and **DELETE** line 50. Your program should now **LIST** as follows:

```

10  PRINT " (CLR/HOME) "
20  FOR A = 1 TO 10
30  PRINT A; "*"; A; " = "; A*A
40  NEXT A

```

It will **RUN** exactly the same as before.

### In Plain English . . .

```

20  FOR A = 1 TO 10

```

**FOR** marks the beginning of a **FOR/NEXT** loop (my favorite kind). The *control loop variable* (A in line 20) always follows the word **FOR**. The numbers on either side of the word **TO** are the loop parameters that determine the value of the control loop variable. They are the lowest and highest values that the variable can attain.

```

40  NEXT A

```

**NEXT A** recalls the **FOR/NEXT** loop and increases the value of A by 1.

In essence, the **FOR/NEXT** loop acted as an internal counter, counting the number of times you wanted the loop to run—a very useful tool.

**note:** You can't use integer variables (A%) in **FOR/NEXT** loops.

Here's another application of the **FOR/NEXT** loop:

NEW

```

10 PRINT "(CLR/HOME) "; "JUST HANGING AROUND FOR A MOMENT"
20 FOR X=1 TO 1000
30 NEXT
40 PRINT "(CLR/HOME) "; "BE BACK IN A FLASH"
50 FOR X=1 TO 750:NEXT
60 PRINT "(CLR/HOME) "; "BACK AGAIN"

```

and RUN. Swift, isn't it?

### In Plain English . . .

```

10 PRINT "(CLR/HOME) "; "JUST HANGING AROUND FOR A MOMENT"
20 FOR X=1 TO 1000
30 NEXT

```

Line 10 clears the display, then PRINTs JUST HANGING AROUND FOR A MOMENT.

Line 20 sets the parameters of FOR/NEXT loop control variable X from 1 to 1000.

Line 30 recalls FOR/NEXT loop X a total of 1000 times (1–1000). Note that we didn't specify NEXT X, but simply NEXT. A NEXT statement without a variable returns to the most recently used FOR statement still in effect.

**note:** Lines 20 and 30 form an *empty loop*, a programming device used to make your Commodore pause. You may increase or decrease the length of each pause by increasing or decreasing the FOR/NEXT loop parameter (1 TO 3000 will pause approximately 3 times longer than 1 TO 1000).

```

40 PRINT "(CLR/HOME) "; "BE BACK IN A FLASH"
50 FOR X=1 TO 750:NEXT
60 PRINT "(CLR/HOME) "; "BACK AGAIN"

```

Line 40 clears the display, then PRINTs BE BACK IN A FLASH.

Line 50 combines two separate statements into one complete empty FOR/NEXT loop.

Line 60 clears the display, then PRINTs BACK AGAIN.

**note:** It is possible to have two *separate* FOR/NEXT loops with the same control variable in the same program. (Both FOR/NEXT loops in the above program use control variable X.)



**note:** A nested loop control variable must have a name that is different from the name of the outside loop (H and V in the above program).

```

40 PRINT "*";
50 NEXT H
60 NEXT V

```

Line 40 PRINTs an asterisk. (Don't forget the semicolon.)

NEXT H in line 50 recalls nested FOR/NEXT loop H a total of 40 times (1 TO 40), thereby PRINTing a line of 40 asterisks. Nested FOR/NEXT loop H is *completely* nested within FOR/NEXT loop V—exactly as it should be. Nested loops must be completely nested.

Line 60 recalls FOR/NEXT loop V a total of 10 times. Each loop PRINTs another line of 40 asterisks (nested FOR/NEXT loop H).

**note:** Lines 50 and 60 could be combined into a single line—line 50:

```

50 NEXT H: NEXT V

```

## STEP LIVELY

### Incrementing

The following program lists all multiples of the number 4 from 0 to 100:

```

NEW
10 PRINT " (CLR/HOME) "
20 FOR X=0 TO 100 STEP 4
30 PRINT X,
40 NEXT

```

and RUN. Your display should appear as follows:

0	4	8	12
16	20	24	28
32	36	40	44
48	52	56	60
64	68	72	76
80	84	88	92
96	100		

**In Plain English . . .**

```

10 PRINT " (CLR/HOME) "
20 FOR X=0 TO 100 STEP 4

```

Line 20 sets the parameters of **FOR/NEXT** loop X from 0 **TO** 100 at increments or “steps” of 4. **STEP** increases any **FOR/NEXT** loop by the number indicated.

```

30 PRINT X,
40 NEXT

```

Line 30 **PRINTs** the current value of X (note the comma); line 40 recalls **FOR/NEXT** loop X and increases control variable X by 4 (**STEP** 4).

Now add the following lines to the above program:

```

50 FOR X=96 TO 0 STEP-4
60 PRINT X,
70 NEXT

```

and **RUN**. Once the program reaches the number 100 at line 40, it reverses itself back to 0 via lines 50–70.

**In Plain English . . .**

Line 50 sets another (separate) **FOR/NEXT** loop X from 96 **TO** 0 at negative increments of 4 (**STEP** - 4).

Line 60 **PRINTs** the current value of X, as does line 30.

Line 70 recalls **FOR/NEXT** loop X and decreases the value of X by 4 with each loop until the parameters are satisfied (**STEP** - 4).

**CHAPTER 6 REVIEW**


---

Empty loop	Loop parameters
<b>FOR/NEXT</b> loop	Nested loops
Loop control variable	<b>STEP</b>

---



## **PRACTICE EXERCISES**

---

- 6-1. Write a program to list multiples of 25 from 0 to 1000.
  - 6-2. Amend your program from Exercise 4-1 using a **FOR/NEXT** loop to reveal your yearly balance from 1980 to the year 2000.
-

OLD MAN SEVEN COME DOWN FROM HEAVEN

## Random Number Generator

Another talent of your gifted Commodore is that it can produce a list (of any length) of random numbers. *Type* and **RUN** this program:

```
10 PRINT " (CLR/HOME) "
20 FOR X=1 TO 10
30 PRINT RND (1) ,
40 NEXT
```

Your display should appear something like this (your actual numbers will be different, though):

```
. 716284506      . 0345277492
. 965938555      . 890989285
. 556669511      . 589110064
. 685832073      . 889211415
. 262441911      . 450346693
```

Each subsequent **RUN** will produce a different set of numbers.

### In Plain English . . .

```
10 PRINT " (CLR/HOME) "
20 FOR X=1 TO 10
```

Line 20 sets the parameters of **FOR/NEXT** loop X from 1 **TO** 10.

```
30 PRINT RND (1)
40 NEXT
```

Line 30 **PRINTs** a random number (**RND**). **RND** will produce a random number between 0 and 1.

**note:** For our purposes, you must include a whole number greater than 0 in parentheses after **RND**. The number in parentheses does not affect the random generation, but it does control the repeatability of the number sequence generated.

Line 40 recalls **FOR/NEXT** loop X and adds 1 to the value of X for a total of 10 loops.

Now let's try to simulate the roll of a single die. First make the following change to line 30:

```
30 PRINT INT (6*RND (1) ) ,
```

and **RUN**. Oops! You get integers, all right, but only from 0 to 5. Close, but no cigar.

- **INT** will truncate any number down to its integer form. The number following **INT** must be in parentheses. **PRINT INT 5.6** will result in a ?SYNTAX ERROR prompt, whereas **PRINT INT(5.6)** will return the integer 5, which is what you want.

Time to amend line 30 again:

```
30 PRINT INT (6*RND (1) ) + 1 ,
```

and **RUN**. By George, I think you've got it! Now all your random numbers should be from 1 to 6.

The following program provides you with the formula necessary to determine the high and low parameters for *any* range of random numbers.

**NEW**

```
10 REM RANDOM NUMBER GENERATOR
20 PRINT " (CLR/HOME) "
30 PRINT TAB (7) ; "RANDOM NUMBER GENERATOR"
40 PRINT:PRINT TAB (7) ; "PRESS ANY KEY TO BEGIN"
50 GET A$: IF A$ = "" THEN 50
60 PRINT " (CLR/HOME) "
70 INPUT "ENTER LOW NUMBER"; L
80 PRINT:INPUT "ENTER HIGH NUMBER"; H
90 RH = (H - L + 1)
100 PRINT " (CLR/HOME) "
110 FOR X = 1 TO 52
120 RN = INT (RH*RND (1) ) + L
130 PRINT RN,
140 NEXT
```

```

150 PRINT:PRINT "PRESS ANY KEY TO GO AGAIN"
160 GET A$: IF A$="" THEN 160
170 GOTO 60

```

Now **RUN** RANDOM NUMBER GENERATOR and press any key to begin, as prompted. The second prompt will request you to enter the low number. *Type* in the number 5 and *press* **RETURN**. The third prompt will request you to enter the high number. *Type* in the number 10 and *press* **RETURN**. Your display should appear something like this (your number sequence will differ, of course):

```

7      10      8      10
8      10      8      9
6      9       5      10
5      5       7      9
8      6       7      10
5      6       8      6
7      10      10     6
6      7       5      6
7      5       10     8
6      9       8      7
7      10      10     7
6      7       5      5
8      8       7      5

```

Each of the random numbers generated falls in the range of 5 to 10, exactly as you predetermined. Now press any key to go again as instructed. This time, try setting your low number at 1 and your high number at 20. All of the resultant random numbers now range from 1 to 20.

### In Plain English . . .

```

10 REM RANDOM NUMBER GENERATOR
20 PRINT " (CLR/HOME) "
30 PRINT TAB(7); "RANDOM NUMBER GENERATOR"
40 PRINT:PRINT TAB(7); "PRESS ANY KEY TO BEGIN"
50 GET A$: IF A$="" THEN 50

```

Line 30 **PRINT**s RANDOM NUMBER GENERATOR at column position **TAB**(7).

Line 40 **PRINT**s a blank line and then **PRINT**s PRESS ANY KEY TO BEGIN at column position **TAB**(7).

Line 50 scans the keyboard (**GET A\$**) and loops itself (waits) until you press any key, as requested.

```

60 PRINT " (CLR/HOME) "
70 INPUT "ENTER LOW NUMBER"; L
80 PRINT: INPUT "ENTER HIGH NUMBER"; H
90 RH = (H - L + 1)

```

Line 70 requests your **INPUT** (ENTER LOW NUMBER) and stores it in variable L.

Line 80 requests your **INPUT** (ENTER HIGH NUMBER) and stores it in variable H.

Line 90 calculates the random high (RH) number required later by the random number generator in line 120. If your high number (H) is 10 and your low number (L) is 5, then your random high number (RH) would be calculated as follows:  $RH = (10 - 5 + 1)$ . Thus your random high number would be 6.

```

100 PRINT " (CLR/HOME) "
110 FOR X=1 TO 52
120 RN = INT (RH*RND (1) ) + L
130 PRINT RN,
140 NEXT

```

Line 110 sets the parameters of **FOR/NEXT** loop X from 1 to 52.

Line 120 is the formula that generates a random number (RN) with a value of from 5 to 10. In other words,  $RN = \text{INT}(6 * \text{RND}(1)) + 5$  will result in a random number with a value of from 5 to 10.

Line 130 **PRINTs** the random value of RN.

Line 140 recalls **FOR/NEXT** loop X and increases the value of X by 1 for a total of 52 loops.

```

150 PRINT:PRINT "PRESS ANY KEY TO GO AGAIN"
160 GET A$: IF A$ = "" THEN 160
170 GOTO 60

```

Line 150 **PRINTs** a blank line, then **PRINTs** PRESS ANY KEY TO GO AGAIN.

Line 160 scans the keyboard (waits) until you press any key as requested.

Line 170 loops back to line 60 for another set of high and low parameters.

The RANDOM NUMBER GENERATOR program will work on any range of numbers, whether negative or positive.

## KNOCKING IT DOWN

---

### Rounding Off

You may have already noticed that your Commodore 64 sometimes produces an absurd number of decimal places. **INT** alone will truncate downward but will not round off numbers to the nearest integer. For example, try the following in Immediate Mode:

```
PRINT INT (1.1)
```

and then try

```
PRINT INT (1.9)
```

Both will truncate downward and return the number 1. Not so good. What to do? Try adding .5 to each number.

```
PRINT INT (1.1 + .5)
```

which returns 1 because  $1.1 + .5 = 1.6$  and  $\text{INT}(1.6) = 1$ . Now try this:

```
PRINT INT (1.9 + .5)
```

which returns 2 because  $1.9 + .5 = 2.4$  and  $\text{INT}(2.4) = 2$ . So far, all you've gotten are integers. But how would you round off to one decimal place? Try this: Multiply your number by 10, add .5, then divide by 10:

```
PRINT INT (5.77*10 + .5) / 10
```

which returns 5.8. You did it! It follows then that multiplying and dividing any number by 100 will round off to two decimal places:

```
PRINT INT (5.777*100 + .5) / 100
```

which returns 5.78. And so on. . . .

## CHAPTER 7 REVIEW

---

**INT**

Random number generator

**RND**

Rounding Off

---

## PRACTICE EXERCISES

---

- 7-1. Use **RND** to generate a random number from 1 to 50. Use **INPUT** and a **FOR/NEXT** loop to give yourself 8 chances to guess the random number. Have the computer tell you whether each guess is too high, too low, or correct, and then tell you the correct number after 8 incorrect guesses.
  - 7-2. Write a program to list the years 1980–2000, telling whether each year is or isn't a leap year. [*Hint*: Any year that is a multiple of 4 is a leap year (remember **INT**).]
  - 7-3. Amend Exercise 4-1 to round off your yearly savings total to the nearest cent.
-

---

**READ, DATA, and RESTORE:**

You already know how to assign variables directly and via **INPUT** or **GET**. But what happens when you have loads of data? Surely there must be a faster method than laboriously assigning variables to long lists of numbers (and I wouldn't mention the possibility if it didn't exist). Well, it does. Begin by typing the following program:

```
1Ø PRINT " (CLR/HOME) "  
2Ø X=1  
3Ø READ N  
4Ø PRINT X; " = "; N  
5Ø X=X+1  
6Ø GOTO 3Ø  
7Ø DATA 1, 2, 3, 4, 5
```

and **RUN**. Your display should appear as follows:

```
1 = 1  
2 = 2  
3 = 3  
4 = 4  
5 = 5  
?OUT OF DATA ERROR IN 3Ø  
READY.
```

**In Plain English . . .**

```
1Ø PRINT " (CLR/HOME) "  
2Ø X=1  
3Ø READ N  
4Ø PRINT X; " = "; N
```

Line 2Ø sets variable X equal to 1.



Line 30 (**READ N**) instructs the computer to **READ** the first element of the **DATA** statement from line 70 (**DATA 1,2,3,4,5**) and to store that **DATA** element in variable N. A **DATA pointer** in the computer tracks each element of **DATA** sequentially (1,2,3,4,5), and then each subsequent **READ** statement **READs** the remaining **DATA**—one element at a time—until there are no more elements to be **READ**, causing the ?OUT OF DATA ERROR prompt to appear. **DATA** elements can be listed in several **DATA** statements, and the computer will move from one statement to the next until all have been **READ**. You'll find multiple **DATA** statements helpful when you want to **READ** a large number of **DATA** elements.

Line 40 **PRINTs** the current value of X equal to the current value of N as assigned by line 30.

```
50  X = X + 1
60  GOTO 30
70  DATA 1, 2, 3, 4, 5
```

Line 50 sets variable X equal to X + 1.

Line 60 loops back to line 30 to **READ** another element of **DATA**.

Line 70 contains the required **DATA**.

Note these important points about **READ/DATA** commands:

1. Each individual element of **DATA** is separated from any subsequent **DATA** by a comma—except at the end of the **DATA** line.
2. It is possible to use real numbers in any format or integers as **DATA**, but you cannot **READ** variables or math expressions.
3. Make sure that there are enough **DATA** elements to satisfy the required number of **READ** statements in your program.
4. Certain **DATA** elements, including punctuation and spaces, must be contained within quotes.

Now amend the above program as follows to see how **READ/DATA** can be used with string variables:

```
30  READ N$
40  PRINT X; " = "; N$
70  DATA ONE, ON TOP OF, THE OTHER
```

and **RUN**. Now your display should appear as follows:

```
1 = ONE
2 = ON TOP OF
3 = THE OTHER
?OUT OF DATA ERROR IN 30
READY.
```

### In Plain English . . .

```
30 READ N$
40 PRINT X; " = "; N$
70 DATA ONE, ON TOP OF, THE OTHER
```

Line 30 now instructs the computer to **READ** the first string **DATA** element (ONE) of line 70 and store it in string variable N\$.

Line 40 **PRINTs** the current value of X followed by the current value of string variable N\$ as determined by line 30.

Line 70 contains all the string **DATA**.

String **DATA** elements generally do not require quotation marks. Individual string elements, however, must be separated by commas, as with numeric **DATA**. If you wish to incorporate commas in any string **DATA** element (for example, 1 Fifth Avenue, New York), enclose the entire element in quotation marks (**DATA** "1 Fifth Avenue, New York"). Otherwise, the computer will **READ** the comma as a break between string **DATA** elements and will automatically truncate the string at the comma.

**note:** Be certain that your **READ** variables are compatible with your **DATA** elements. In other words, don't request your Commodore 64 to store string **DATA** in numeric variables.

**DATA** may be **READ** from within **FOR/NEXT** loops as well:

```
NEW
10 PRINT " (CLR/HOME) "
20 PRINT TAB(10); "MULTIPLICATION TABLE": PRINT
30 READ N
40 FOR X=1 TO 12
50 PRINT TAB(10); N; "spXsp"; X; TAB(21); "sp = sp"; N*X
60 NEXT
70 DATA 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
80 FOR X=1 TO 3000:NEXT
90 GOTO 10
```

and **RUN**. The following display appears:

MULTIPLICATION TABLE

```

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

The display pauses for a few seconds, then clears before reappearing with a new table—and so on from 1 to 12. An ?OUT OF DATA ERROR IN 30 prompt appears after the last table clears (number 12).

**In Plain English . . .**

```

10 PRINT " (CLR/HOME) "
20 PRINT TAB (10) ; "MULTIPLICATION TABLE": PRINT
30 READ N
    
```

Line 20 **PRINTs** MULTIPLICATION TABLE at **TAB** column 10, then **PRINTs** a blank line.

Line 30 **PRINTs** the first element of **DATA** (1) from line 70 and stores it in variable N.

```

40 FOR X=1 TO 12
50 PRINT TAB (10) ; N; "spXsp"; X; TAB (21) ; "sp = sp"; N*X
60 NEXT
    
```

Line 40 sets the parameters of **FOR/NEXT** loop X from 1 **TO** 12.

Line 50 **PRINTs** the current value of N times the current value of X as equal to the result of the mathematical expression N \* X.

Line 60 recalls **FOR/NEXT** loop X and increases the value of X by 1 for a total of 12 loops.

```

70 DATA 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
80 FOR X=1 TO 3000: NEXT
90 GOTO 10

```

Line 70 contains the **DATA**.

Line 80 is an empty loop which interrupts the program for several seconds, causing the computer to pause.

Line 90 loops back to program line 10. The value of N increases by 1 with each loop until the ?OUT OF DATA ERROR IN 30 prompt stops the program.

**question:** How do we get the above program to repeat itself 12 times (for all 12 tables) without encountering the ?OUT OF DATA ERROR IN LINE 30 prompt?

**answer:** First amend line 70 as follows:

```

70 DATA 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13

```

Now add the following lines:

```

35 IF N=13 THEN 100
100 RESTORE
110 PRINT:PRINT TAB(7); "PRESS ANY KEY TO GO AGAIN"
120 GET A$: IF A$="" THEN 120
130 GOTO 10

```

and **RUN**. This time, the multiplication tables appear as before, 1–12, except that the ?OUT OF DATA ERROR IN LINE 30 prompt is replaced by PRESS ANY KEY TO GO AGAIN. So, go ahead and press any key to go again.

### In Plain English . . .

```

35 IF N=13 THEN 100
70 DATA 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13

```

The number 13 in lines 35 and 70 is called a *flag*. It tells the computer when to exit or move on to something else.

Line 35 tests the value of N and will (**GOTO**) 100 only **IF** N = 13.

```

100 RESTORE
110 PRINT:PRINT TAB(7); "PRESS ANY KEY TO GO AGAIN"
120 GET A$: IF A$="" THEN 120
130 GOTO 10

```

Line 100 is where your program will wind up only IF N = 13 (if the flag in DATA line 70 is encountered). RESTORE resets the DATA pointer to the first DATA element. So, IF N = 13, THEN (GOTO) 100 to RESTORE (reset) the DATA pointer to the first DATA element (1).

Line 110 PRINTs a blank line, then PRINTs PRESS ANY KEY TO GO AGAIN at TAB column 7.

Line 120 scans the keyboard (GET A\$) and waits until you press any key.

Line 130 loops back to line 10 to begin again.

**CHAPTER 8 REVIEW**

---

DATA	READ
Flag	RESTORE
Pointer	

---

**PRACTICE EXERCISES**

---

- 8-1. Use a FOR/NEXT loop and a READ/DATA statement to PRINT the sentence THIS WAS EASIER THAN I THOUGHT, letting each word represent a single string DATA element.
- 8-2. Write a program that uses DATA statements to store the alphabet (A through Z), and a corresponding numerical value for each letter, for example,

```
DATA A, 1, B, 2, C, 3, D, 4, . . . .
```

Have the program (a) request you to INPUT any letter, search the DATA statements for the requested letter, then PRINT the letter and its numeric equivalent; (b) inform you when your INPUT was not a letter (a number or other character), and consequently not found in the DATA statements; (c) loop back to beginning to request and search for another letter (remember RESTORE).

---

---

## Subroutines

The following program is a rather frustrating example of subroutines:

```
10 PRINT " (CLR/HOME) "  
20 FOR X=1 TO 10:PRINT:NEXT  
30 PRINT TAB(15);"WHERE AM I?"  
40 GOSUB 200  
50 PRINT "UPSTAIRS?"  
60 GOSUB 200  
70 PRINT "NOT UP HERE"  
80 GOSUB 200  
90 FOR X=1 TO 20:PRINT:NEXT  
100 PRINT "DOWNSTAIRS?"  
110 GOSUB 200  
120 FOR X=1 TO 20:PRINT:NEXT  
130 PRINT "NOT DOWN HERE EITHER"  
140 GOSUB 200  
150 FOR X=1 TO 10:PRINT:NEXT  
160 PRINT TAB(16);"FORGET IT"  
170 GOSUB 200  
180 END  
200 FOR X=1 TO 2000  
210 NEXT  
220 PRINT " (CLR/HOME) "  
230 RETURN
```

and RUN. Never found me, right?

### In Plain English . . .

```
10 PRINT " (CLR/HOME) "  
20 FOR X=1 TO 10:PRINT:NEXT
```

```

30 PRINT TAB(15); "WHERE AM I?"
40 GOSUB 200

```

Line 20 sets the parameters of **FOR/NEXT** loop X from 1 **TO** 10, **PRINTs** a blank line, then recalls itself, **PRINTing** a total of 10 blank lines.

Line 30 **PRINTs** WHERE AM I? at **TAB** column 15.

Line 40 (**GOSUB 200**) recalls the *subroutine* beginning at line 200. This **GOSUB** command initiates a three-step process:

1. Go to (**GOSUB**) the indicated program line (line 200).
2. Perform the subroutine in lines 200–230 (see explanation of lines 200–230 below).
3. **RETURN** (line 230) again to the main program, resuming at the line directly following the **GOSUB** command (line 50).

The subroutine commencing at line 200 is recalled and executed a total of 6 times (lines 40, 60, 80, 110, 140, and 170) and requires a total of 10 lines (40, 60, 80, 110, 140, 170, 200, 210, 220, and 230). The exact same task repeated six times without the use of a subroutine would require 18 program lines (200, 210, and 220 times 6). *Conclusion:* subroutines save programming time and effort—not to mention computer memory.

Use subroutines whenever you want to branch away from the main program several times to perform the same operation. There is no need to type identical program lines two or more times. Type them once, then recall them as often as required with **GOSUB**.

**RETURN** marks the end of the subroutine and **RETURNs** the computer to the main program line directly following the original **GOSUB** command.

```

50 PRINT "UPSTAIRS"
    :
    :
140 GOSUB 200

```

Lines 50 through 140 send us through the same subroutine four times, with various statements being **PRINTed** in between.

```

150 FOR X=1 TO 10:PRINT:NEXT
160 PRINT TAB(16); "FORGET IT"
170 GOSUB 200
180 END

```

Line 150 **PRINTs** 10 blank lines.

Line 160 **PRINTs** FORGET IT (may be sound advice) at **TAB** column 16.

Line 170 recalls the subroutine at line 200 for the last time.

Line 180 (**END**) prevents the main program from colliding with the subroutine at line 200.

Delete line 180 and **RUN** the program. Everything proceeds exactly as before, except at the very end of the program: the display becomes blank for a moment and the prompt ?RETURN WITHOUT GOSUB ERROR IN 230 appears. The main program crashed into the subroutine at line 200 and eventually encountered the **RETURN** statement in line 230, even though the subroutine was never recalled via a **GOSUB** command. Hence the ?RETURN WITHOUT GOSUB ERROR prompt.

```

200 FOR X=1 TO 2000
210 NEXT
220 PRINT " (CLR/HOME) "
230 RETURN

```

Lines 200 and 210, the beginning of the subroutine, form an empty loop to create a pause for a few moments.

Line 220 clears the display.

Line 230 **RETURNS** the computer to the main program line that directly follows the recalling **GOSUB** statement.

**note:** Some programs will require several different subroutines to perform several different tasks. You can reduce the resulting confusion in your own mind by labeling each subroutine with a **REM** statement. For instance, the subroutine in the above program might have been labeled **REM WAIT AND CLEAR**.

Here's a quick program to demonstrate how main programs can be instructed to *branch* off to specific subroutines:

```

NEW
10 PRINT " (CLR/HOME) "
20 INPUT "ENTER A NUMBER FROM 1 TO 5";N
30 IF N<1 OR N>5 THEN 10
40 ON N GOSUB 100,200,300,400,500
50 FOR X=1 TO 2000:NEXT
60 GOTO 10
100 REM SUBROUTINE #1
110 PRINT " (CLR/HOME) "
120 PRINT "I'M SUBROUTINE NUMBER 1, YOU CALLED?"
130 RETURN
200 REM SUBROUTINE #2
210 PRINT " (CLR/HOME) "
220 PRINT "I'M SUBROUTINE NUMBER 2, YOU CALLED?"

```



```

230 RETURN
300 REM SUBROUTINE #3
310 PRINT " (CLR/HOME) "
320 PRINT "I'M SUBROUTINE NUMBER 3, YOU CALLED?"
330 RETURN
400 REM SUBROUTINE #4
410 PRINT "I'M SUBROUTINE NUMBER 4, YOU CALLED?"
430 RETURN
500 REM SUBROUTINE #5
510 PRINT " (CLR/HOME) "
520 PRINT "I'M SUBROUTINE NUMBER 5, YOU CALLED?"
530 RETURN

```

and **RUN**. The program will direct you to whichever subroutine corresponds to your **INPUT** number.

### In Plain English . . .

```

10 PRINT " (CLR/HOME) "
20 INPUT "ENTER A NUMBER FROM 1 TO 5";N
30 IF N<1 OR N>5 THEN 10

```

Line 20 requests you to **INPUT** any number from 1 to 5 and then stores your **INPUT** in variable N.

Line 30 tests **INPUT** variable N. **IF** N is less than 1 **OR** greater than 5, **THEN** the program goes to line 10 to request correct **INPUT**.

```

40 ON N GOSUB 100,200,300,400,500

```

Line 40 does the actual branching (recalls the appropriate subroutine). **ON N** performs a function similar to the **IF-THEN** "test." For example, if N = 1, then **ON** would direct the program to the first line (100) indicated in the list (100,200,300,400,500) following the word **GOSUB**. So it is the same as saying **IF** N = 1, **THEN GOSUB** 100, **IF** N = 2 **THEN GOSUB** 200, etc.

**note:** **ON** can be used equally well with **GOTO** statements. If N were equal to 0 or a number greater than those in the list of line numbers, then the computer would **GOTO** the program line directly following the **ON** statement.

```

50 FOR X=1 TO 2000:NEXT
60 GOTO 10

```

Line 50 is an empty loop to pause a few moments.

Line 60 loops back to line 10 for another round.

```

100 REM SUBROUTINE #1
110 PRINT " (CLR/HOME) "
120 PRINT "I'M SUBROUTINE NUMBER 1, YOU CALLED?"
130 RETURN

```

Line 100 marks the beginning of SUBROUTINE #1.

Line 110 clears the display.

Line 120 PRINTs I'M SUBROUTINE #1, YOU CALLED?

Line 130 RETURNs the computer to the main program (at line 50).

Lines 200 through 530 are minor variations on SUBROUTINE #1.

## CHAPTER 9 REVIEW

---

Branching	<b>RETURN</b>
GOSUB	Subroutine
ON	

---

## PRACTICE EXERCISES

---

- 9-1. Write a single subroutine to flash HEADS or TAILS, depending on which phrase is chosen by a randomly generated number in the main program.
  - 9-2. In Exercise 7-1 you wrote a program to list the multiples of 25 from 0 to 1000 using a FOR/NEXT loop. Now write a program that requests you to INPUT any number from 5 to 25, then branches to a subroutine that PRINTs the multiples of your INPUT number to 1000 before RETURNing to request another INPUT number.
-

## BEAM ME UP, MR. SPOCK

## Logic

Did you ever think you might be losing your marbles?

The following program starts you out with two bags of marbles and asks you to indicate whether each is open (enter Y for yes or N for no):

```
10 PRINT "(CLR/HOME) "  
20 INPUT "IS BAG #1 OPEN (Y OR N) "; A$  
30 IF A$<>"Y" AND A$<>"N" THEN 10  
40 PRINT "(CLR/HOME) "  
50 INPUT "IS BAG #2 OPEN (Y OR N) "; B$  
60 IF B$<>"Y" AND B$<>"N" THEN 40  
70 IF A$="Y" AND B$="Y" THEN 130  
80 IF A$="Y" AND B$="N" OR A$="N" AND B$="Y" THEN 150  
90 PRINT "(CLR/HOME) "  
100 PRINT "BOTH BAGS ARE SHUT TIGHT":PRINT "YOU HAVE ALL  
    YOUR MARBLES"  
110 FOR X=1 TO 3000:NEXT:GOTO 10  
120 PRINT "(CLR/HOME) "  
130 PRINT "BOTH BAGS ARE OPEN-":PRINT "YOU'VE LOST ALL  
    YOUR MARBLES"  
140 GOTO 110  
150 PRINT "(CLR/HOME) "  
160 PRINT "ONLY ONE BAG IS OPEN-":PRINT "YOU ARE LOSING YOUR  
    MARBLES"  
170 GOTO 110
```

and **RUN**. Try all four of the possible **INPUT** combinations:

	Is bag open	
	A	B
1.	Y	Y
2.	N	N
3.	Y	N
4.	N	Y

### In Plain English . . .

```

10 PRINT " (CLR/HOME) "
20 INPUT "IS BAG #1 OPEN (Y OR N) "; A$
30 IF A$ <> "Y" AND A$ <> "N" THEN 10

```

Line 20 requests your **INPUT** and stores it in string variable A\$.

Line 30 tests your **INPUT**. **IF** A\$ does not equal the letter Y **AND** does not equal the letter N, **THEN (GOTO) 10** to request additional **INPUT**. Only **IF** both conditions test true will the computer loop back to program line 10.

- **AND** is a *logical operator*. It requires all conditions to test true.

```

40 PRINT " (CLR/HOME) "
50 INPUT "IS BAG #2 OPEN (Y OR N) "; B$
60 IF B$ <> "Y" AND B$ <> "N" THEN 40

```

Line 50 requests your **INPUT** and stores it in string variable B\$.

Line 60 tests your **INPUT** again. This time, **IF** string variable B\$ does not equal the letter Y **AND** B\$ does not equal the letter N, **THEN (GOTO) 40** to request additional **INPUT**. Again, both conditions (B\$ <> "Y" **AND** B\$ <> "N") must test true before the computer will loop back to program line 40.

```

70 IF A$ = "Y" AND B$ = "Y" THEN 130
80 IF A$ = "Y" AND B$ = "N" OR A$ = "N" AND B$ = "Y" THEN 150

```

Line 70 tests both string variables A\$ **AND** B\$. Only **IF** both A\$ = "Y" **AND** B\$ = "Y" (possibility #1—Y/Y), **THEN** branch to program line 130.

Line 80 tests two different possibilities (#3, Y/N and #4, N/Y). **IF** either A\$ = "Y" **AND** B\$ = "N" **OR** A\$ = "N" **AND** B\$ = "Y" **THEN (GOTO)** program line 150.

- **OR** is a logical operator that requires only one of the two or more possibilities to test true. "Either/or," your Commodore says. **AND** requires all conditions to test true, whereas **OR** requires only one.

```

90 PRINT " (CLR/HOME) "
100 PRINT "BOTH BAGS ARE SHUT TIGHT":PRINT "YOU HAVE ALL
    YOUR MARBLES"
110 FOR X=1 TO 3000:NEXT:GOTO 10

```

Line 100 **PRINT**s BOTH BAGS ARE SHUT TIGHT, then drops a line to **PRINT YOU HAVE ALL YOUR MARBLES**. This line will **PRINT** only **IF** both lines 70 and 80 test false (possibility #2, N/N).

Line 110 includes an empty loop to pause the program, then loops back to line 10 to begin again.

```

120 PRINT " (CLR/HOME) "
130 PRINT "BOTH BAGS ARE OPEN-":PRINT "YOU'VE LOST ALL
    YOUR MARBLES"
140 GOTO 110

```

Line 130 **PRINT**s BOTH BAGS ARE OPEN-, then drops a line to **PRINT YOU'VE LOST ALL YOUR MARBLES**, only **IF** line 70 tests true (possibility #1, Y/Y).

Line 140 loops back to line 110.

The remainder of the program (lines 150–170) takes you through the sequence just described, but with variations on the **PRINT** statement.

- **NOT** is another logical operator. The computer will interpret **IF A <> 5** and **IF NOT A = 5** exactly the same. Therefore, the use of <> or **NOT** is simply a matter of preference.

Your Commodore will always perform all logic tests in this order of priority: **NOT**, then **AND**, then **OR**. As with math operators, parentheses can be used to alter the priority.

---

## Relational Tests

Try the following:

```

NEW
10 PRINT " (CLR/HOME) "
20 A = (5>6)
30 B = (5<6)
40 PRINT A, B

```

and **RUN**. The numbers 0 and -1 will appear.

**In Plain English . . .**

Line 20 tests the conditions within parentheses, then assigns an appropriate value to variable A: -1 **IF** true and 0 **IF** false. Thus,  $A = 0$ , because 5 is not greater than 6.

Line 30 also tests the conditions within parentheses and assigns the appropriate value to variable B. Since your Commodore assigns a value of -1 to any true test,  $B = -1$ , because 5 is less than 6.

Now try the following program:

```
NEW
10 PRINT " (CLR/HOME) "
20 INPUT "ENTER YOUR YEARLY INCOME"; INC
30 I = 2 + (INC >= 20000)
40 ON I GOTO 100, 200
100 PRINT "RICH MAN"
110 STOP
200 PRINT "POOR MAN"
```

and **RUN**.

**In Plain English . . .**

```
10 PRINT " (CLR/HOME) "
20 INPUT "ENTER YOUR YEARLY INCOME"; INC
30 I = 2 + (INC >= 20000)
40 ON I GOTO 100, 200
```

Line 20 requests your **INPUT** and stores it in variable INC.

Line 30 tests the expression  $INC \geq 20000$ : **IF** variable INC is greater than or equal to 20000 (i.e., if the test is true), **THEN**  $I = 2 + (-1)$ . Therefore, **IF**  $INC \geq 20000$  **THEN**  $I = 1$ . **IF** variable INC is **NOT** greater than or equal to 20000 (i.e., if the test is false), **THEN**  $I = 2 + (0)$ . Therefore, **IF NOT**  $INC \geq 20000$  **THEN**  $I = 2$ .

Line 40 will branch to the appropriate program line according to the value of variable I (**ON I**).

```
100 PRINT "RICH MAN"
110 STOP
200 PRINT "POOR MAN"
```

Line 100 **PRINTs** RICH MAN only **IF**  $INC \geq 20000$  ( $I = 1$ ).

Line 110 **STOPs** the program to prevent it from crashing into line 200.

Line 200 **PRINTs** POOR MAN only **IF**  $INC < 20000$  ( $I = 2$ ).

**CHAPTER 10 REVIEW**

---

**AND**

Logical operator

**NOT****OR**Relational test

---

**PRACTICE EXERCISES**

---

- 10-1. Write a program that requests you to **INPUT** two whole numbers, **PRINTs** them, and tells whether each is odd or even.
- 10-2. Write a program that generates a random number between 1 and 5, then requests you to **INPUT** any two numbers. Have your Commodore **PRINT** "BRAVO! YOU WIN!" only if the second **INPUT** number equals either the first **INPUT** number *plus* the random number or the first **INPUT** number *minus* the random number. Otherwise, have the program loop back to request two new **INPUT** numbers.
-

## String Handling Functions

You've already been introduced to literal strings and string variables. But what more can you do with them? Try this program:

```

10 PRINT " (CLR/HOME) "
20 FOR X=1 TO 10:PRINT:NEXT: INPUT "ENTER ANY WORD";W$
30 PRINT " (CLR/HOME) "
40 FOR X=1 TO 10:PRINT:NEXT
50 PRINT W$;"sp=sp";LEN(W$);"spCHARACTERS"
60 FOR X=1 TO 2000:NEXT:GOTO 10

```

**RUN** and enter any word as prompted. The screen will clear and your Commodore will tell you exactly how many characters are in your word.

### In Plain English . . .

```

10 PRINT " (CLR/HOME) "
20 FOR X=1 TO 10:PRINT:NEXT: INPUT "ENTER ANY WORD";W$
30 PRINT " (CLR/HOME) "
40 FOR X=1 TO 10:PRINT:NEXT

```

**FOR/NEXT** loop X in line 20 **PRINTs** 10 blank lines, requests your **INPUT**, and then stores it in string variable W\$.

Line 30 clears the display, and **FOR/NEXT** loop X in line 40 **PRINTs** 10 blank lines.

```

50 PRINT W$;"sp=sp";LEN(W$);"spCHARACTERS"
60 FOR X=1 TO 2000:NEXT:GOTO 10

```

Line 50 **PRINTs** the current value of string variable W\$ as equal to the number of characters in W\$, that is, the **LENgth** of W\$ or **LEN(W\$)**, followed by the word **CHARACTERS**. **LEN**(string variable) will return the exact number of characters (including blank spaces, punctuation (except commas), and graphics) in any string.

Empty **FOR/NEXT** loop X in line 60 causes the program to pause for a few seconds, then loops it back to line 10.



You can use **LEN** to test the **LEN**gth of any **INPUT** string:

```

10 PRINT " (CLR/HOME) "
20 PRINT:PRINT: INPUT "ENTER ANY FOUR LETTER WORD";W$
30 IF LEN(W$)<>4 THEN 10
40 PRINT " (CLR/HOME) "
50 PRINT:PRINT:PRINT W$;"spIS A FOUR-LETTER WORD"
60 PRINT:PRINT:PRINT "PRESS ANY KEY TO CONTINUE"
70 GET A$: IF A$="" THEN 70
80 GOTO 10

```

and **RUN**. Now your Commodore simply will not accept any word of more or less than four characters (only four-letter words!).

### In Plain English . . .

Line 20 **PRINT**s two blank lines, then requests you to **INPUT ANY FOUR-LETTER WORD**, storing your **INPUT** in string variable **W\$**.

Line 30 tests the **LEN**gth of string variable **W\$**. Only **IF LEN(W\$)** is not equal to 4, **THEN (GOTO) 10**.

Line 50 **PRINT**s two blank lines, then the current value of **W\$**, followed by **IS A FOUR LETTER WORD**.

Line 60 **PRINT**s a couple more blank lines and then prompts you to **PRESS ANY KEY TO CONTINUE**.

Line 70 loops itself until you press any key.

Line 80 loops back to program line 10 to begin again.

The following is a handy little program that allows you to center any string in the middle of your screen:

```

NEW
10 PRINT " (CLR/HOME) "
20 PRINT:PRINT: INPUT "ENTER ANY STRING (MAX. 40
   CHARACTERS) : ";A$
30 C=20-LEN(A$)/2
40 PRINT " (CLR/HOME) "
50 FOR X=1 TO 10:PRINT:NEXT
60 PRINT TAB(C);A$
70 FOR X=1 TO 10:PRINT:NEXT:PRINT "PRESS ANY KEY TO GO
   AGAIN"

```

```

80 GET A$: IF A$="" THEN 80
90 GOTO 10

```

and **RUN**.

### In Plain English . . .

Line 20 **PRINTs** two blank lines, requests you to **INPUT** any string, storing your **INPUT** in string variable A\$. (For obvious reasons, your string must not contain commas.)

Line 30 uses **LEN** to calculate a value for C which is then used in line 60 to set the **TAB** position.

Line 50 **PRINTs** 10 blank lines.

Line 60 **PRINTs** string variable A\$ at the **TAB** position determined by the current value of C (calculated in line 30).

Line 70 **PRINTs** 10 blank lines, then **PRINTs** PRESS ANY KEY TO GO AGAIN.

Line 80 loops itself until you press any key.

Line 90 loops back to line 10.

## CUTTING THE STRINGS

---

### Left\$, Mid\$, and Right\$

The following program introduces you to the process of *slicing* strings:

```

NEW
10 PRINT " (CLR/HOME) "
20 A$="CHEAPMODERATEEXPENSIVE"
30 PRINT:PRINT:PRINT TAB(14);"CTRL-9 THE NEW SUIT"
40 PRINT:PRINT:TAB(14);"1sp=sp";LEFT$(A$,5)
50 PRINT TAB(14);"2sp=sp";MID$(A$,6,8)
60 PRINT TAB(14);"3sp=sp";RIGHT$(A$,9)
70 PRINT:PRINT "CHOOSE 1, 2, OR 3"
72 INPUT SUIT
75 IF SUIT<>1 AND SUIT<>2 AND SUIT<>3 THEN 10
80 ON SUIT GOSUB 100,200,300
90 FOR X=1 TO 3000:NEXT:GOTO 10
100 PRINT " (CLR/HOME) "

```

```

110 PRINT:PRINT:PRINT LEFT$(A$,5);"?sp--spSORRY, NO
    ALTERATIONS":RETURN
200 PRINT "(CLR/HOME) "
210 PRINT:PRINT:PRINT MID$(A$,6,8);"?sp--LOOKS ALMOST
    TAILOR MADE":RETURN
300 PRINT "(CLR/HOME) "
310 PRINT:PRINT:PRINT RIGHT$(A$,9);"?sp--spTHIS SUIT
    IS YOU!!":RETURN

```

and **RUN**. Your display should appear as follows:

### THE NEW SUIT

```

1 = CHEAP
2 = MODERATE
3 = EXPENSIVE

```

```

CHOOSE 1, 2, OR 3
?

```

THE NEW SUIT will appear in *inverse* letters. Now choose 1, 2, or 3 as prompted, and your Commodore will make the appropriate remark.

### In Plain English . . .

```

10 PRINT "(CLR/HOME) "
20 A$="CHEAPMODERATEEXPENSIVE"
30 PRINT:PRINT:PRINT TAB(14);"CTRL-9 THE NEW SUIT"

```

Line 20 sets string variable A\$ to equal CHEAPMODERATEEXPENSIVE.

Line 30 **PRINTs** two blank lines, then **PRINTs** THE NEW SUIT in inverse letters at **TAB** position 14. **CTRL-9** enters the inverse mode. *Press RETURN* to exit again.

```

40 PRINT:PRINT TAB(14);"1sp=sp";LEFT$(A$,5)
50 PRINT TAB(14);"2sp=sp";MID$(A$,6,8)
60 PRINT TAB(14);"3sp=sp";RIGHT$(A$,9)

```

Line 40 **PRINTs** one blank line, then moves to **TAB** position 14, where the 5 left-hand characters of string variable A\$, or **LEFT**(A\$,5), are **PRINTed**, which is the word CHEAP.

Line 50 moves to **TAB** position 14 of the next line and **PRINTs** the **MID**\$ characters of string variable A\$ (the word MODERATE). **MID**\$(A\$,6,8) slices the **MID**dle char-

acters of string variable A\$. The 6 indicates at which character to start the slice, and the 8 indicates how many consecutive characters to include in the slice. The word MODERATE starts at character 6 of string variable A\$ and has a total of 8 characters.

Line 60 moves to **TAB** position 14 of the next line and **PRINTs** the 9 right-hand characters, or **RIGHT\$(A\$,9)**, of string variable A\$, which is the word EXPENSIVE.

```

70 PRINT:PRINT "CHOOSE 1, 2, OR 3"
72 INPUT SUIT
75 IF SUIT<>1 AND SUIT<>2 AND SUIT<>3 THEN 10
80 ON SUIT GOSUB 100,200,300
90 FOR X=1 TO 3000:NEXT:GOTO 10

```

Line 70 **PRINTs** a blank line, then requests you to CHOOSE 1, 2, OR 3. Line 72 stores your **INPUT** in variable SUIT.

Line 75 tests your **INPUT** and will allow the program to loop back to line 10 only **IF** SUIT does not equal 1, 2, or 3.

Line 80 branches to the appropriate subroutine (**ON SUIT GOSUB**) as determined by the current value of SUIT.

Empty **FOR/NEXT** loop X in line 90 causes the program to pause for a few seconds before looping back to start again (**GOTO 10**).

```

100 PRINT " (CLR/HOME) "
110 PRINT:PRINT:PRINT LEFT$(A$,5); "? sp--spSORRY, NO
    ALTERATIONS":RETURN
200 PRINT " (CLR/HOME) "
210 PRINT:PRINT:PRINT MID$(A$,6,8); "? sp--spLOOKS ALMOST
    TAILOR MADE":RETURN
300 PRINT " (CLR/HOME) "
310 PRINT:PRINT:PRINT RIGHT$(A$,9); "? sp--spTHIS SUIT
    IS YOU!!":RETURN

```

Lines 100, 200, and 300 clear the display.

Line 110 **PRINTs** a blank line, then **PRINTs** CHEAP? [from **LEFT\$(A\$,5)**], then the remark SORRY, NO ALTERATIONS before **RETURNing** to the main program at line 90.

Line 210 **PRINTs** two blank lines, then MODERATE? [from **MID\$(A\$,6,8)**], then the remark LOOKS ALMOST TAILOR MADE before **RETURNing** to the main program at line 90.

Line 310 **PRINTs** two blank lines, then EXPENSIVE? [from **RIGHT\$(A\$,9)**], then the remark THIS SUIT IS YOU!! before **RETURNing** to the main program at line 90.

**STRINGS TO NUMBERS AND BACK AGAIN****VAL**

The number 123 and the string "123" both **PRINT** exactly the same, but there is an important difference: You can add, subtract, multiply, and divide the number 123, but you can't perform any math functions on the string "123". What to do? Try the following:

```

NEW
10 PRINT " (CLR/HOME) "
20 FOR X=1 TO 10:PRINT:NEXT
30 PRINT TAB (10) : INPUT "ENTER ANY NUMBER";N$
40 PRINT " (CLR/HOME) "
50 FOR X=1 TO 10:PRINT:NEXT
60 PRINT TAB (10) ;N$;
70 N=VAL (N$) -1
80 PRINT "sp-";N; "="; VAL (N$) -N
90 PRINT:PRINT TAB (7) ; "PRESS ANY KEY TO GO AGAIN"
100 GET A$: IF A$="" THEN 100
110 GOTO 10

```

and **RUN**. Try several different numbers, positive and negative. Your Commodore will forever return the answer 1.

**In Plain English . . .**

```

10 PRINT " (CLR/HOME) "
20 FOR X=1 TO 10:PRINT:NEXT
30 PRINT TAB (10) : INPUT "ENTER ANY NUMBER";N$

```

Line 20 **PRINTs** 10 blank lines.

Line 30 requests you to enter any number, storing your **INPUT** in string variable N\$.

```

40 PRINT " (CLR/HOME) "
50 FOR X=1 TO 10:PRINT:NEXT
60 PRINT TAB (10) ;N$;

```

Line 50 **PRINTs** 10 more blank lines.

Line 60 **PRINTs** the current value of string variable N\$ at **TAB** position 10.

```
70 N=VAL(N$)-1
```

Line 70 converts string variable N\$ to its numeric **VAL**ue, subtracts 1, and stores the new total in variable N. **VAL**(string variable) converts any suitable string into its numeric equivalent (**VAL**ue). Suitable strings include both numbers and math expressions.

To see what **VAL** does, *type* the following in Immediate Mode: **PRINT "5\*1"** and *press* **RETURN**. The string 5\*1 will appear. Now *type* **PRINT VAL("5\*1")** and *press* **RETURN**. Your Commodore converts the string "5\*1" to its numerical expression value and returns the number 5.

```
80 PRINT "sp-";N;"="; VAL(N$)-N
90 PRINT:PRINT TAB(7);"PRESS ANY KEY TO GO AGAIN"
100 GET AS:IF AS="" THEN 100
110 GOTO 10
```

Line 80 **PRINTs** the current value of numeric variable N, then the result of **VAL**(N\$) minus N.

Line 90 **PRINTs** PRESS ANY KEY TO GO AGAIN at **TAB** position 7.

Line 100 loops itself until you press any key.

Line 110 loops back to line 10.

**note:** The function **STR\$** behaves in a manner exactly opposite to that of **VAL**. Whereas **VAL** converts suitable strings into numbers (e.g., "123" into 123), **STR\$** converts numbers into strings (e.g., 123 into "123").

## ARE WE RELATED

### String Relations

By testing the relationship of one string against another, your computer can act as a sorter. The most obvious application of this is to have it alphabetize lists for you. The following program requests you to **INPUT** two names, then lists the names in alphabetical order:

```
NEW
10 PRINT "(CLR/HOME) "
20 FOR N=1 TO 2
```

```

30 INPUT "ENTER ANY NAME"; N$(N)
40 NEXT
50 PRINT " (CLR/HOME) "
60 IF N$(1) < N$(2) THEN 100
70 PRINT N$(2) : PRINT N$(1)
80 END
100 PRINT N$(1) : PRINT N$(2)

```

and **RUN**. Sure enough, it worked.

### In Plain English . . .

Line 20 sets the parameters of **FOR/NEXT** loop N from 1 **TO** 2.

Line 30 requests you to enter any name, storing your **INPUT** in string variable N\$ as the current value of N: 1 or 2.

Line 40 recalls **FOR/NEXT** loop N and adds 1 to the value of N.

Line 60 tests the relationship of N\$(1) against N\$(2). Only **IF** N\$(1) precedes N\$(2) alphabetically—N\$(1) < N\$(2)—**THEN (GOTO) 100**.

Line 70 **PRINTs** N\$(2), then **PRINTs** N\$(1) only **IF** line 60 tests false, i.e., **IF** N\$(1) is not less than N\$(2). Simply stated, line 70 will execute only **IF** N\$(1) does not precede N\$(2) alphabetically.

Line 80 stops the program to prevent it from crashing into line 100 and will also execute only **IF** line 60 tests false.

Line 100 **PRINTs** N\$(1), then **PRINTs** N\$(2) only **IF** line 60 tests true, i.e., **IF** N\$(1) precedes N\$(2) alphabetically.

As you witnessed in this program, relations can be used to sort strings alphabetically. Your Commodore assigns each letter of the alphabet to a consecutive code in the ASCII (American Standard Code for Information Interchange) table. For example,

A	is less than	B
AM	is less than	AMPLE
ANT	is less than	ANTS

Uppercase letters precede lowercase letters, so,

Z	is less than	a
---	--------------	---

See Chapter 13 for more about ASCII.

**STRING VARIABLE TI\$****Time to Spare**

The string variable **TI\$** should be used only in special circumstances. For instance,

```

NEW
10 PRINT " (CLR/HOME) "
20 FOR X=1 TO 10:PRINT:NEXT
30 PRINT "SET YOUR CLOCK, PLEASE"
40 PRINT:PRINT " (00HRS/00MINS/00SECS) ":PRINT
50 INPUT TI$
60 PRINT " (CLR/HOME) "
70 PRINT " (HOME) ":PRINT TI$:GOTO 70

```

and **RUN**. SET YOUR CLOCK as prompted, using six digits with no spaces, then *press RETURN*. Voila! Clock.

**In Plain English . . .**

```

10 PRINT " (CLR/HOME) "
20 FOR X=1 TO 10:PRINT:NEXT
30 PRINT "SET YOUR CLOCK, PLEASE"
40 PRINT:PRINT " (00HRS/00MINS/00SECS) ":PRINT

```

**FOR/NEXT** loop X in line 20 **PRINTs** 10 blank lines.

Line 30 **PRINTs** SET YOUR CLOCK, PLEASE.

Line 40 **PRINTs** (00HRS/00MINS/00SECS), then **PRINTs** a blank line.

```

50 INPUT TI$
60 PRINT " (CLR/HOME) "
70 PRINT " (HOME) ":PRINT TI$:GOTO 70

```

Line 50 requests you to **INPUT** the **Time**—hours (00HRS), minutes (00MINS), and seconds (00SECS).

The variable **TI\$** stores the current value of the *real-time* clock inside your Commodore, starting with 0 the moment you power up. You can reset the variable **TI\$** by either of two methods: (1) turn the power off, then on again (crude), or (2) change the value of **TI\$**. You are, in a real sense, altering the value of **TI\$** in program line 50 by **INPUTing** your own value.



Line 70 positions the cursor at the **HOME** position, **PRINTs** the current value of **TI\$**, then loops itself.

## CHAPTER 11 REVIEW

---

ASCII	SHIFT-CTRL-9
Inverse	Slicing
<b>LEFT\$</b>	<b>STR\$</b>
<b>LENgth</b>	String relation
<b>MID\$</b>	<b>TI\$</b>
<b>RIGHT\$</b>	<b>VAL</b>

---

## PRACTICE EXERCISES

---

- 11-1. Words that begin or end with the letters ASS are ASSes. Write a program that requests you to **INPUT** a word, then lists whether or not your **INPUT** word is an ASS according to the above definition.
- 11-2. Write a program to alphabetize the last names of up to 10 of your friends.
-

## Arrays

You've already learned how to assign data to variables, either directly through **INPUT** statements or via **READ/DATA** statements. There is yet another method: *arrays*—sets of numbers that share something in common. Consider the following:

Theater row A = seats 5, 6, 7, and 8

These four seats share the fact that they all reside in row A. Working from left to right, we might have, for example:

Seat 1 in row A = seat #5  
 Seat 2 in row A = seat #6  
 Seat 3 in row A = seat #7  
 Seat 4 in row A = seat #8

Thus arrayed, each seat 5 through 8 in row A can be assigned a sequential *subscript number* 1 to 4, which you type inside parentheses following the variable. Your Commodore assigns a different *subscript* number to each element—always starting with the number 0. Thus A(0) refers to seat #5 in row A, A(1) to seat #6, A(2) to seat #7, and A(3) to seat #8.

**note:** An array name consists of a variable followed by the number of *elements* (as many as you want subject only to available memory) in parentheses.

Let's expand row A to 15 seats and try an array on the computer. The following program assigns a different seat price to each of the 15 seats in row A.

```
10 PRINT " (CLR/HOME) "
30 P=48
40 FOR R=1 TO 15
50 A(R)=P-(R*3)
60 PRINT "SEAT"; R; TAB(8); "spCOSTS $"; A(R)
70 NEXT
```

and **RUN**. The above program assigns prices to seats 1 through 10, then stops with the prompt ?BAD SUBSCRIPT ERROR IN 50. (You'll see why in a moment.) Remedy? Add the following program line:

```
20 DIM A(15)
```

and **RUN**. Now the program assigns prices to all 15 seats.

### In Plain English . . .

```
10 PRINT "(CLR/HOME) "
20 DIM A(15)
```

**DIM A(15)** in line 20 reserves memory space for 16 individual subscript elements (theater seats) of *single-dimension* array (or row) A, and sets each subscript element's value to 0. We have chosen to ignore A(0), a legitimate element, and work with those elements that correspond to our situation—A(1) through A(15).

**note:** As demonstrated earlier, you must use a **DIM** statement when working with arrays of more than 10 subscript elements. Also, you can conserve on memory if you use integer variables, when practical.

```
30 P=48
40 FOR R=1 TO 15
50 A(R)=P-(R*3)
60 PRINT "SEAT";R;TAB(8);"spCOSTS $";A(R)
70 NEXT
```

Line 30 sets the initial seat price (variable P) to 48.

Line 40 sets the parameters of **FOR/NEXT** loop R (mnemonic for row) from 1 **TO** 15.

Line 50 calculates the value of each seat—A(R)—via the formula  $P - (R * 3)$ . Thus, in the first loop of **FOR/NEXT** loop R, A(R) = A(1) and A(1) = 48 - (1 \* 3), so seat 1 in row A—A(1)—costs \$45. In the second loop, A(R) = A(2) and A(2) = 48 - (2 \* 3), so seat 2 in row A—A(2)—costs \$42.

Line 60 **PRINTs** SEAT, the current value of R, COSTS, and the current value of A(R).

Line 70 recalls **FOR/NEXT** loop R and automatically increases the value of R by 1.

To check the stored values of single-dimension array A use the Immediate Mode:

```
Type PRINT A(1) and press RETURN
Type PRINT A(2) and press RETURN
```

Type **PRINT A(3)** and press **RETURN**

Type **PRINT A(4)** and press **RETURN**

The values of single-dimension array **A** are stored exactly as they appeared before:  
**A(1) = 45, A(2) = 42, A(3) = 39, A(4) = 36, etc.**

Now add the following lines to the above program.

```
80 FOR X=1 TO 3000:NEXT
90 PRINT "(CLR/HOME) "
100 INPUT "WHICH SEAT WOULD YOU LIKE (1-15) "; R
110 IF R<1 OR R>15 THEN 90
120 PRINT "(CLR/HOME) "
130 PRINT "SEAT"; R; "spCOSTS $"; A(R)
140 GOTO 80
```

and **RUN**.

### In Plain English . . .

Empty **FOR/NEXT** loop **X** in line 80 merely causes the program to pause for a few seconds.

Line 100 requests you to **INPUT** your desired seat number and then stores your **INPUT** in variable **R**.

Line 110 tests your **INPUT**. The computer will loop back to line 90 only **IF** your **INPUT R** is less than 1 **OR** greater than 15.

Line 130 **PRINTs** the price **A(R)** of whichever seat (**R**) you selected.

Line 140 loops back to line 80.

---

## String Arrays

Single-dimension string arrays may sound complex but they're really not. For example, try this:

```
NEW
10 PRINT "(CLR/HOME) "
20 DIM A$(15)
30 FOR R=1 TO 15
```

```

40 PRINT "SEAT";R;TAB(9);"IS ASSIGNED TO: ";
50 INPUT A$(R)
60 NEXT
70 PRINT "(CLR/HOME) "
80 FOR R=1 TO 15
90 PRINT A$(R);"spIS IN SEAT";R
100 NEXT
110 FOR X=1 TO 3000:NEXT
120 PRINT "(CLR/HOME) "
130 INPUT "ENTER SEAT NUMBER (1-15)";R
140 IF R<1 OR R>15 THEN 120
150 PRINT "(CLR/HOME) "
160 PRINT A$(R);"spIS SITTING IN SEAT";R
170 GOTO 110

```

and **RUN**. The program first requests you to **INPUT** 15 names, lists the total seat assignment, requests you to **INPUT** a seat number, then tells you who has been assigned to that seat.

### In Plain English . . .

```

10 PRINT "(CLR/HOME) "
20 DIM A$(15)
30 FOR R=1 TO 15
40 PRINT "SEAT";R;TAB(9);"IS ASSIGNED TO: ";
50 INPUT A$(R)
60 NEXT

```

**DIM A\$(15)** in line 20 reserves memory space for 16 subscript elements of single-dimension string array A.

Line 30 sets the parameters of **FOR/NEXT** loop R from 1 **TO** 15.

Line 40 **PRINTs** SEAT, the current value of R (at **TAB** position 9), and the phrase IS ASSIGNED TO:

Line 50 requests you to **INPUT** a name and stores your **INPUT** in subscript element A\$(R).

Line 60 recalls **FOR/NEXT** loop R and increases the value of R by 1.

```

80 FOR R=1 TO 15
90 PRINT A$(R);"spIS IN SEAT";R
100 NEXT

```

Line 80 sets the parameters of **FOR/NEXT** loop R from 1 **TO** 15.

Line 90 **PRINT**s the value of A\$(R), IS IN SEAT, and the current value of R.  
 Line 100 recalls **FOR/NEXT** loop R and increases the value of R by 1.

```

110 FOR X=1 TO 3000:NEXT
120 PRINT " (CLR/HOME) "
130 INPUT "ENTER SEAT NUMBER (1-15) ";R
140 IF R<1 OR R>15 THEN 120
150 PRINT " (CLR/HOME) "
160 PRINT A$(R) : "spIS SITTING IN SEAT";R
170 GOTO 110

```

Line 110 is an empty loop (to pause for a moment).

Line 130 requests you to enter a seat number and stores your **INPUT** in R.

Line 140 tests the value of **INPUT** R. Your Commodore will loop back to program line 120 only **IF** R is less than 1 **OR** greater than 15.

Line 160 **PRINT**s the current value of A\$(R), IS SITTING IN SEAT, and the current value of R.

Line 170 loops back to line 110.

---

## Multidimensional Arrays

But what if the numbers in your arrays have more than one thing in common? Consider the following as an example:

	Seat			
	1	2	3	4
ROW 1	1	2	3	4
ROW 2	1	2	3	4
ROW 3	1	2	3	4
ROW 4	1	2	3	4

Let's say we've expanded our seating capacity for an encore performance. Accordingly, we must expand from a single-dimensional array to a *multidimensional* array. Now we need to know not only the row but also the specific seat (or column) within that row. Thus we must expand our subscript from the single-DIMENSION A(1) to the multi-DIMENSION A(1,1), which gives us the row number, then the specific seat number within that row (following the comma). Reserving memory space in the computer is just as simple (the number of dimensions is limited only by available memory): **DIM** A(4,4) will reserve memory space for a multidimensional array of five rows (0 through 4), each

with five seats (0 through 4). We'll ignore those seats with a designated row or seat number of 0.

The following program will take your ticket reservation request based on the above seating diagram, then tell you whether or not your desired seat is available:

```

NEW
10 DIM A(4,4)
20 PRINT "(CLR/HOME) "
30 INPUT "ENTER ROW (1-4) ";R
40 IF R<1 OR R>4 THEN 10
50 PRINT "(CLR/HOME) "
60 INPUT "ENTER SEAT NUMBER (1-4) ";S
70 IF S<1 OR S>4 THEN 50
80 PRINT "(CLR/HOME) "
90 PRINT "YOU REQUESTED SEAT";S;"spIN ROW";R
100 FOR X=1 TO 3000:NEXT
110 IF A(R,S)=1 THEN 220
120 FOR X=1 TO 10:PRINT:NEXT
130 PRINT "SEAT";S;"spIN ROW";R;"spIS AVAILABLE"
140 PRINT:PRINT "I HAVE RESERVED IT FOR YOU"
150 A(R,S)=1
160 FOR X=1 TO 3000:NEXT
170 PRINT "(CLR/HOME) "
180 FOR X=1 TO 10:PRINT:NEXT
190 PRINT "PRESS ANY KEY FOR ANOTHER SEAT"
200 GET A$:IF A$="" THEN 200
210 GOTO 20
220 PRINT "(CLR/HOME) "
230 FOR X=1 TO 10:PRINT:NEXT
240 PRINT "SORRY, SEAT";S;"spIN ROW";R;"spIS RESERVED"
250 GOTO 160

```

and **RUN**. Get your requests in early.

### In Plain English . . .

```

10 DIM A(4,4)
20 PRINT "(CLR/HOME) "
30 INPUT "ENTER ROW (1-4) ";R
40 IF R<1 OR R>4 THEN 10
50 PRINT "(CLR/HOME) "

```

```

60 INPUT "ENTER SEAT NUMBER (1-4) "; S
70 IF S<1 OR S>4 THEN 50

```

**DIM** A(4,4) in line 10 reserves memory space for multidimensional array A—four rows, each with four seats.

Line 30 requests you to enter the row and then stores your **INPUT** in variable R.

Line 40 tests **INPUT** R and will loop back to line 20 only **IF** R is less than 1 **OR** R is greater than 4. (In other words, it only allows you to **INPUT** a number from 1 to 4, in case you're trying to be tricky.)

Line 60 requests you to enter the seat number and then stores your **INPUT** in variable S.

Line 70 tests **INPUT** S and will loop back to line 50 only **IF** S is less than 1 **OR** greater than 4.

```

80 PRINT " (CLR/HOME) "
90 PRINT "YOU REQUESTED SEAT"; S; "spIN ROW"; R
100 FOR X=1 TO 3000:NEXT
110 IF A(R,S)=1 THEN 220
120 FOR X=1 TO 10:PRINT:NEXT
130 PRINT "SEAT"; S; "spIN ROW"; R; "spIS AVAILABLE"
140 PRINT:PRINT "I HAVE RESERVED IT FOR YOU"
150 A(R,S)=1

```

Line 90 **PRINTs** YOU REQUESTED SEAT (the current value of S) IN ROW (the current value of R).

Line 100 is an empty loop to slow down the program.

Line 110 tests the value of the seat you selected—A(R,S)—and will branch to line 220 only **IF** A(R,S)=1.

Line 120 **PRINTs** 10 blank lines.

Line 130 **PRINTs** SEAT (the current value of S) IN ROW (the current value of R) IS AVAILABLE.

Line 140 **PRINTs** one blank line, then **PRINTs** I HAVE RESERVED IT FOR YOU.

Line 150 assigns the value of 1 to A(R,S) so that the next eager ticket buyer can't reserve your seat.

```

160 FOR X=1 TO 3000:NEXT
170 PRINT " (CLR/HOME) "
180 FOR X=1 TO 10:PRINT:NEXT
190 PRINT "PRESS ANY KEY FOR ANOTHER SEAT"
200 GET AS$: IF AS$="" THEN 200
210 GOTO 20
220 PRINT " (CLR/HOME) "

```



```

230 FOR X=1 TO 10:PRINT:NEXT
240 PRINT "SORRY, SEAT";S;"spIN ROW";R;"spIS RESERVED"
250 GOTO 160

```

Line 160 is an empty loop.

Lines 180 and 230 PRINT 10 blank lines.

Line 190 PRINTs PRESS ANY KEY FOR ANOTHER SEAT, which is a prompt for you.

Line 200 loops itself until you press any key as requested.

Line 210 loops to program line 20 to request another seat.

Lines 220 to 250 execute only IF program line 110 tests true, i.e., IF A(R,S)=1.

Line 240 PRINTs SORRY, SEAT (the current value of S) IN ROW (the current value of R) IS RESERVED.

Line 250 loops back to line 160.

MultiDIMensional string arrays are very similar and no more difficult. Consider the following:

```

NEW
10 PRINT " (CLR/HOME) "
20 DIM N$(3,8)
30 FOR X=1 TO 8
40 READ N$(1,X)
50 READ N$(2,X)
60 READ N$(3,X)
70 NEXT
80 S=9
90 FOR X=1 TO 6:PRINT:NEXT
100 PRINT TAB(17);"COLUMN":PRINT:PRINT TAB(3);"ROW";
    TAB(9);1;TAB(17);2;TAB(27);3:PRINT
110 FOR X=1 TO 8
120 PRINT TAB(4);X;"SHIFT-H";TAB(S);N$(1,X);TAB(S*2);
    N$(2,X);TAB(S*3);N$(3,X)
130 NEXT
140 FOR X=1 TO 3000:NEXT
150 PRINT " (CLR/HOME) "
160 FOR X=1 TO 5:PRINT:NEXT
170 INPUT "WHICH ROW (1-8)";R
180 IF R<1 OR R>8 THEN 150
190 INPUT "WHICH COLUMN (1-3)";C
200 IF C<1 OR C>3 THEN 190
210 PRINT " (CLR/HOME) "
220 FOR X=1 TO 10:PRINT:NEXT

```

```

230 PRINT N$(C,R); "spSITS IN ROW";R; "spCOLUMN";C
240 FOR X=1 TO 3000:NEXT
250 PRINT " (CLR/HOME) ":GOTO 90
300 DATA ESTEE,DAVID,MICHAEL,HERM
310 DATA RUTH,LAURIE,JANET,ILLEANA
320 DATA PETER,SY,BOB,BILL
330 DATA RON,RUDI,MARTIN,SUSAN
340 DATA SAM,JUDY,PAUL,BARRY
350 DATA PAULA,CAROL,MARK,CHARLES

```

and **RUN**. With any luck, your display will appear as follows:

	COLUMN		
ROW	1	2	3
1	ESTEE	DAVID	MICHAEL
2	HERM	RUTH	LAURIE
3	JANET	ILLEANA	PETER
4	SY	BOB	BILL
5	RON	RUDI	MARTIN
6	SUSAN	SAM	JUDY
7	PAUL	BARRY	PAULA
8	CAROL	MARK	CHARLES

Then the screen clears, your Commodore requests you to **INPUT** row and column information, and then it **PRINTs** the name of the person occupying the specified seat.

### In Plain English . . .

```

10 PRINT " (CLR/HOME) "
20 DIM N$(3,8)
30 FOR X=1 TO 8
40 READ N$(1,X)
50 READ N$(2,X)
60 READ N$(3,X)
70 NEXT

```

**DIM N\$(3,8)** in line 20 reserves memory space for 8 rows, each with 3 seats, in multi**DIM**ensional string array **N\$**. (We'll once again ignore rows or seats designated by 0.)

Line 30 sets the parameters of **FOR/NEXT** loop from 1 **TO** 8.

Line 40 **READs** the first element of **DATA** (ESTEE in line 300) on the first loop, the fourth element of **DATA** (HERM) on the second loop, the seventh element of **DATA**

(JANET) on the third loop, etc., and stores each subsequent element in  $N\$(1,X)$ , forming the first column in the array. Therefore,

```

Row 1 Seat 1 = N$(1,1) = ESTEE
Row 1 Seat 2 = N$(1,2) = HERM
Row 1 Seat 3 = N$(1,3) = JANET
Row 1 Seat 4 = N$(1,4) = SY
Row 1 Seat 5 = N$(1,5) = RON
Row 1 Seat 6 = N$(1,6) = SUSAN
Row 1 Seat 7 = N$(1,7) = PAUL
Row 1 Seat 8 = N$(1,8) = CAROL

```

Line 50 **READS** the second element of **DATA** (DAVID in line 300) on the first loop, the fifth element of **DATA** (RUTH) on the second loop, the eighth element of **DATA** (ILLEANA) on the third loop, etc., and stores each subsequent element in  $N\$(2,X)$ , forming the second column.

Line 60 **READS** the third element of **DATA** (MICHAEL in line 300) on the first loop, the sixth element of **DATA** (LAURIE) on the second loop, the ninth element of **DATA** (PETER) on the third loop, etc., and stores each subsequent element in  $N\$(3,X)$ , forming the third column.

Line 70 recalls **FOR/NEXT** loop X and increases the value of X by 1.

```

80 S=9
90 FOR X=1 TO 6:PRINT:NEXT
100 PRINT TAB (17) ; "COLUMN" : PRINT: PRINT TAB (3) ; "ROW" ;
    TAB (9) ; 1 ; TAB (17) ; 2 ; TAB (27) ; 3 : PRINT
110 FOR X=1 TO 8
120 PRINT TAB (4) ; X ; "SHIFT-H" ; TAB (S) ; N$ (1, X) ; TAB (S*2) ;
    N$ (2, X) ; TAB (S*3) ; N$ (3, X)
130 NEXT

```

Lines 80 through 130 **PRINT** the seating diagram. Line 80 sets variable S equal to 9; this variable controls the **TAB** positions of the columns in the seating diagram.

Line 90 **PRINTS** six blank lines.

Line 100 **PRINTS** COLUMN at TAB 17, drops two lines, then **PRINTS** ROW at TAB 3, the number 1 at TAB 9, the number 2 at TAB 17, the number 3 at TAB 27, and finally a blank line. This sets up the table format of the display.

Line 110 sets the parameters of **FOR/NEXT** loop X from 1 TO 8.

Line 120 **PRINTS** the current value of X at TAB 4, a bold vertical line (SHIFT-H), the string value of  $N\$(1,X)$  at TAB S (9), the string value of  $N\$(2,X)$  at TAB S\*2, and the string value of  $N\$(3,X)$  at TAB S\*3.

Line 130 recalls **FOR/NEXT** loop X and increases the value of X by 1.

```

140 FOR X=1 TO 3000: NEXT
150 PRINT " (CLR/HOME) "
160 FOR X=1 TO 5: PRINT: NEXT
170 INPUT "WHICH ROW (1-8) "; R
180 IF R<1 OR R>8 THEN 150
190 INPUT "WHICH COLUMN (1-3) "; C
200 IF C<1 OR C>3 THEN 190

```

Lines 170 through 200 request **INPUT** information. Line 140 is an empty loop to slow the program.

Line 160 **PRINTs** five blank lines.

Line 170 requests you to **INPUT** which row, storing your **INPUT** in variable R.

Line 180 tests **INPUT** R and will loop back to line 150 only **IF** R is less than 1 **OR** greater than 8.

Line 190 requests you to **INPUT** which column, storing your **INPUT** in variable C.

Line 200 tests **INPUT** C and will loop back to line 190 only **IF** C is less than 1 **OR** greater than 3.

```

210 PRINT " (CLR/HOME) "
220 FOR X=1 TO 10: PRINT: NEXT
230 PRINT N$(C,R); "spSITS IN ROW"; R; "spCOLUMN"; C
240 FOR X=1 TO 3000: NEXT
250 PRINT " (CLR/HOME) ": GOTO 90

```

Line 220 **PRINTs** 10 blank lines.

N\$(C,R) in line 230 **PRINTs** the string value of the selected seat, then **PRINTs** SITS IN ROW (the current value of R) COLUMN (the current value of C).

Line 240 is an empty loop.

Line 250 clears the display, then loops back to line 90.

```

300 DATA ESTEE, DAVID, MICHAEL, HERM
      :
      :
350 DATA PAULA, CAROL, MARK, CHARLES

```

Lines 300 through 350 store the **DATA**, a total of 24 names.

## CHAPTER 12 REVIEW

---

Array	Single-dimension array
<b>DIM</b>	String array
Element	Subscript
Multidimensional array	

---

**PRACTICE EXERCISES**

---

- 12-1. Write a program that will (a) simulate the roll of a pair of dice 10 times; (b) display the individual face value of each die and the total for each roll; (c) display the total number of 1's, 2's, 3's, 4's, 5's, and 6's rolled.
- 12-2. *Hit or Miss.* Write a program that will generate a random four-digit number, with each digit having a value from 1 to 4. Give yourself 8 chances to guess the number. Have the computer tell you how many HITS (right digits in the right places) you score for each guess and then tell you the right number after eight incorrect guesses.
-

---

## CHR\$, ASC, and PEEK and POKE

Your Commodore has several built-in functions that will help you keep track of where information is being stored.

---

### Character String (CHR\$) and ASCII Codes (ASC)

Each character on the Commodore 64 keyboard has a built-in ASCII code number assigned to it. The following program will list the alphabet and codes for you:

```

10 PRINT CHR$(147)
20 FOR X=65 TO 90
30 PRINT "ASCII CODE NUMBER"; X; " =spLETTERsp-sp"; CHR$(X)
40 FOR P=1 TO 1000:NEXT
50 PRINT CHR$(147)
60 NEXT
70 FOR X=1 TO 26
80 READ L$
90 PRINT "ASCII CODE # OF LETTERsp-sp"; L$; "sp="; ASC(L$)
100 FOR P=1 TO 1000:NEXT
110 PRINT CHR$(147)
120 NEXT
130 DATA A,B,C,D,E,F,G,H,I,J,K,L,M
140 DATA N,O,P,Q,R,S,T,U,V,W,X,Y,Z

```

and RUN. This line appears:

```
ASCII CODE NUMBER 65 = LETTER - A
```

Each loop changes the ASCII CODE NUMBER (65–90) and the LETTER (A–Z). When the entire alphabet has **RUN**, the line changes to

```
ASCII CODE # OF LETTER - A = 65
```

Again, each loop changes the letter (A–Z) and the ASCII code number (65–90).

### In Plain English . . .

```
10 PRINT CHR$(147)
20 FOR X=65 TO 90
30 PRINT "ASCII CODE NUMBER"; X; "=spLETTERsp-sp"; CHR$(X)
40 FOR P=1 TO 1000: NEXT
50 PRINT CHR$(147)
60 NEXT
```

**PRINT CHR\$(147)** in line 10 clears the display. The function **CHR\$(any number from 0–255)** returns the *character string* value of the number in parentheses. **CHR\$(147)** corresponds to the **CLR/HOME** key. Each character accessible from your Commodore 64 keyboard has an assigned ASCII Code Number from 0–255. Try the following example: *Type PRINT CHR\$(65) in Immediate Mode and press RETURN.* The letter A should appear. The letter A is assigned the ASCII Code Number 65. Now *type PRINT CHR\$(66) and press RETURN.* The letter B will appear. The letter B is assigned the ASCII Code Number 66. ASCII Code Numbers 65–90 correspond to the entire alphabet. A complete list of ASCII and **CHR\$** Codes can be found in Appendix 3.

Lines 20–60 **PRINT** the entire alphabet and the corresponding ASCII Code Numbers (65–90). Line 20 sets the parameters of **FOR/NEXT** loop X from 65 **TO** 90. Line 30 **PRINTs** ASCII CODE NUMBER (the current value of X) equal to LETTER—the character string value of X—**CHR\$(X)**. Line 40 is an empty loop to slow the program and line 60 recalls **FOR/NEXT** loop X and increases the value of X by 1.

```
70 FOR X=1 TO 26
80 READ L$
90 PRINT "ASCII CODE # OF LETTERsp-sp"; L$;
   "sp="; ASC(L$)
100 FOR P=1 TO 1000: NEXT
110 PRINT CHR$(147)
120 NEXT
130 DATA A, B, C, D, E, F, G, H, I, J, K, L, M
140 DATA N, O, P, Q, R, S, T, U, V, W, X, Y, Z
```

Line 70 sets the parameters of **FOR/NEXT** loop X from 1 to 26.

```

30 FOR X=1 TO 8
40 READ N$(1, X)
50 READ N$(2, X)
60 READ N$(3, X)
70 NEXT

```

**DIM N\$(3,8)** in line 20 reserves memory space for 8 rows, each with 3 seats, in multidimensional string array N\$. (We'll once again ignore rows or seats designated by 0.)

Line 30 sets the parameters of **FOR/NEXT X** from 1 TO 8.

Line 40 **READs** the first element of **DATA** (ESTEE in line 300) on the first loop, the fourth element of **DATA** (HERM) on the second loop, the seventh element of **DATA** (JANET) on the third loop, etc., and stores each subsequent element in N\$(1,X), forming the first column in the array. Therefore,

```

Row 1 seat 1 = N$(1,1) = ESTEE
Row 1 seat 2 = N$(1,2) = HERM
Row 1 seat 3 = N$(1,3) = JANET
Row 1 seat 4 = N$(1,4) = SY
Row 1 seat 5 = N$(1,5) = RON
Row 1 seat 6 = N$(1,6) = SUSAN
Row 1 seat 7 = N$(1,7) = PAUL
Row 1 seat 8 = N$(1,8) = CAROL

```

Line 50 **READs** the second element of **DATA** (JUAN in line 300) on the first loop, the fifth element of **DATA** (RUTH) on the second loop, the eighth element of **DATA** (ILLEANA) on the third loop, etc., and stores each subsequent element in N\$(2,X), forming the second column.

Line 60 **READs** the third element of **DATA** (MICHAEL in line 300) on the first loop, the sixth element of **DATA** (LAURIE) on the second loop, the ninth element of **DATA** (PETER) in the third loop, etc., and stores each subsequent element in N\$(3,X), forming the third column.

Line 70 recalls **FOR/NEXT** loop X and increases the value of X by 1.

```

80 S=9
90 FOR X=1 TO 6:PRINT:NEXT
100 PRINT TAB(17); "COLUMN":PRINT:PRINT TAB(2); "ROW";
    TAB(11); 1; TAB(18); 2; TAB(28); 3:PRINT
110 FOR X=1 TO 8
120 PRINT TAB(4); X; TAB(S); N$(1, X); TAB(S*2); N$(2, X)
    TAB(S*3); N$(3, X)
130 NEXT

```



Lines 80 through 130 **PRINT** the seating diagram. Line 80 sets variable S equal to 9; this variable controls the position of the headings in the seating diagram.

Line 90 **PRINTs** six blank lines.

Line 100 **PRINTs** COLUMN at **TAB** 17, drops two lines, then **PRINTs** ROW at **TAB** 2, the number 1 at **TAB** 11, the number 2 at **TAB** 18, the number 3 at **TAB** 28, and finally a blank line. This sets up the table format of the display.

Line 110 sets the parameters of **FOR/NEXT** loop X from 1 to 8.

Line 120 **PRINTs** the current value of X at **TAB** 4, the string value of N\$(1,X) at **TAB** S (9), the string value of N\$(2,X) at **TAB** S\*2, and the string value of N\$(3,X) at **TAB** S\*3.

Line 130 recalls **FOR/NEXT** loop X and increases the value of X by 1.

```

140  FOR X=1 TO 3000:NEXT
150  HOME
160  FOR X=1 TO 5:PRINT:NEXT
170  INPUT "WHICH ROW (1-8) sp";R
180  IF R<1 OR R>8 THEN 150
190  INPUT "WHICH COLUMN (1-3) sp";C
200  IF C<1 OR C>3 THEN 190

```

Lines 140 through 200 request **INPUT** information. Line 140 is an empty loop to slow the program.

Line 160 **PRINTs** five blank lines.

Line 170 requests you to **INPUT** which row, storing your **INPUT** in variable R.

Line 180 tests **INPUT** R and will loop back to line 150 only **IF** R is less than 1 **OR** greater than 8.

Line 190 requests you to **INPUT** which column, storing your **INPUT** in variable C.

Line 200 tests **INPUT** C and will loop back to line 190 only **IF** C is less than 1 **OR** greater than 3.

```

210  HOME
220  FOR X=1 TO 10:PRINT:NEXT
230  PRINT N$(C,R); "spSITS IN ROW";R; "spCOLUMN";C
240  FOR X=1 TO 3000:NEXT
250  HOME:GOTO 90

```

Line 220 **PRINTs** 10 blank lines.

N\$(C,R) in line 230 **PRINTs** the string value of the selected seat, then **PRINTs** SITS IN ROW (the current value of R) COLUMN (the current value of C).

Line 240 is an empty loop to create a pause.

Line 250 clears the display then loops back to line 90.

```

1026 = 3
1027 = 4
1028 = 5
1029 = 6
1030 = 7
1031 = 8
1032 = 9
1033 = 10

```

### In Plain English . . .

```

10 PRINT CHR$(147)
20 FOR X=1 TO 5:PRINT: NEXT
30 FOR X=1024 TO 1033
40 READ N

```

Line 20 **PRINTs** five blank lines.

Line 30 sets the parameters of **FOR/NEXT** loop X from 1024 to 1033.

Line 140 **READs** the **DATA** in line 100, one element per loop, and stores it in variable N.

```

50 POKE X,N
60 PRINT TAB(13);X;"=sp";PEEK(X)
70 NEXT
100 DATA 1,2,3,4,5,6,7,8,9,10

```

**POKE X,N** in line 50 inserts the current value of variable N into the memory cell at the address determined by the current value of X. Each memory cell in your Commodore 64 has its own specific address. **POKE X,N** stores the number 1 in the memory cell at address 1024 on the first loop, the number 2 at address 1025 on the second loop, the number 3 at address 1026 on the third loop, and so on. The command **POKE** is always followed by an address, a comma, then the value to be stored (0–255). So, **POKE X,N** stores the value of N in the memory cell at address X. You are, in essence, **POKEing** a specific value (N) into a specific address (X).

Line 60 **PRINTs** the current value of X at **TAB** position 13, and then, by using **PEEK X**, **PRINTs** the current value of the memory cell at address X. The command **PEEK** returns the value of the memory cell as determined by the address in parentheses. You are, in essence, **PEEKing** inside a memory cell to see what's there.

Line 70 recalls **FOR/NEXT** loop X and increases the value of X by 1.

Line 100 stores the **DATA**, numbers 1 through 10.

Later chapters detail more specific uses of **PEEK** and **POKE**.

**CHAPTER 13 REVIEW**

---

ASC  
CHR\$  
PEEK  
POKE

---

**PRACTICE EXERCISES**

---

- 13-1. Write a program to **POKE** any **INPUT** number (between 0–255) into memory addresses 35001, 35002, and 35003.
  - 13-2. Amend the above program to **PRINT** the characters corresponding to the numerical values now stored in memory addresses 35001, 35002, and 35003.
-

---

# Graphics

---

## COLOR

You can access 16 different colors on your Commodore 64. Follow these steps:

1. *Press* **SHIFT-CLR/HOME** to clear the display.
2. *Press* **CTRL-9** to enter inverse mode.
3. *Press* **CTRL-1**. The cursor changes from light blue to black. Now *press* **SPACE** 5 times.
4. *Press* **CTRL-2**. The cursor changes from black to white. Now *press* **SPACE** 5 times.
5. *Press* **CTRL-3**. The cursor changes from white to red. Now *press* **SPACE** 5 times.
6. *Press* **CTRL-4**. The cursor changes from red to cyan. Now *press* **SPACE** 5 times.
7. *Press* **CTRL-5**. The cursor changes from cyan to purple. Now *press* **SPACE** 5 times.
8. *Press* **CTRL-6**. The cursor changes from purple to green. Now *press* **SPACE** 5 times.
9. *Press* **CTRL-7**. The cursor changes from green to blue. Now *press* **SPACE** 5 times (you won't see anything because of the blue background).

10. Press **CTRL-8**. The cursor changes from blue to yellow. Now *press* **SPACE** 5 times.

As demonstrated above, the 8 colors accessed via the **CTRL** key are:

**CTRL-1** = black  
**CTRL-2** = white  
**CTRL-3** = red  
**CTRL-4** = cyan  
**CTRL-5** = purple  
**CTRL-6** = green  
**CTRL-7** = blue  
**CTRL-8** = yellow

Pretty, isn't it? But we're not done yet. Repeat Step 3. But this time replace the **CTRL** key with the **COMMODORE** key (lower-left corner of the keyboard). The 8 colors accessed via the **COMMODORE** key are

**COMMODORE-1** = orange  
**COMMODORE-2** = brown  
**COMMODORE-3** = light red  
**COMMODORE-4** = gray 1  
**COMMODORE-5** = gray 2  
**COMMODORE-6** = light green  
**COMMODORE-7** = light blue  
**COMMODORE-8** = gray 3

Now you know how to access all 16 colors via Immediate Mode. Another handy thing to know is how to **RESTORE** your display to the normal light blue on dark blue. Simply *press* **RUN/STOP-RESTORE**.

There are several different methods of incorporating color into your programs. Try the following program:

```
10 PRINT CHR$(147)
20 FOR X=1 TO 5:PRINT:NEXT
30 PRINT "THE NEXT LINE WILL APPEAR...":PRINT
40 PRINT "CTRL-1 I CTRL-2 N CTRL-7 SPACE CTRL-3 C CTRL-4 O
CTRL-5 L CTRL-6 O CTRL-8 R"
```

and **RUN**. Each letter in the words IN COLOR will **PRINT** in a different color.

Each color in program line 40 is represented by a different graphic symbol: your display now reflects the latest color entered into the program (yellow, in this instance). *Press* **RUN/STOP-RESTORE** to return everything to normal.

Each color has its own ASCII Code Number as the following program demonstrates.

```

NEW
10 PRINT CHR$(147)
20 FOR X=1 TO 8
30 READ C
40 PRINT TAB(15); CHR$(18); CHR$(C); "CTRL"; X
50 NEXT
100 DATA 144, 5, 28, 159, 156, 30, 31, 158

```

and **RUN**. Your program will list **CTRL-1** through **CTRL-8**, each in its corresponding color (the 7 will be blue on blue, and therefore invisible).

### In Plain English . . .

Line 20 sets the parameters of **FOR/NEXT** loop X from 1 **TO** 8.

**READ C** in line 30 **READS** the **DATA** from line 100, storing it, one element per loop, in variable C.

**CHR\$(18)** in line 40 enters inverse mode. **CHR\$(C)** represents the color value as determined by the current value of variable C. **CTRL** (the current value of X) **PRINTS** at **TAB** position 15 in the color determined by the current value of **CHR\$(C)**.

Line 50 recalls **FOR/NEXT** loop X, increases the value of X by 1, and **READS** the next element of **DATA**.

Line 100 stores the **DATA**, the various ASCII code numbers of the **CTRL-1** through -8 colors.

**note:** **RUN** the above program, then *press* **SHIFT-COMMODORE**. Your display shifts from upper to lower case. *Press* **SHIFT-COMMODORE** again and the display returns to upper case.

Perhaps the easiest, most versatile method of accessing colors on your Commodore is via **PEEK/POKE**. Try the following program:

```

NEW
10 PRINT CHR$(147)
20 INPUT "ENTER MESSAGE (40 CHAR. MAX.) "; M$
30 T=20-LEN(M$)/2
40 FOR BDR=0 TO 15
50 FOR BKG=0 TO 15
60 POKE 53280, BDR
70 POKE 53281, BKG
80 FOR X=1 TO 10: PRINT: NEXT
90 PRINT TAB(T); M$

```

```
100 FOR X=1 TO 200:NEXT
110 NEXT BKG:NEXT BDR
```

and **RUN**. Your message will scroll while the background and border colors change. The background color will change 16 times for each change of the border color. The program will halt after the border color has changed 16 times.

**In Plain English . . .**

```
10 PRINT CHR$(147)
20 INPUT "ENTER MESSAGE (40 CHAR. MAX. ) "; M$
30 T=20-LEN(M$) / 2
40 FOR BDR=0 TO 15
50 FOR BKG=0 TO 15
```

Line 20 requests that you enter a message and stores your **INPUT** in string variable M\$.

Line 30 is a formula used to center a string of any **LEN**gth. **IF** M\$ = "SHALOM", **THEN** the formula would read  $T = 20 - 6/2$ . Therefore, SHALOM would **PRINT** (line 90) at **TAB** (T) position 17.

Line 40 sets the parameters of **FOR/NEXT** loop BDR (for border) from 0 **TO** 15.

Line 50 sets the parameters of **FOR/NEXT** loop BKG (for background) from 0 **TO** 15.

```
60 POKE 53280, BDR
70 POKE 53281, BKG
```

Line 60 **POKEs** the current value of variable BDR into location 53280, the address that determines the border color. Try the following test: *Type* **POKE 53280,0** and *press* **RETURN**. The border will change to black.

Line 70 **POKEs** the current value of variable BKG into location 53281, the address that determines the background color. Try the following test: *Type* **POKE 53281,0** and *press* **RETURN**. Now the background will also be black. The **POKE** values for each color are

POKE	Color	POKE	Color
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

```

80 FOR X=1 TO 10:PRINT:NEXT
90 PRINT TAB(T);M$
100 FOR X=1 TO 200:NEXT
110 NEXT BKG:NEXT BDR

```

Line 80 **PRINTs** 10 blank lines.

Line 90 **PRINTs** the value of string variable M\$ at the **TAB** position determined by the value of variable T.

Line 100 is an empty loop to slow the program.

Line 110 first recalls **FOR/NEXT** loop BKG, then **FOR/NEXT** loop BDR. In essence, **FOR/NEXT** loop BKG must loop 16 times for every loop of BDR.

---

## Graphic Symbols

Most keys on your Commodore 64 keyboard have two graphic symbols on the front. The symbol on the right front of each key can be accessed by pressing **SHIFT**. For instance,

```

SHIFT-A = Spade
SHIFT-S = Heart
SHIFT-Z = Diamond
SHIFT-X = Club

```

The graphic symbol on the left front of each key is accessed via the **COMMODORE** key: **COMMODORE-A** and **COMMODORE-S** together form a bracket opening downward; **COMMODORE-Z** and **COMMODORE-X** form a bracket opening upward.

---

## YOU CAN'T GET THERE FROM HERE

### Display Memory Map

The display on your Commodore 64 is *memory-mapped*. Simply stated, a separate memory address exists for each of the 1000 (40 columns per line times 25 lines) possible character locations (see Figure 1). The starting address (upper-left corner location) for your display is 1024. The last address (lower-right corner) is 2023.

Similarly, each of the 1000 possible character locations has a corresponding color memory address (see Figure 2), allowing you to determine the color of any character





		Column																													
		0	1	2	3	4	5							34	35	36	37	38	39												
Row	0	55296	55297	55298	55299	55300	55301																								
	1	55336												55330	55331	55332	55333	55334	55335												
	2	55376																													
	3	55416																													
	4	55456																													
	5	55496																													
	6	55536																													
	7	55576																													
	8	55616																													
	9	55656																													
	10	55696																													
	11	55736																													
	12	55776																													
	14	55816																													
	14	55856																													
	15	55896																													
	16	55936																													
	17	55976																													
	18	56016																													
	19	56056																													
	20	56096																													
	21	56136																													
	22	56176																													
	23	56216																													
	24	56256	56257	56258	56259	56260	562																								
																			56290	56291	56292	56293	56294	56295							

Figure 2. Commodore 64 color memory map, 40 columns × 25 lines.

at any screen address. The starting address (upper-left corner) for the color memory is 55296. The last address (lower-right corner) is 56295.

The following program will help you get the feel of things.

**NEW**

```

10 PRINT CHR$(147)
20 CM=55296
30 DM=1024
40 INPUT "ENTER ROW (0-24) ";R
50 IF R<0 OR R>24 THEN 40
60 INPUT "ENTER COLUMN (0-39) ";C
70 IF C<0 OR C>39 THEN 60
80 INPUT "ENTER COLOR CODE (0-15) ";CC
90 IF CC<0 OR CC>15 THEN 80
100 PRINT CHR$(147)
110 POKE DM+(R*40)+C, 81
120 POKE CM+(R*40)+C, CC
130 FOR X=1 TO 3000:NEXT
140 GOTO 10

```

and **RUN**. **INPUT** the row and column numbers when requested. A single dot will appear, corresponding to your row and column coordinates and color designation.

### In Plain English . . .

```

10 PRINT CHR$(147)
20 CM=55296
30 DM=1024

```

Line 20 sets variable CM to equal the starting address of the color memory map.

Line 30 sets variable DM equal to the starting address of the display memory map.

```

40 INPUT "ENTER ROW (0-24) ";R
50 IF R<0 OR R>24 THEN 40
60 INPUT "ENTER COLUMN (0-39) ";C
70 IF C<0 OR C>39 THEN 60
80 INPUT "ENTER COLOR CODE (0-15) ";CC
90 IF CC<0 OR CC>15 THEN 80

```

Line 40 requests you to enter the row number, storing your **INPUT** in variable R. Line 50 tests **INPUT** R and will loop back to program line 40 only **IF** R is less than 0 **OR** greater than 24.

Line 60 requests you to enter the column number, storing your **INPUT** in variable C. Line 70 tests **INPUT** C and will loop back to program line 60 only **IF** C is less than 0 **OR** greater than 39.

Line 80 requests you to enter the color code number, storing your **INPUT** in variable CC. Line 90 tests **INPUT** CC and will loop back to program line 80 only **IF** CC is less than 0 **OR** greater than 15.

```

100 PRINT CHR$(147)
110 POKE DM+(R*40)+C, 81
120 POKE CM+(R*40)+C, CC
130 FOR X=1 TO 3000: NEXT
140 GOTO 10

```

Line 110 **POKE**s the character 81 (the dot) into the display memory map address located at  $DM + (R * 40) + C$ . If, for instance, you enter row 1, column 1, the display memory address would be  $1024 + (1 * 40) + 1 = 1065$ . Row 2, column 2 would **PRINT** the dot at display memory address  $1024 + (2 * 40) + 2 = 1106$ . The dot is represented by the number 81. Each memory address can store a number from 0 through 255, and each number represents a different character (see Appendix 3).

Line 120 uses virtually the same formula as line 110 to determine the corresponding color memory address. Row 1, column 1 corresponds to color memory address of  $55296 + (1 * 40) + 1 = 55337$ . Row 2, column 2 corresponds to color memory address of  $55296 + (2 * 40) + 2 = 55378$ .

Line 130 is an empty loop to slow the program.

Line 140 loops back to line 10 to request new coordinates.

Now try the following program:

```

NEW
10 PRINT CHR$(147)
20 CM=55296
30 FOR DM=1024 TO 2023
40 CC=INT(16*RND(1))
50 POKE DM, 81
60 POKE CM, CC
70 CM=CM+1
80 NEXT
90 GOTO 10

```

and **RUN**. The entire display memory map fills with random-color dots.

**In Plain English . . .**

```

10 PRINT CHR$(147)
20 CM=55296
30 FOR DM=1024 TO 2023

```

Line 20 sets variable CM to equal the starting address of the color memory map.

Line 30 sets the parameters of **FOR/NEXT** loop DM (for display memory map) from 1024 TO 2023—the sequential addresses for the display memory map locations.

```

40 CC=INT(16*RND(1))
50 POKE DM, 81
60 POKE CM, CC
70 CM=CM+1
80 NEXT
90 GOTO 10

```

Line 40 sets variable CC (color code) to equal a random integer from 0–15.

Line 50 **POKEs** the character value 81 (a dot) into the display memory map address determined by the current value of DM.

Line 60 **POKEs** the color code determined by the current value of CC into the color memory address determined by the current value of CM.

Line 70 adds 1 to the current value of CM with each loop.

Line 80 recalls **FOR/NEXT** loop DM and increases the value of DM by 1.

Line 90 loops back to line 10 to begin again.

**ON THE MOVE****Moving Graphics**

The theory behind most moving graphics is simple: **PRINT** an object, erase it, then **PRINT** it somewhere else. If you repeat this process quickly enough, you create the illusion of movement, as the following program demonstrates:

```

NEW
10 PRINT CHR$(147)
20 CM=55696
30 FOR DM=1424 TO 1463
40 POKE DM, 81

```

```

50 POKE CM, 1
60 FOR X=1 TO 3:NEXT
70 POKE DM, 32
80 CM=CM+1
90 NEXT DM

```

and **RUN**. A little white ball flashes across the screen.

### In Plain English . . .

```

10 PRINT CHR$(147)
20 CM=55696
30 FOR DM=1424 TO 1463

```

Line 20 sets variable CM (color memory map) equal to 55696 (the equivalent of row 10, column 0).

Line 30 sets the parameters of **FOR/NEXT** loop DM (display memory map) from 1424 **TO** 1463 (the equivalent of row 10, columns 0-39).

```

40 POKE DM, 81
50 POKE CM, 1
60 FOR X=1 TO 3:NEXT
70 POKE DM, 32
80 CM=CM+1
90 NEXT DM

```

Line 40 **POKEs** the character value 81 (a dot) into the display memory map address determined by the current value of DM.

Line 50 **POKEs** the color code value 1 (white) into the color memory map address determined by the current value of CM.

Line 60 is an empty loop to slow the display. See what happens when you **RUN** the program without line 60.

Line 70 **POKEs** the character value 32 (**SPACE**) into the display memory map address determined by the current value of DM. Essentially, line 70 “erases” the dot **PRINTed** earlier by line 40. (Remember the theory—**PRINT**, erase, **PRINT**.)

Line 80 adds 1 to the current value of CM.

Line 90 recalls **FOR/NEXT** loop DM and increases the value of DM by 1.

Now add the following lines to the above program:

```

100 CM=55735
110 FOR DM=1463 TO 1424 STEP-1

```

```
120 POKE DM, 81
130 POKE CM, 1
140 FOR X=1 TO 3: NEXT
150 POKE DM, 32
160 CM=CM-1
170 NEXT DM
180 GOTO 20
```

and **RUN**. Now the little ball bounces back and forth across the screen.

### In Plain English . . .

```
100 CM=55735
110 FOR DM=1463 TO 1424 STEP-1
```

Line 100 sets variable **CM** (color memory map) equal to 55735 (the equivalent of row 10, column 39).

Line 110 sets the parameters of **FOR/NEXT** loop **DM** (display memory map) from 1463 **TO** 1424 (the equivalent of row 10, columns 39-0), moving backward in increments of 1 (**STEP-1**).

```
120 POKE DM, 81
130 POKE CM, 1
140 FOR X=1 TO 3: NEXT
150 POKE DM, 32
160 CM=CM-1
170 NEXT DM
180 GOTO 20
```

Line 120 **POKEs** the character value 81 into the display memory map address determined by the current value of **DM**.

Line 130 **POKEs** the color code value 1 into the color memory map address determined by the current value of **CM**.

Line 140 is an empty loop to slow the program a bit.

Line 150 **POKEs** a blank **SPACE** (32) into the display memory map address determined by the current value of **DM**, thereby “erasing” the ball.

Line 160 subtracts 1 from the current value of **CM**.

Line 170 recalls **FOR/NEXT** loop **DM** and decreases the value of **DM** by 1 (**STEP-1**).

Line 180 loops back to line 20 to keep bouncing.

Here's another example:

```

NEW
10 PRINT CHR$(147)
20 CM=55316
30 DM=1044
40 FOR B=0 TO 24
50 POKE DM, 81
60 POKE CM, 1
70 CM=CM+40: DM=DM+40
80 NEXT

```

and **RUN**. This time, the little ball plummets from the top to the bottom of the screen, leaving a trail as it falls.

### In Plain English . . .

Line 20 sets variable **CM** (color memory) equal to 55316 (the equivalent of row 0, column 20).

Line 30 sets variable **DM** (display memory) equal to 1044 (the equivalent of row 0, column 20).

Line 40 sets the parameters of **FOR/NEXT** loop **D** (for descent) from **0 TO 24**.

Line 50 **POKEs** the character value 81 into the display memory address determined by the current value of **DM**.

Line 60 **POKEs** the color code 1 (white) into the color memory address determined by the current value of **CM**.

Line 70 adds 40 to the current values of **CM** and **DM**.

Line 80 recalls **FOR/NEXT** loop **D** and adds 1 to the value of **D**.

Now add the following lines:

```

90 DM=2004
100 FOR A=24 TO 0 STEP-1
110 POKE DM, 32
120 DM=DM-40
130 NEXT
140 GOTO 10

```

and **RUN**. A yo-yo—enough to give you a headache.



**In Plain English . . .**

Line 90 sets variable DM (display memory) equal to 2004 (the equivalent of row 24, column 20).

Line 100 sets the parameters of **FOR/NEXT** loop A (for ascent) from 24 **TO** 0, moving backward at increments of 1 (**STEP**-1).

Line 110 **POKEs** a blank **SPACE** (32) into the current display memory address determined by the current value of DM, thereby “erasing” the white ball at that address.

Line 120 subtracts 40 from the current value of DM.

Line 130 recalls **FOR/NEXT** loop A and decreases the value of A by 1 (**STEP**-1).

Line 140 loops back to line 10 to go again.

**CHAPTER 14 REVIEW**

---

Character code	<b>POKE</b> 1024–2023(display memory)
Color code 0–15	<b>POKE</b> 53280(border color address)
<b>COMMODORE</b> -key graphic symbol	<b>POKE</b> 53281(background color address)
<b>COMMODORE</b> -1 through -9 colors	<b>POKE</b> 55296–56295(color memory)
<b>CTRL</b> -1 through -9 colors	<b>SHIFT-COMMODORE</b> (Upper/lowercase)
Memory-mapped	<b>SHIFT</b> -key graphic symbol
Moving graphics	

---

**PRACTICE EXERCISES**

---

- 14-1. Write a program to construct a four-walled playing court, then send a ball bouncing all around the court.
- 14-2. Write a program to generate different random size and color rectangles.
-

## Sprite Graphics

*Sprite graphics* are high-resolution images or shapes that you design, store in memory, then recall at the appropriate time and place. You control not only sprite shapes and colors, but movement as well, affording you a world of creative graphic possibilities (my favorite kind).

So much for the good news. Now for the bad: There is a major disadvantage in programming sprites, a price to pay for all that flexibility. To wit: sprites are far more difficult and time-consuming to configure and employ than regular graphics. But let's throw caution to the winds and give it a whirl.

A *sprite* is a user-defined figure composed of 504 dots in a field 24 dots wide by 21 dots long ( $24 \times 21 = 504$ ). The sprite display grid is 320 dots wide by 220 dots long, and your Commodore can handle up to 8 sprites at one time on this display. Each dot in the sprite is represented in the computer's memory by one *bit*—the smallest unit of memory storage—and each bit can be either “on” (dot present), or “off” (dot absent). An appropriate combination of dots and spaces will produce a fair representation of almost any object. In addition, you can design a single sprite in up to four colors, although the horizontal and vertical resolution of the object is decreased (see the *Commodore 64 Programmer's Reference Guide*).

A linear group of 8 dots (8 bits) represents one *byte*. Each horizontal row of 24 bits—a single sprite line—equals 3 bytes of information. You specify the dots that are “on” in each byte with a single number. Consider the graphic representation of a byte:

1 Byte

8 Bits

128	64	32	16	8	4	2	1
-----	----	----	----	---	---	---	---

To generate an alternating pattern of “on” (shaded) and “off” (blank) dots,



simply add the numbers that represent the “on” bits:

$$128 + 32 + 8 + 2 = 170$$

You should note that any pattern is represented by one and only one number, and that 255 patterns are possible in a single byte. A series of three numbers representing 3 bytes specifies the pattern for a single sprite line: 21 such lines define a total sprite.

This may sound complicated, but don't panic: With some graph paper and a few simple calculations, you can have your Commodore jumping through hoops.

Try the following program:

```

NEW
5 POKE 53281,0
10 PRINT CHR$(147)
20 PRINT TAB(12);"THE RUNNING NOSE"
30 DC=53248
40 SO=21+DC
50 S1=2
60 POKE SO,S1
70 POKE 2041,13
80 FOR D=0 TO 62
90 READ SD
100 POKE D+832,SD
110 NEXT
120 FOR HM=1 TO 255
130 POKE DC+2,HM
140 POKE DC+3,100
150 NEXT
160 GOTO 120
200 DATA 1,192,0,1,224,0,1,240,0
210 DATA 1,248,0,1,252,0,1,254,0
220 DATA 1,255,0,1,255,128,1,255,192
230 DATA 1,255,224,1,255,240,1,255,248
240 DATA 1,255,252,1,255,254,1,192,126
250 DATA 1,192,62,0,255,252,0,0,64
260 DATA 0,1,192,0,1,192,0,1,192

```

and **RUN**. An utterly ridiculous running nose dashes across the screen.

### In Plain English . . .

```

5 POKE 53281,0
10 PRINT CHR$(147)
20 PRINT TAB(12);"THE RUNNING NOSE"
30 DC=53248

```

```

40 SO=21+DC
50 S1=2

```

**POKE** 53281,0 in line 5 changes (**POKEs**) the background color (0 = black) into address 53281.

Line 20 **PRINTs** THE RUNNING NOSE at **TAB** position 12.

Line 30 sets variable DC (for Display Chip) to 53248. Address 53248 is the location of the display chip.

Variable SO (mnemonic for Sprite On) in line 40 is set to the value of DC plus 21. The address to turn on any sprite is always 53248 + 21.

Variable S1 (for Sprite #1) in line 50 is set to 2. You can display up to 8 different sprites on the same screen. Sprite values are

```

Sprite #0 = 1
Sprite #1 = 2
Sprite #2 = 4
Sprite #3 = 8
Sprite #4 = 16
Sprite #5 = 32
Sprite #6 = 64
Sprite #7 = 128

```

```

60 POKE SO, S1
70 POKE 2041, 13

```

Line 60 **POKEs** the value of sprite #1 into the memory address for Sprite On, turning sprite #1 on.

Line 70 **POKEs** 13 into address 2041. The number 13 represents the storage “block” equivalent of 1 of 3 possible sprite **DATA** storage locations. The 3 possible sprite **DATA** storage locations are 832, 896, and 960. Each sprite comprises 63 bytes of information (3 bytes per line × 21 lines), and 832/63 = 13. Likewise, the storage block equivalent for sprite **DATA** storage location 896 is 14 (896/63), while the storage block equivalent for sprite **DATA** storage 960 is 15 (960/63).

The number 2041 in line 70 is the address for the *sprite pointer*, which “points to” the memory storage location of a sprite. Each sprite has its own pointer location:

```

Sprite #0 = 2040
Sprite #1 = 2041
Sprite #2 = 2042
Sprite #3 = 2043
Sprite #4 = 2044
Sprite #5 = 2045
Sprite #6 = 2046
Sprite #7 = 2047

```

```

80 FOR D=0 TO 62
90 READ SD
100 POKE D+832, SD
110 NEXT

```

Line 80 sets the parameters of **FOR/NEXT** loop D (for **DATA**) from 0 TO 62.

Line 90 **READS** the sprite **DATA** (SD) contained in program lines 200–260, 1 element per loop, for a total of 63 (0 TO 62) elements. Remember, each sprite comprises 63 bytes of information.

**POKE D + 832,SD** in line 100 **POKEs** the current value of SD into the sprite **DATA** location address determined by the current value of D added to 832. Thus, the first 9 sprite **DATA** locations (832 through 840) would contain the following **DATA** (as **READ** from program line 200):

Location		DATA
832	=	1
833	=	192
834	=	0
835	=	1
836	=	224
837	=	0
838	=	1
839	=	240
840	=	0

Line 110 recalls **FOR/NEXT** loop D and increases the value of D by 1.

```

120 FOR HM=1 TO 255
130 POKE DC+2, HM
140 POKE DC+3, 100
150 NEXT
160 GOTO 120

```

Line 120 sets the parameters of **FOR/NEXT** loop HM (for Horizontal Movement) from 1 TO 255. Recall that the display grid for sprites is 320 dots horizontally (across) by 220 dots vertically (up and down). Sprites are moved along X,Y coordinates. The X axis controls horizontal movement, while the Y axis controls vertical movement.

**POKE DC + 2,HM** in line 130 **POKEs** the current X-axis value (HM) into the address located at DC + 2. **POKE DC + 3,100** in line 140 **POKEs** the Y-axis value (100) into the address located at DC + 3. A *register* is a byte or series of bytes that performs a specific task. Each sprite 0 through 7 is assigned a pair of X- and Y-axis registers as follows:

	<u>X-Axis Register</u>	<u>Y-Axis Register</u>
Sprite 0	0	1
Sprite 1	2	3
Sprite 2	4	5
Sprite 3	6	7
Sprite 4	8	9
Sprite 5	10	11
Sprite 6	12	13
Sprite 7	14	15

Therefore, the X-axis register for sprite 1 is 2, as noted by line 130 (DC + 2), and the Y-axis register for sprite 1 is 3, as noted by line 140 (DC + 3).

Line 150 recalls **FOR/NEXT** loop HM and increases the value of HM by 1.

Line 160 loops back to program line 120 to continually move THE RUNNING NOSE across the screen.

```

200 DATA 1,192,0,1,224,0,1,240,0
      :
      :
260 DATA 0,1,192,0,1,192,0,1,192

```

Lines 200 through 260 store the sprite **DATA**.

Now consult Figure 3, the actual sprite model for THE RUNNING NOSE. The 24 columns across are divided into 3 bytes, and three bytes per line multiplied by 21 lines totals 63 bytes per sprite. As mentioned earlier, each byte in your Commodore's memory is divided into 8 bits, and each bit has one of the following decimal equivalents: 128, 64, 32, 16, 8, 4, 2, or 1.

```

200 DATA 1,192,0,1,224,0,1,240,0

```

Line 200 stores the **DATA** for the first 3 lines of THE RUNNING NOSE sprite. Simply add the decimal equivalents for each byte to determine the column position of each shaded box. For example,

```

Line 1, byte 1 = 1
Line 1, byte 2 = 128 + 64 = 192
Line 1, byte 3 = 0

Line 2, byte 1 = 1
Line 2, byte 2 = 128 + 64 + 32 = 224
Line 2, byte 3 = 0

Line 3, byte 1 = 1
Line 3, byte 2 = 128 + 64 + 32 + 16 = 240
Line 3, byte 3 = 0

```

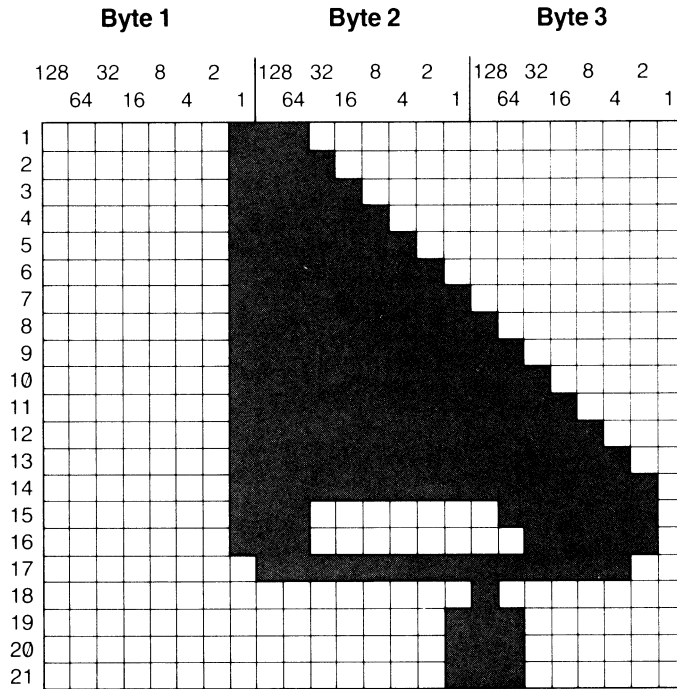


Figure 3.

Program lines 210, 220, 230, 240, 250, and 260 store the **DATA** for lines 3–6, 7–9, 10–12, 13–15, 16–18, and 19–21, respectively.

Now add the following program line:

```
105 POKE DC+29, 2: POKE DC+23, 2
```

and **RUN**. Gadzooks! A giant nose!

**In Plain English . . .**

**POKE DC + 29,2** **POKEs** the value of sprite #1 (2) into the address DC + 29, location of the X-axis expansion. **POKE DC + 23,2** **POKEs** the value of sprite #1 into the address DC + 23, the location of the Y-axis expansion. These addresses allow you to double the size of any sprite along one axis, or quadruple the size along both.

Now amend your program as follows:

```
50 S12=6
60 POKE S0, S12
```

```
70 POKE 2041,13:POKE 2042,13
105 POKE DC+29,6:POKE DC+23,6
```

Then add the following program lines:

```
135 POKE DC+4, HM
145 POKE DC+5, 150
```

and **RUN**. A pair of running noses!

### In Plain English . . .

S12 = 6 in line 50 now sets the variable S12 (for sprites 1 and 2) equal to 6—the combined values of sprite #1 and sprite #2.

Line 60 now **POKEs** the combined values of sprite #1 and sprite #2 into the address SO (for Sprite On).

Line 70 now **POKEs** storage block 13 into sprite pointers for both sprite #1 (2041) and sprite #2 (2042).

Line 105 now **POKEs** the combined values (6) of sprites 1 and 2 into the X,Y expansion address.

Line 135 **POKEs** the current value of HM into the X-axis register address for sprite #2.

Line 145 **POKEs** a value of 150 into the Y-axis register address for sprite #2.

Time for more changes:

```
140 POKE DC+3, HM
145 POKE DC+5, 255-HM
```

and **RUN**. Diagonally running noses! Lines 140 and 145 cause the Y-axis (vertical) movement of sprites 1 and 2.

Now add the following program line:

```
75 POKE DC+39+1, 1:POKE DC+39+2, 2
```

and **RUN**. One white nose, one red nose. **POKE DC + 39** is the address that stores the sprite color code.  $DC + 39 + 1$  is the color address for sprite #1 and  $DC + 39 + 2$  is the color address for sprite #2. Line 75 first **POKEs** the standard color code 1 (white) into the color address for sprite #1 ( $DC + 39 + 1$ ), then **POKEs** the standard color code 2 (red) into the color address for sprite #2 ( $DC + 39 + 2$ ).



**CHAPTER 15 REVIEW**

---

Bit	Sprite On ( $53248 + 21$ )
Byte	Sprite pointer
Display chip address (53248)	X, Y coordinates
Display grid ( $320 \times 220$ )	

---

**PRACTICE EXERCISES**

---

- 15-1. Amend the RUNNING NOSE program to change nose colors with each loop.
- 15-2. Amend the RUNNING NOSE program to display one expanded nose and one standard nose.
-

**DANCING IN THE STREETS****Sound**

Your Commodore would be hard pressed to reproduce Beethoven's Fifth Symphony, but it can generate a wide variety of sound and music. As an example, try the following program:

```

10 PRINT CHR$(147)
20 V=54296:REM VOLUME
30 AD=54277:REM ATTACK/DECAY
40 SR=54278:REM SUSTAIN/RELEASE
50 VW=54276:REM WAVEFORM
60 HF=54273:REM HIGH FREQUENCY
70 LF=54272:REM LOW FREQUENCY
100 POKE V,15
110 POKE AD,10
120 POKE SR,10
130 READ H:READ L
140 IF L=0 THEN END
150 POKE HF,H:POKE LF,L
160 POKE VW,17
170 FOR D=1 TO 250:NEXT:POKE VW,16
180 GOTO 130
200 DATA 8,97,9,104,10,143,11,48
210 DATA 12,143,14,24,15,210,16,195
220 DATA 0,0

```

Now turn up the volume on your TV and **RUN**. Your standard, everyday, C Major scale.

**In Plain English . . .**

```

10 PRINT CHR$(147)
20 V=54296:REM VOLUME

```

```

30 AD=54277:REM ATTACK/DECAY
40 SR=54278:REM SUSTAIN/RELEASE
50 VW=54276:REM WAVEFORM
60 HF=54273:REM HIGH FREQUENCY
70 LF=54272:REM LOW FREQUENCY

```

Line 20 sets variable V to 54296—the memory location for volume. The volume range is 1–15.

Line 30 sets variable AD to 54277—the memory location for voice #1 attack/decay. The AD rate determines how fast a note rises then falls (attacks then decays) from its peak volume level. The AD range is 0–255.

Line 40 sets SR to 54278—the memory location for voice #1 sustain/release. SR determines the rate to prolong a note at a specified level before releasing it (sustaining then releasing). The SR range is 0–255.

Line 50 sets variable VW to 54276—the memory location for voice #1 voice/waveform. Your Commodore 64 permits definition of up to 3 different voices, and each voice is assigned its own memory location (register). The following table lists the 3 different voice addresses and their corresponding attack/decay and sustain/release addresses:

	Memory Locations		
	<u>Voice</u>	<u>Attack/Decay</u>	<u>Sustain/Release</u>
Voice #1	54276	54277	54278
Voice #2	54283	54284	54285
Voice #3	54290	54291	54292

Each voice may be further defined and shaped by any of 4 different waveforms: triangle, sawtooth, pulse, and noise, and each waveform is assigned a pair of start/stop numbers:

	<u>Start</u>	<u>Stop</u>
Triangle	17	16
Sawtooth	33	32
Pulse	65	64
Noise	129	128

Line 60 sets variable HF to 54273—the memory location for high frequency; line 70 sets variable LF to 54272—the memory location for low frequency. Your Commodore 64 has a note range of 8 octaves, numbered 0 through 7, and each note requires both a high-frequency *and* a low-frequency value. For instance, the third octave C requires a high-frequency value of 8 and a low-frequency value of 97. A table of high/low frequency-note values is in Appendix 6.

```

100 POKE V, 15
110 POKE AD, 10
120 POKE SR, 10
130 READ H: READ L
140 IF L = 0 THEN END
150 POKE HF, H: POKE LF, L
160 POKE VW, 17
170 FOR D=1 TO 250: NEXT: POKE VW, 16
180 GOTO 130

```

Line 100 **POKEs** the maximum volume value 15 into the volume memory location as stored in variable V.

Line 110 **POKEs** the value 10 into the attack/decay memory location as stored in variable AD.

Line 120 **POKEs** the value 10 into the sustain/release memory location as stored in variable SR.

Line 130 **READs** the first **DATA** element and stores it in variable H (for high frequency), then **READs** the second **DATA** element and stores it in variable L (for low frequency).

Line 140 is a flag to determine the **END** of the program and to prevent an ?OUT OF DATA ERROR IN 130 prompt.

Line 150 **POKEs** the current **READ** value of variable H into the high-frequency memory location (HF), then **POKEs** the current **READ** value of variable L into the low-frequency memory location (LF).

Line 160 **POKEs** the value 17 (the start-note value for the triangle waveform) into the memory location of voice #1 (VW).

Line 170 sets the parameters of **FOR/NEXT** loop D (for duration) from 1 **TO** 250, loops itself, then **POKEs** the value 16 (the stop-note value for the triangle waveform) into the memory location of voice #1 (VW). Duration determines the length of each note. Sample duration equivalents are

```

1000 = whole note
500  = half note
250  = quarter note
125  = eighth note

```

Line 180 loops back to 130 for another note.

```

200 DATA 8, 97, 9, 104, 10, 143, 11, 48
210 DATA 12, 143, 14, 24, 15, 210, 16, 195
220 DATA 0, 0

```

Lines 200–210 store the high-frequency/low-frequency note values for the third octave C major scale.

Line 220 stores the flag **DATA** to tell the program when to **END** (line 140).

The following program lets you shape and define your own sounds:

```

NEW
10 PRINT CHR$(147)
20 INPUT "WAVEFORM = 17,33,65,OR 129";W
30 IF W<>17 AND W<>33 AND W<>65 AND W<>129 THEN 20
40 INPUT "ENTER ATTACK/DECAY (0-255)";AD
50 IF AD<0 OR AD>255 THEN 40
60 INPUT "ENR SUSTAIN/RELEASE (0-255)";SR
70 IF SR<0 OR SR>255 THEN 60
80 INPUT "ENTER DURATION (125-1000)";D
90 IF D<125 OR D>1000 THEN 80
100 POKE 54296,15:REM-SET VOLUME TO MAX
110 POKE 54277,AD:REM-SET ATTACK/DECAY
120 POKE 54278,SR:REM-SET SUSTAIN/RELEASE
130 READ H:READ L
140 IF L=0 THEN 1000
150 POKE 54273,H:REM-SET HIGH FREQUENCY
160 POKE 54272,L:REM-SET LOW FREQUENCY
170 POKE 54276,W:REM-VOICE 1 START NOTE
180 FOR X=1 TO D:NEXT:REM-SET DURATION
190 POKE 54276,W-1:REM-VOICE 1 STOP NOTE
200 GOTO 130
300 DATA 8,97,9,104,10,143,11,48
310 DATA 12,143,14,24,15,210,16,195
320 DATA 0,0
1000 PRINT:INPUT "LISTEN AGAIN (Y/N)";A$
1010 IF A$<>"Y" AND A$<>"N" THEN 1000
1020 IF A$="Y" THEN RESTORE:GOTO 130
1030 RESTORE:GOTO 20

```

and **RUN**. Enter the waveform, attack/decay, sustain/release, and duration as prompted, then sit back and listen to your creation. *Typing* “Y” in response to the LISTEN AGAIN (Y/N) prompt will repeat the same scale with the same values. *Typing* “N” will loop back to the start of the program to request new values. Try the following values:

Waveform	Attack/Delay	Sustain/Release	Duration
17	100	0	250
17	10	10	125
33	64	64	250
33	255	255	500
65	0	0	250
65	0	64	125
129	0	0	250
129	10	10	500

From Chopin to the OK Corral.

### In Plain English . . .

```

10 PRINT CHR$(147)
20 INPUT "WAVEFORM = 17, 33, 65, OR 129"; W
30 IF W<>17 AND W<>65 AND W<>129 THEN 20
40 INPUT "ENTER ATTACK/DECAY (0-255)"; AD
50 IF AD<0 OR AD>255 THEN 40
60 INPUT "ENTER SUSTAIN/RELEASE (0-255)"; SR
70 IF SR<0 OR SR>255 THEN 60
80 INPUT "ENTER DURATION (125-1000)"; D
90 IF D<125 OR D>1000 THEN 80

```

Line 20 requests you to **INPUT** the waveform and stores your **INPUT** in variable W. Line 30 tests **INPUT** W and loops back to line 20 only **IF** W does not equal 17 **AND** does not equal 33 **AND** does not equal 65 **AND** does not equal 129.

Line 40 requests you to enter the attack/decay rate, storing your **INPUT** in variable AD. Line 50 tests **INPUT** AD and will loop back to line 40 only **IF** AD is less than 0 **OR** greater than 255.

Line 60 requests you to enter the sustain/release value, storing your **INPUT** in variable SR. Line 70 tests **INPUT** SR and will loop back to line 60 only **IF** SR is less than 0 **OR** greater than 255.

Line 80 requests you to enter the duration, storing your **INPUT** in variable D. Line 90 tests **INPUT** D and will loop back to line 80 only **IF** D is less than 125 **OR** greater than 1000.

```

100 POKE 54296, 15: REM-SET VOLUME TO MAX
110 POKE 54277, AD: REM-SET ATTACK/DECAY
120 POKE 54278, SR: REM-SET SUSTAIN/RELEASE
130 READ H: READ L
140 IF L=0 THEN 1000

```

```

150 POKE 54273,H:REM-SET HIGH FREQUENCY
160 POKE 54272,L:REM:-SET LOW FREQUENCY
170 POKE 54276,W:REM-VOICE 1 START NOTE
180 FOR X=1 TO D:NEXT:REM-SET DURATION
190 POKE 54276,W-1:REM-VOICE 1 STOP NOTE
200 GOTO 130

```

Line 100 **POKEs** the value 15 into the memory location 54296, setting the volume to maximum.

Line 110 **POKEs** the current value of AD into memory location 54277, setting the attack/decay rate.

Line 120 **POKEs** the current value of SR into memory location 54278, setting the sustain/release rate.

Line 130 **READs** (stores) the first **DATA** element in variable H, then **READs** and stores the second **DATA** element in variable L.

Line 140 is a flag, telling the program to branch (**GOTO**) program line 1000 only **IF** L equals 0.

Line 150 **POKEs** the current value of H into memory location 54273, setting the high-frequency value. Line 160 **POKEs** the current value of L into memory location 54272, setting the low-frequency value.

Line 170 **POKEs** the current waveform value of variable W into memory location 54276, starting the note in voice #1.

**FOR/NEXT** loop X in line 180 sets the duration (1 **TO** D) as determined by the current value of variable D.

Line 190 **POKEs** the value of W-1 into memory location 54276, stopping the note in voice #1.

Line 200 loops back to line 130 for another note.

```

300 DATA 8,97,9,104,10,143,11,48
310 DATA 12,143,14,24,15,210,16,195
320 DATA 0,0

```

Lines 300 and 310 store the high/low frequency **DATA**.

Line 320 stores the flag **DATA** as required by line 140.

```

1000 PRINT: INPUT "LISTEN AGAIN (Y/N) ";A$
1010 IF A$<>"Y" AND A$<>"N" THEN 1000
1020 IF A$="Y" THEN RESTORE: GOTO 130
1030 RESTORE: GOTO 20

```

Line 1000 **PRINTs** a blank line, then requests you to **INPUT** "Y" for Yes or "N" for No, storing your **INPUT** in string variable A\$.

Line 1010 tests **INPUT A\$** and will loop back to line 1000 only **IF A\$** is not equal to "Y", **AND** is not equal to "N".

Line 1020 will **RESTORE** the **DATA** pointer, then loop back to program line 130 only **IF A\$** equals "Y".

Line 1030 will restore the **DATA** pointer, then loop back to program line 20 only **IF** line 1020 tests false.

## CHAPTER 16 REVIEW

---

Attack/Decay  
Duration  
Sustain/Release

Voice  
Volume  
Waveform

---



---

# Math and Trigonometric Functions

---

## Math Functions

Your Commodore 64 makes short work of long numbers. Consider the following program:

```
10 PRINT CHR$(147)
20 FOR X=1 TO 10
30 PRINT "SQUARE ROOT OF"; X; "="; SQR(X)
40 NEXT
```

and **RUN**. Your display will appear as follows:

```
SQUARE ROOT OF 1 = 1
SQUARE ROOT OF 2 = 1.41421356
SQUARE ROOT OF 3 = 1.73205081
SQUARE ROOT OF 4 = 2
SQUARE ROOT OF 5 = 2.23606798
SQUARE ROOT OF 6 = 2.44948974
SQUARE ROOT OF 7 = 2.64575131
SQUARE ROOT OF 8 = 2.82842713
SQUARE ROOT OF 9 = 3
SQUARE ROOT OF 10 = 3.16227766
```

**In Plain English . . .**

Line 20 sets the parameters of **FOR/NEXT** loop X from 1 to 10.

Line 30 **PRINTs** SQUARE ROOT OF (the current value of X) equal to **SQR** (the current value of X). **SQR(X)** is the function used to calculate the square root of X.

Line 40 recalls **FOR/NEXT** loop X and increases the value of X by 1.

Now try this program:

**NEW**

```
10 PRINT CHR$(147)
20 PRINT:PRINT:PRINT TAB(9); "NO. "; TAB(19); "SGN";
  TAB(29); "ABS":PRINT
30 FOR X=-5 TO 5
40 PRINT TAB(9); X; TAB(19); SGN(X); TAB(29); ABS(X)
50 NEXT
```

and **RUN**. Your display should appear as follows:

NO.	SGN	ABS
-5	-1	5
-4	-1	4
-3	-1	3
-2	-1	2
-1	-1	1
0	0	0
1	1	1
2	1	2
3	1	3
4	1	4
5	1	5

**In Plain English . . .**

Line 20 **PRINTs** two blank lines, then **PRINTs** NO., SGN, and ABS at **TAB** positions 9, 19, and 29, respectively; and then one final blank line.

Line 30 sets the parameters of **FOR/NEXT** loop X from -5 to 5.

Line 40 **PRINTs** the current value of X at **TAB** position 9, the *signum* of X—**SGN(X)**—at **TAB** position 19, and the **ABS**olute value of X at **TAB** position 29. **SGN(X)** will always return -1 if the value of X is negative (-5 through -1 in the table), 0 if X equals 0, and 1 if the value of X is positive (1 through 5 in the table). **ABS(X)** will

always return the *absolute value* of X, whether the value of X is negative, zero, or positive.

Line 50 recalls **FOR/NEXT** loop X and increases the value of X by 1.

Here's another new program that uses several math functions:

```

NEW
10 PRINT CHR$(147)
20 PRINT: PRINT: PRINT "NO. "; TAB(13); "LOG"; TAB(25); "EXP";
   TAB(35); "NO. "; PRINT
30 FOR X=1 TO 10
40 PRINT X; TAB(8); LOG(X); TAB(20); EXP(X); TAB(35); EXP(LOG(X))
50 NEXT

```

and **RUN**. Your display will appear as follows:

NO.	LOG	EXP	NO.
1	0	2.71828183	1
2	.693147181	7.3890561	2
3	1.09861229	20.0855369	3
4	1.38629436	54.5981501	4
5	1.60943791	148.413159	5
6	1.79175947	403.428793	6
7	1.94591015	1096.63316	7
8	2.07944154	2980.95799	8
9	2.19722458	8103.08393	9
10	2.30258509	22026.4658	10

### In Plain English . . .

Line 20 **PRINTs** two blank lines, then **PRINTs** NO., LOG, EXP, and NO., followed by a final blank line.

Line 30 sets the parameters of **FOR/NEXT** loop X from 1 to 10.

Line 40 **PRINTs** the current value of X, the *natural logarithm* of X at **TAB** position 8, the *exponential* value of X at **TAB** position 20, and the exponential value of **LOG(X)** at **TAB** position 35. **LOG(X)** returns the natural logarithm of the value X. The formula to convert a natural logarithm to its *common logarithm* is **LOG(X)/LOG(10)**. **EXP(X)** returns the exponential value of X (**EXP(X) = e<sup>X</sup>**, where *e* is defined as 2.718281828).

**EXP(LOG(X))** illustrates the relationship of **LOG** and **EXP** as *inverse functions*. The exponential value of the logarithm of X is also equal to the logarithm of the exponential value of X. For example, **EXP(LOG(5)) = 5** and **LOG(EXP(5)) = 5**.

Line 50 recalls **FOR/NEXT** loop X and increases the value of X by 1.

---

## Function Keys

Your Commodore 64 has four programmable *function keys* on the far-right side of the keyboard. Each function key allows you to recall up to two preprogrammed subroutines—one for each non-SHIFTed key (f1, f3, f5, and f7), and one for each SHIFTed key (f2, f4, f6, and f8)—at any predetermined point in your program, as the following program illustrates:

```

NEW
10 PRINT CHR$(147)
20 PRINT:PRINT: INPUT "ENTER ANY NUMBER";N
30 PRINT:PRINT "PRESS ANY FUNCTION KEY, ":PRINT "SHIFTED OR
   NONSHIFTED": PRINT
40 GET A$: IF A$="" THEN 40
50 IF A$=CHR$(133) THEN PRINT N;"*";1;"=";N*1
60 IF A$=CHR$(137) THEN PRINT N;"*";2;"=";N*2
70 IF A$=CHR$(134) THEN PRINT N;"*";3;"=";N*3
80 IF A$=CHR$(138) THEN PRINT N;"*";4;"=";N*4
90 IF A$=CHR$(135) THEN PRINT N;"*";5;"=";N*5
100 IF A$=CHR$(139) THEN PRINT N;"*";6;"=";N*6
110 IF A$=CHR$(136) THEN PRINT N;"*";7;"=";N*7
120 IF A$=CHR$(140) THEN PRINT N;"*";8;"=";N*8
130 GOTO 30

```

and **RUN**. First enter any number as requested. Then press any function key. Each function key 1–8 produces a slightly different equation. Function key 1 produces  $N * 1 =$  (the value of  $N*1$ ), function key 2 produces  $N * 2 =$  (the value of  $N*2$ ), function key 3 produces  $N * 3 =$  (the value of  $N*3$ ), and so on.

### In Plain English . . .

```

10 PRINT CHR$(147)
20 PRINT:PRINT: INPUT "ENTER ANY NUMBER";N
30 PRINT:PRINT "PRESS ANY FUNCTION KEY, ":PRINT "SHIFTED OR
   NONSHIFTED": PRINT
40 GET A$: IF A$="" THEN 40

```

Line 20 **PRINT**s two blank lines, then requests you to enter any number, storing your **INPUT** in variable N.

Line 30 **PRINTs** one blank line, then **PRESS ANY FUNCTION KEY**, followed by **SHIFTED OR NONSHIFTED** and another blank line.

**GET A\$** in line 40 scans the keyboard and loops itself until you **PRESS ANY FUNCTION KEY** as prompted in line 30.

```

50 IF A$=CHR$(133) THEN PRINT N; "*"; 1; "="; N*1
60 IF A$=CHR$(137) THEN PRINT N; "*"; 2; "="; N*2
70 IF A$=CHR$(134) THEN PRINT N; "*"; 3; "="; N*3
80 IF A$=CHR$(138) THEN PRINT N; "*"; 4; "="; N*4
90 IF A$=CHR$(135) THEN PRINT N; "*"; 5; "="; N*5
100 IF A$=CHR$(139) THEN PRINT N; "*"; 6; "="; N*6
110 IF A$=CHR$(136) THEN PRINT N; "*"; 7; "="; N*7
120 IF A$=CHR$(140) THEN PRINT N; "*"; 8; "="; N*8
130 GOTO 30

```

Lines 50–120 test A\$ to determine which function key was pressed and then respond with the appropriate formula.

Line 130 loops back to line 30 for another function key.

You can program the function keys for just about any numeric or string function. For example,

**NEW**

```

10 PRINT CHR$(147)
20 GET A$: IF A$="" THEN 20
30 IF A$=CHR$(133) THEN PRINT CHR$(147):LIST
40 PRINT A$;
50 GOTO 20

```

and **RUN**. This program creates an elementary word processor. *Press* the f1 key at any time to clear the screen and **LIST** your program.

### **In Plain English . . .**

**GET A\$** in line 20 scans the keyboard and loops itself until you press a key.

Line 30 tests string variable A\$ and will clear the screen, then **LIST** your program only **IF** string variable A\$ equals the **CHR\$** Code for the f1 key (133).

Line 40 **PRINTs** the value of string variable A\$.

Line 50 loops back to line 20 for another letter.

---

## The DEF FN Statement

The **DEF FN** statement lets you define any function and recall it at any time. Here's an example:

```

NEW
10 PRINT CHR$(147)
20 INPUT "ENTER ANY NUMBER"; N
30 DEF FN A(N) = N - N
40 PRINT: PRINT "Nsp="; N
50 PRINT: PRINT "FNA ="; FN A(N)
60 DEF FN B(N) = N - (N - 1)
70 PRINT: PRINT "FNB ="; FN B(N)

```

and **RUN**. No matter what number you **INPUT**, **FN A(N)** in line 50 will always **PRINT** 0, while **FN B(N)** in line 70 will always **PRINT** 1.

### In Plain English . . .

Line 20 requests you to enter any number, storing your **INPUT** in variable N.

**DEF FN A(N)** in line 30 **DEF**ines function A (**FN A**), as it applies to variable N equal to  $N - N$ .

Line 40 **PRINT**s one blank line followed by  $N =$  (the current value of N).

Line 50 **PRINT**s one blank line followed by **FNA =** and the defined function value of A as it applies to variable N—**FN A(N)**.

Line 60 **DEF**ines function B (**FN B**) as it applies to variable N equal to  $N - (N - 1)$ .

Line 70 **PRINT**s one blank line followed by **FNB =** and the defined function value of B as it applies to variable N—**FN B(N)**.

---

## Trigonometric Functions

I rarely write about something I can't spell—and trigonometry is one such thing. But what the hell?

Nearly all trigonometric problems deal with angles, and most angles are defined in degrees. Your Commodore 64, however, steadfastly refuses to deal in degrees and

instead calculates all trigonometric functions in radians. Therefore, all degrees must first be converted to radians by the formula  $D * \pi/180 = \text{radians}$ , where D equals the value of an angle in degrees, and the value of  $\pi$  (PI) is defined as the circumference of a circle divided by its diameter, and is equal to 3.141 592 653. The following program illustrates this conversion and the use of four trigonometric functions:

**NEW**

```

10 PRINT CHR$(147)
20 PRINT: INPUT "ENTER ANY ANGLE IN DEGREES"; D
30 R=D*3.141592653/180
40 PRINT: PRINT D; "DEGREES"; R; "RADIANS"
50 PRINT "SIN"; SIN(R)
60 PRINT "COS"; COS(R)
70 PRINT "TAN"; TAN(R)
80 PRINT "ATN"; ATN(R)
90 GOTO 20

```

and **RUN**.

### In Plain English . . .

Line 20 **PRINTs** one blank line, then requests you to enter any angle in degrees, storing your **INPUT** in variable D.

Line 30 converts the degree value of D into radians.

Line 40 **PRINTs** one blank line, the current value of variable D, **DEGREES =**, then the current value of variable R, **RADIANS**.

**SIN(R)** in line 50 calculates then **PRINTs** the *sine* of angle D in radians. **SIN(R)** is the ratio of the length of the opposite side to the hypotenuse in a right-angle triangle with an angle of R radians (see Figure 4).

**COS(R)** in line 60 calculates then **PRINTs** the *cosine* of angle D in radians. **COS(R)** is the ratio of the length of the adjacent side to the hypotenuse in a right-angle triangle with an angle of R radians.

**TAN(R)** in line 70 calculates then **PRINTs** the *tangent* of angle D in radians. **TAN(R)** is the ratio of the length of the opposite side to the adjacent side in a right-angle triangle with an angle of R radians.

**ATN(R)** in line 80 calculates then **PRINTs** the *arctangent* of angle D in radians. **ATN(R)** is the angle whose tangent is R. Note that **TAN(R)** and **ATN(R)** are inverse functions: **TAN(ATN(R)) = ATN(TAN(R)) = R**.

Line 90 loops back to line 20 to request further **INPUT**.

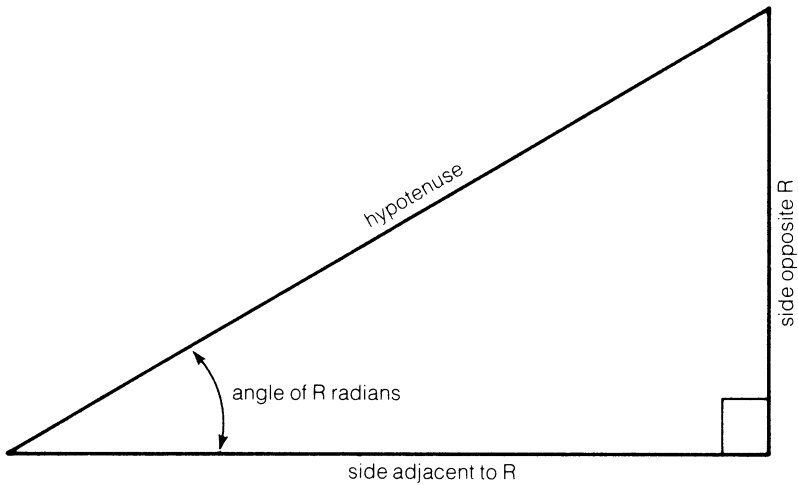


Figure 4. Right triangle nomenclature.

## CHAPTER 17 REVIEW

---

ABSolute value	Natural LOGarithm
ATN (arctangent)	PI ( $\pi$ )
Common logarithm	Radians
COSine	SGN (signum)
DEFine	SINe
EXPonential value	SQR (square root)
FN (function)	TANgent
Function keys	

---

## PRACTICE EXERCISES

---

- 17-1. Write a program that requests you to **INPUT** the radius of a circle, then calculates and **PRINTs** the area of a circle via the **DEF FN** function.
- 17-2. Write a program that requests you to **INPUT** the **TAN**gent of an angle, then calculates and **PRINTs** the value of the corresponding angle in degrees.
-



# Memory, Report and Error Codes, and Applications

## Memory

Your Commodore's memory is made up of individual storage cells called *bytes*. Each byte is divided into 8 separate compartments called *bits*, and each bit contains either a 0 or a 1 in *binary* (base 2) representation. The following table illustrates how the decimal number 54 is converted into its 8-bit (1 byte) binary form:

<b>ROW A</b>	8	7	6	5	4	3	2	1
<b>ROW B</b>	0	0	1	1	0	1	1	0
<b>ROW C</b>	128	64	32	16	8	4	2	1

ROW A labels each bit, 1–8, from right to left.

ROW B represents the binary equivalent of the decimal number 54 (00110110 = 54).

ROW C lists the decimal equivalent of each bit, 1–8.

Now total the decimal equivalents (from ROW C) of the bits in ROW B noted with 1's (bits 2, 3, 5 and 6):

Bit #		Decimal Equivalent
Bit 2	=	2
Bit 3	=	4
Bit 5	=	16
Bit 6	=	32
Total	=	54

The smallest possible number in any 8-bit byte is 0 (00000000), while the largest possible number in any 8-bit byte is 255 (11111111), as illustrated below:

<b>ROW A</b>	8	7	6	5	4	3	2	1
<b>ROW B</b>	1	1	1	1	1	1	1	1
<b>ROW C</b>	128	64	32	16	8	4	2	1

Bit #	Decimal Equivalent
Bit 1	= 1
Bit 2	= 2
Bit 3	= 4
Bit 4	= 8
Bit 5	= 16
Bit 6	= 32
Bit 7	= 64
Bit 8	= 128
Total	= <u>255</u>

Consequently, every byte of memory stores a number between 0 and 255 (remember **ASC**, sprites, and **CHR\$**), and each byte has a specific location, or *address*.

Your Commodore 64's memory is divided into two segments:

1. ROM (Read Only Memory) contains your Commodore's standard operating instructions. Computers understand only binary and therefore must first translate any high-level language (BASIC, COBOL, FORTRAN, PASCAL, etc.) into binary before execution. The ROM permanently stores all the instructions required for this BASIC-to-binary conversion. You cannot alter any of the bytes in ROM, but you can **PEEK** inside any ROM byte to return the stored value at any time.
2. RAM (Random Access Memory, also called *main memory*), consists of 64K bytes (1K = 2<sup>10</sup> = 1024 bytes). RAM provides the storage space for all your programs and program variables. Storage in RAM is not permanent as it is in ROM, and, as the name implies, any RAM byte can be accessed or altered at will via the **POKE** command.

---

## Report and Error Codes

A program *crashes* when it fails to execute as a result of a program error or the entry of invalid data. Program errors are referred to as *bugs*, and programs with bugs must be *debugged* (the computer equivalent of hiring an exterminator).

*Error messages* as they appear on the screen are useful tools for debugging programs. Most error messages relate not only to the nature of the problem, but the number of the offending program line as well. A complete list of error messages appears in Appendix 4.

---

## Applications

The potential applications for your Commodore are limited only by your own imagination and budget.

### Hardware and Software

Thousands of different *hardware* (the actual machinery and circuitry) and *software* (the program to drive the hardware) configurations are possible to suit virtually any application.

### Peripherals

*Peripherals* are satellite hardware that perform specific functions as instructed by your computer. Typical peripherals include

1. **Monitor:** your television or a similar screen designed specifically for use with computers.
2. **Printer:** a device to **PRINT** hardcopy—an indispensable device for either word processing or **PRINTing** graphics. Printers, however, are often expensive. Make sure the printer you want is compatible with your specific output requirements (text or graphics) *and* your Commodore computer before you buy.
3. **Cassette Tape Recorder:** a magnetic software storage device. Although much less expensive, a cassette tape recorder is infinitely slower and more cumbersome than a disk drive.
4. **Disk Drive:** a very quick, facile device for program storage wherein programs are recorded on or retrieved from inexpensive, magnetic (floppy) disks. Your Commodore's Disk Operating System (DOS) is explained in Appendix 1.
5. **Modem:** a device that connects your computer to vast telecommunications networks (anything from Dow Jones stock reports to shop-at-home services to interaction with other remote personal computers).

6. Game Paddles: paddles that plug into control ports 1 and 2 on the right side of your Commodore 64 and can be used in game and graphics applications.

## Languages

Computer languages bridge the gap between you and your Commodore. Your Commodore can access and employ several different *high-level* computer languages (requiring internal translation), each designed for specific applications:

*Parlez-vous BASIC? Oui!* Your Commodore comes equipped with a resident dialect of the BASIC (Beginner's All-purpose Symbolic Instruction Code) computer language. In fact, every program in this book was written in Commodore BASIC.

PASCAL is a language designed primarily as an educational tool to teach structured programming and problem solving. FORTRAN is a mathematically powerful and adroit computer language, suitable for many science and business applications. FORTH is a very fast high-level language (faster in fact than virtually any other high-level language) that allows users to define their own commands and instructions. COBOL is a language designed specifically for business-oriented applications that require both strong math and text capabilities. LOGO presents programming concepts in a simple, natural manner ideal for small children. PILOT was designed as an educational aid, employing color graphics, sound, and text to enhance virtually any curriculum.

6502 Assembly Language is your Commodore's native tongue, a *low-level* language that requires none of the internal translation of high-level languages, and therefore executes at a speed far greater than any high-level language. Although considerably more difficult to learn and employ than high-level languages, assembly language nevertheless is much more flexible and affords much greater control over your Commodore's entire system.

---

## Final Thoughts

Never assume (with computers, or anything else) that what you want is impossible to attain. Don't be afraid to reach out for help; it's out there. Dozens of Commodore user clubs exist throughout the country, and most people interested in computers are more than willing to share their knowledge and experience. Just ask. Also, an extensive library of books (for general or specific applications) already exists for all ages and levels of proficiency. We've only scratched the surface here, so when you're comfortable with the techniques we've presented, branch out and have fun.

## APPENDIX 1

---

# Saving, Loading, and Printing

No need to retype programs each time you want to use them. Simply **SAVE** them to tape or disk, then **LOAD** them back into the computer whenever you want to.

---

## Tape Storage

Your Commodore 64 requires a special Commodore Datassette cassette recorder to **SAVE** and **LOAD** programs to or from tape. Consult your *Commodore 64 User's Guide* for proper hook-up procedures.

### Saving to Tape

Ready? *Type* the following program:

```
1Ø REM ME
2Ø PRINT CHR$(147)
3Ø PRINT "YOU FOUND ME AND I AM LOADED"
```

Now *type* **SAVE** "ME" in Immediate Mode and *press* **RETURN**. The prompt

```
PRESS RECORD AND PLAY ON TAPE
```

will appear. Go ahead, press record and play on your Datassette recorder. Your screen goes blank for several seconds, then returns with the prompt

```
SAVING ME
READY.
```

Press **STOP** on your Datassette. You have now **SAVED** "ME" to tape. The **SAVE** procedure follows the same two steps each time:

1. *Type* **SAVE** "PROGRAM NAME" and *press* **RETURN**. (The program name between quotes can be up to 16 characters.)
2. *Press* record and play on tape as prompted.

## Verifying

How do you know that the **SAVE** process was successful? Simply follow these steps:

1. Rewind the tape to the beginning of the program you wish to **VERIFY**. Then *type* **VERIFY** "ME" (or whatever the program name might be) in Immediate Mode and *press* **RETURN**. The screen will go blank and return a few seconds later with the prompt

```
SEARCHING FOR ME  
FOUND ME
```

2. *Press* the **COMMODORE KEY** (lower-left corner of the keyboard). Your display will go blank again for a few seconds, then return with the prompt

```
VERIFYING  
OK
```

This prompt indicates that the program now **SAVED** on tape precisely matches the program in memory. A ?**VERIFYING ERROR** prompt will appear if the program on tape and the program in memory do not match, your cue to initiate the **SAVE** process again.

## Loading from Tape

No problem! First *type* **NEW** and *press* **RETURN** to clear your Commodore's memory. Now rewind the cassette to the beginning of the program and follow the steps below to **LOAD** "ME" from Datassette tape back into your computer.

1. *Type* **LOAD** "ME" in Immediate Mode and *press* **RETURN**. The following prompt appears:

```
PRESS PLAY ON TAPE
```

2. That's exactly what you do—press the play button on the tape recorder. The screen will go blank for several seconds as before and return with the prompts

```
SEARCHING FOR ME  
FOUND ME
```

3. Now *press* the **COMMODORE KEY**. The display will go blank again and return once more with the prompts

```
LOADING  
READY.
```

4. **RUN** your program.

```
YOU FOUND ME AND I AM LOADED!
```

### Loading Prepackaged Programs from Cassette

A whole world of prepackaged software exists on tape cassette for your Commodore 64. Simply make sure that the tape cassette is completely rewound to the beginning, *type* **LOAD** and *press* **RETURN**. No need to enclose the program name in quotes this time. Unless specified otherwise, your Commodore will always **LOAD** the first program it encounters on the cassette.

### Loading Cartridge Programs

Three steps are involved:

1. Turn off your Commodore 64. Never insert or remove cartridges with your computer on.
2. Insert the cartridge into the cartridge port.
3. Turn your Commodore back on.

### Saving and Loading Hints

Learn to use the tape counter on the Datassette. Knowing where your programs start on tape will save you a lot of time and frustration.

**LOAD**ing a program from tape automatically erases any program previously in memory. So be sure to **SAVE** any resident programs you want to tape *before* you **LOAD**.

You may interrupt the **SAVE/LOAD** process at any time by pressing **RUN/STOP**.

The longer a program, the longer it will take to **SAVE** and **LOAD**.

---

## Disk Drive

Disk drives store and retrieve information at a much greater speed than cassette tapes. Moreover, your Datassette recorder must read files consecutively via a process called *sequential access*, whereas a disk drive can read any required program from anywhere on the disk via *random access*.

The Commodore 1541 Disk Drive is recommended for use with your Commodore 64. Consult either the manual that comes with the drive or your *Commodore 64 User's Guide* to determine the proper hook-up procedure. A couple of important points to remember *before* you proceed:

1. Always turn on your disk drive *before* you turn on the computer. In fact, always turn your computer on last, no matter how many other peripherals are connected.
2. Never turn your disk drive's power on or off while a disk is in it. Doing so might erase all of the programs on your disk.

## Floppies

Sounds a little like a Walt Disney feature. Your Commodore 1541 Disk Drive is designed to use *single-sided 5¼" floppy* diskettes (you can store information on one side only). A few tips on handling your diskettes:

1. Never touch any exposed surface of the interior vinyl disk.
2. Never write on a disk label while the label is affixed to the disk.
3. Don't bend your disk (check first with your chiropractor).
4. Don't expose your disk to magnetic devices.
5. Don't expose your disk to extreme heat or cold.

In other words, let's be careful out there.

## DOS

The Disk Operating System (DOS) simplifies disk drive operation via a preprogrammed set of DOS commands. Each floppy disk has a total storage capacity of 174,848 bytes divided into 683 *sectors* of 256 bytes each ( $683 \times 256 = 174,848$ ). DOS enables you to perform several different functions on any sector at random.



## Formatting

All blank disks must be formatted before any other DOS function can be applied (saving, loading, etc.). Formatting erases all previous data from the disk, then formats it by dividing the read/write surface into 35 *tracks* of 17–21 sectors each. Here's the format procedure:

1. First, insert a blank disk, label side up, into the drive and close the door.
2. *Type*

```
OPEN 15, 8, 15 RETURN
PRINT#15, "NEWØ: 1ST DISK, Ø1" RETURN
```

The red in-use lamp on your disk drive will light while the drive whirs and buzzes for a few seconds. Your disk has been formatted and is ready for use the moment the red lamp goes out and the word **READY** appears on your screen. You never have to format the same disk twice. Once is enough.

The **OPEN** and **PRINT#** commands above are explained in greater detail later in this appendix.

## Saving to Disk

First, *type*

```
1Ø REM ME
2Ø PRINT CHR$(147)
3Ø PRINT "YOU FOUND ME AND I AM LOADED"
```

Now simply *type* the following to **SAVE ME** to disk:

```
SAVE "ME", 8
```

and *press* **RETURN**. The prompt **SAVING ME** appears for a brief moment while the drive whirs and buzzes and stops finally with the **READY** prompt on the display. Saving to disk always follows the same format:

```
SAVE "PROGRAM NAME", 8
```

The **SAVE** process for disk is exactly the same as for tape, except for the addition of the comma and the device number 8 after the program name in quotes. If you omit the comma and the 8, your Commodore defaults to the Datassette with the prompt **PRESS RECORD AND PLAY ON TAPE**.

## Verifying

Again, this is the same procedure you followed for tape, with slight modifications. To VERIFY "ME", type

```
VERIFY "ME", 8
```

and *press* **RETURN**. The following prompts will appear while your drive does its thing:

```
SEARCHING FOR ME
VERIFYING
OK
```

OK means that the program stored on disk precisely matches the program currently in your Commodore's memory. The prompt ?VERIFY ERROR indicates that the two programs do not match, your cue to repeat the **SAVE** procedure.

## Loading to Disk

First *type* **NEW** and *press* **RETURN** to erase the above program ME from memory. To **LOAD ME** from disk back into the computer's memory, *type*

```
LOAD "ME", 8
```

and *press* **RETURN**. The following prompts appear:

```
SEARCHING FOR ME
LOADING
READY.
```

Now *type* **RUN** and *press* **RETURN**. YOU FOUND ME (again) AND I AM LOADED.

**LOADing** prepackaged software follows the same simple procedure:

```
LOAD "PROGRAM NAME", 8
```

## Disk Directory

Your disk drive can provide a current list of all programs on any given disk. Just *type*

```
LOAD "$", 8
```

and *press* **RETURN**. When the READY prompt appears, *type* **LIST** and *press* **RETURN**. The following display will appear:

```

0 "1STDISK" " 01 2A
1 "ME" PRG
663 BLOCKS FREE.
READY.

```

1ST DISK and 01 at the top refer, of course, to the disk name and code number you assigned earlier. The number 1 on the next line indicates the number of disk storage blocks occupied by the program "ME." 663 BLOCKS FREE means just that: Of the original 683 blocks, 1STDISK has 663 BLOCKS FREE for additional files.

### Saving and Replacing a Program

What happens when you want to **LOAD** a program file, amend it, then re-**SAVE** the program file to disk? First **LOAD** the program ME back into the computer with **LOAD** "ME",8. Then add program line 40 as follows:

```
40 LIST
```

Now ME will **LIST** automatically each time you **RUN** it. (Grammar like this could drastically shorten my career.) To **SAVE** the amended version of ME to disk, simply *type*

```
SAVE "@0:ME",8
```

and **RETURN**. Now *type* **VERIFY** "ME",8 and *press* **RETURN** to insure that the amended program file on disk corresponds to the program in memory. In short, to replace an old program file with an amended version of the same file, simply prefix the filename with the characters @0:.

---

## DOS Commands

This section provides a list of the various DOS commands and a brief explanation of each.

### OPEN

1. All DOS commands require that a channel to the disk drive first be **OPENed**.
2. The **OPEN** command opens any given file for subsequent functions or alterations.

3. The **OPEN** command is also used to open a channel between the computer and the disk drive and follows this format:

```
OPEN file #, device #, channel #
```

The file number can be from 1–255. The device number for your disk drive is always 8. Channel number 15 is reserved for DOS commands only. Channel numbers 2 through 14 are available to **READ** or **WRITE** data to disk files. The command

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

opens the DOS *command channel* for subsequent DOS commands.

## CLOSE

1. Always **CLOSE** files when finished via the command

```
CLOSE file #
```

2. The **CLOSE** file number should always correspond to the preceding **OPEN** file number.
3. Command channel 15 should always be the first channel **OPENed** and the last channel **CLOSEd**.

## PRINT#

1. The **PRINT#** command **PRINTs** data to a previously **OPENed** disk file.
2. **PRINT#** is also used to transmit DOS commands to the disk drive:

```
OPEN 15, 8, 15  
PRINT#15, "NEW0: 1STDISK, 01"
```

Use these commands to transmit the **FORMAT** command (NEW0) to the disk drive once DOS command channel 15 has been **OPENed**.

## NEW

1. As demonstrated earlier, the **NEW** command is used to **FORMAT** a disk and assign its ID code via the format

```
PRINT#15, "NEWØ: DISKNAME, ID CODE"
```

2. The abbreviation NØ may be used instead of NEWØ.

## COPY

1. **COPY** is used to **SAVE** a separate copy of a program or data file to the same disk as the original by assigning the file a different name.
2. **LOAD** the program ME from disk into memory, then *type*

```
PRINT#15, "COPYØ: YOU=COPYØ: ME"
```

and *press RETURN*. You have just copied ME under the new filename YOU. Now both ME and YOU, identical except for the filenames, reside on the same disk.

3. To **COPY** any program or data file,

```
PRINT#15, "COPYØ: NEW FILENAME=COPYØ: OLD FILE NAME"
```

## RENAME

**RENAME** allows you to change an existing filename. For example,

```
PRINT#15, "RØ: I=ME", 8
```

will change the file named "ME" to a file named "I". Now YOU and I reside together on disk instead of YOU and ME. To **RENAME** any disk file *type*

```
PRINT#15, "RØ: NEW FILENAME=OLD FILENAME"
```

## SCRATCH

The **SCRATCH** command erases unwanted files from the disk:

```
PRINT#15, "SØ: ME"
```

erases the file ME from disk. To erase any file from disk, *type*

```
PRINT#15, "SØ: FILE NAME"
```

## INITIALIZE

The red lamp on your Commodore 1541 Disk Drive will flash whenever a DOS or a disk error is encountered. The command

```
PRINT#15, " INITIALIZE"
```

resets the disk drive to its INITIAL condition and turns off the flashing red error lamp.

---

## Printing

First **OPEN** a channel to your printer as follows:

```
OPEN 5, 4 RETURN
```

The 5 in the command refers to the file number (any number between 1 and 255). The 4 refers to the device number (4 or 5 for your printer).

Next you must detour output to the printer via the command

```
CMD5 RETURN
```

Your printer will whirl and print **READY** exactly as it would normally appear on the screen. **CMD** instructs your Commodore to send output to the printer instead of the display. Every command following **CMD** will go to the printer. The file (not device) number always follows **CMD**.

Now *type* the following in Immediate Mode:

```
FOR X=1 TO 5:PRINT:NEXT RETURN
```

Your printer advanced the paper 5 lines, exactly as the above **FOR/NEXT** loop would have **PRINTed** five blank lines on the display. Whereas **CMD** sends everything after it to the printer, **PRINT#** allows you to alter output at will between the display and printer, as the following program demonstrates:

```
10 PRINT CHR$(147)
20 OPEN 5, 4
30 INPUT "ENTER YOUR NAME"; N$
40 PRINT#5, N$
50 PRINT "PRESS ANY KEY TO LIST THIS PROGRAM"
60 GET A$: IF A$="" THEN 60
70 CMD5
```

```
80 LIST
90 PRINT#5:CLOSE5
```

and **RUN**. Your name is **PRINTed** in hard copy followed by the entire program **LISTing**.

### **In Plain English . . .**

Line 20 **OPENS** file number 5 to device 4.

Line 30 requests your **INPUT** on the display, storing it in string variable N\$.

**PRINT#5,N\$** in line 40 sends the current value of N\$ to the printer. **PRINT#** is always followed by the file (not device) number.

**CMD5** in line 70 sends all subsequent output to the printer.

Line 80 **LISTs** the entire program on the printer.

**PRINT#5:CLOSE5** in line 90 **CLOSEs** file number 5. **CLOSE(file number) CLOSEs** the channel between your computer and printer.

**note:** **PRINT#(file number)** must always precede **CLOSE(file number)** as in line 90 above whenever **CMD** precedes the **CLOSE** command.

---

# Commodore 64 BASIC

The Commodore 64 BASIC commands, statements, and functions are listed below, with proper syntax and brief definitions of each, where

$a$  = any argument

$S\$$  = any string variable

$n$  = any number or numeric expression

$ln$  = program line number

$N$  = any numeric variable

$D\#$  = device number

$s$  = any string

$f\#$  = file number

Entries marked with an asterisk are not covered in this book (see your *Commodore 64 User's Manual*).

**AND** Logical operator—all conditions must test true.

**ASC( $s/S\$$ )** Returns ASCII code number for the first character of any string.

**ATN( $n/N$ )** Returns the arctangent of  $n/N$  in radians.

**CHR\$( $n/N$ )** Returns the character with the ASCII code number specified by  $n/N$ .

**CLOSE  $d\#$**  CLOSEs the channel to any device ( $d\#$ ) previously OPENed for input/output.

**CLR\*** Clears variable, array and user-defined function values from memory; resets **GOSUB** and **DATA** pointers.

**CMD  $f\#$**  Directs output of file ( $f\#$ ) to device other than the screen display.

**CONT** Continues a program after encountering either a **STOP** or **END** statement.

**COS( $n/N$ )** Returns the COSine of  $n/N$  in radians.

**DATA** Provides a list of elements (string or numeric) to be **READ**.



- DEF FN  $X(N)$**  User-defined function (**DEF FN**) with the name  $X$  as it applies to variable  $N$ .
- DIM  $A(n/N)$ ; DIM  $A$( $n/N$ )$**  Reserves memory space for 1-, 2-, or 3-dimensional arrays of 11 or more elements.
- END** Indicates the **END** of a program.
- EXP( $n/N$ )** Returns the exponential value of  $n/N$ .
- FN  $X(n/N)$**  Returns the function (**FN**) named  $X$  as it applies to  $n/N$ .
- FOR  $X=n$  TO  $m$ ; NEXT  $X$**  Sets the parameters ( $n$  TO  $m$ ) of **FOR/NEXT** loop  $X$ , where  $X$  increases by 1 with each loop when recalled by **NEXT  $X$** .
- FRE( $a$ )** Returns the number of free bytes in RAM (Random Access Memory).
- GET  $N$  or  $S$$**  Assigns a value, one character at a time, to a variable, numeric or string.
- GOSUB  $ln$ ; RETURN** Branches program control to the subroutine starting at program line  $ln$ . The **RETURN** statement at the end of the subroutine branches back to the main program line directly following the **GOSUB** statement.
- GOTO  $ln$**  Branches the program to program line  $ln$ .
- IF/THEN** Conditional statement; only **IF** the condition is satisfied, **THEN** execute.
- INPUT  $N$  or  $S$$**  Assigns user-**INPUT** to variables, either numeric or string.
- INT( $n/N$ )** Returns **INT**eger of  $n/N$ .
- LEFT\$( $s/S$, $n/N$ )$**  Slices the **LEFT**-most ( $n/N$ ) characters of a string ( $s/S$$ ).
- LEN( $s/S$$ )** Returns the total number of characters and spaces of  $s/S$$ .
- LIST; LIST  $ln$ ; LIST  $ln-ln$**  **LIST**s a program, a single program line, or set of consecutive program lines, respectively.
- LOAD "X"** **LOAD**s the program named  $X$  from either tape or disk into computer memory.
- LOG( $n/N$ )** Returns the natural **LOG**-arithm of  $n/N$ .
- MID\$( $s/S$, $n/N$ , $n/N$ )$**  Slices the **MID**-dle characters (from  $n/N$  to  $n/N$ ) of a string ( $s/S$$ ).
- NEW** Erases the current program in memory.
- NEXT** See **FOR/NEXT**.
- NOT** Logical operator for negation (same as  $<>$ ).

**ON  $n/N$  GOTO  $ln1,ln2,\dots$ ; ON  $n/N$  GOSUB  $ln1,ln2,\dots$**  Branches (**GOTO** or **GOSUB**) to program line  $ln1$ ,  $ln2$ , etc., depending on the value of  $n/N$ .

**OPEN  $f\#,d\#,o\#$**  **OPENS** input/output channel for file  $f\#$  from device  $d\#$  for operation  $o\#$  (operation argument not required if device is either keyboard or display).

**OR** Logical operator—one of at least two conditions must test true.

**PEEK( $n/N$ )** Returns the value at memory address  $n/N$ .

**POKE  $n/N,X$**  **POKEs** (stores) one byte of information ( $X$ ) into memory at address  $n/N$ .

**POS( $a$ )** Returns the absolute column number **POSITION** ( $0-39$ ) of the cursor on display.

**PRINT  $n/N$ ; PRINT "s" or  $S\#$**  **PRINTs** numbers or strings to display or other output device.

**READ  $N$  or  $S\#$**  **READs**, then assigns numeric or string **DATA** from **DATA** statements to corresponding numeric or string variables.

**REM** **REMARK** statement stores comments or notations in programs for user benefit only. All **REM** statements are ignored by the computer.

**RESTORE** **RESTOREs** the **DATA** statement pointer to allow **DATA** to be re-**READ**.

**RETURN** See **GOSUB**.

**RIGHT\$( $s/S\#,n/N$ )** Slices **RIGHT**most letters ( $n/N$ ) from string  $s/S\#$ .

**RND( $a$ )** Generates a pseudorandom number greater or equal to  $0$  but less than  $1$ .

**RUN; RUN  $ln$**  **RUNs** the program currently in memory from the beginning, or from a specified program line.

**SAVE "file name", $d\#$**  **SAVEs** the file name indicated in quotes to a specified device ( $d\#$ ). If no device is specified, device number automatically defaults to  $1$  (tape).

**SGN( $n/N$ )** Returns the signum— $1$  if  $n/N$  is positive,  $0$  if  $n/N$  is zero, and  $-1$  if  $n/N$  is negative.

**SIN( $n/N$ )** Returns the sine of  $n/N$  in radians.

**SPC( $a$ )** Inserts spaces in a **PRINT** statement.

**SQR( $n/N$ )** Returns the square root of  $n/N$ .

**STOP** **STOPs** execution of a **BASIC** program from within the program mode.

**STR\$(*n/N*)** Converts numeric *n/N* into a string representation—123 to “123”.

**SYS (*n/N*)\*** Recalls machine language subroutine located at address *n/N*.

**TAB(*n/N*)** TABs to the column position indicated by *n/N*.

**TAN(*n/N*)** Tangent of *n/N* in radians.

**USR(*n/N*)\*** Recalls assembly language subroutine located at address *n/N*.

**VAL(*s/S*)** Converts suitable string *s/S* into numeric equivalent—“123” to 123.

**VERIFY “file name”,*d*#** Verifies whether a specific file (“file name”) stored in device *d*# (tape or disk) precisely matches the program currently in memory.

**WAIT \*** Halts the program until the contents of a specified memory location change in a specific way.

## PUNCTUATION

---

- ; PRINT at next column position.
- , PRINT at next TAB position.
- : Separates multiple statements on the same line.
- “ ” Indicates literal string.

## ARITHMETIC OPERATORS

---

- = Assigns value to variable
- + Addition
- − Subtraction, negation
- \* Multiplication
- / Division
- ↑ Exponentiation

## RELATIONAL OPERATORS

---

- = Equal to
- < Less than
- > Greater than
- <= Less than or equal to
- >= Greater than or equal to
- <> Not equal to

**LOGICAL OPERATORS**

---

<b>AND</b>	All conditions test true
<b>OR</b>	One condition tests true
<b>NOT</b>	Condition tests false


































## APPENDIX 3




































# CHR\$/ASC Codes

The following table lists all CHR\$ Characters or Keys and their equivalent ASC Codes:

<b>CHR\$ Character/Key</b>	<b>ASC Code</b>	<b>CHR\$ Character/Key</b>	<b>ASC Code</b>
	0	CLR/HOME	19
	1	INST/DEL	20
	2		21
	3		22
	4		23
WHT	5		24
	6		25
	7		26
SHIFT-COMMODORE	8		27
SHIFT-COMMODORE	9	RED	28
	10	CRSR →	29
	11	GRN	30
	12	BLU	31
RETURN	13	SPACE	32
switch to lower case	14	!	33
	15	"	34
	16	#	35
CRSR ↓	17	\$	36
RVS ON	18	%	37




<b>CHR\$</b> <b>Character/Key</b>	<b>ASC</b> <b>Code</b>	<b>CHR\$</b> <b>Character/Key</b>	<b>ASC</b> <b>Code</b>
&	38	@	64
.	39	A	65
(	40	B	66
)	41	C	67
*	42	D	68
+	43	E	69
,	44	F	70
-	45	G	71
.	46	H	72
/	47	I	73
0	48	J	74
1	49	K	75
2	50	L	76
3	51	M	77
4	52	N	78
5	53	O	79
6	54	P	80
7	55	Q	80
8	56	Q	81
9	57	R	82
:	58	S	83
;	59	T	84
<	60	U	85
=	61	V	86
>	62	W	87
?	63	X	88

CHR\$ Character/Key	ASC Code	CHR\$ Character/Key	ASC Code
Y	89		115
Z	90		116
[	91		117
£	92		118
]	93		119
↑	94		120
←	95		121
	96		122
	97		123
	98		124
	99		125
	100		126
	101		127
	102		128
	103		129
	104		130
	105		131
	106		132
	107	f1	133
	108	f3	134
	109	f5	135
	110	f7	136
	111	f2	137
	112	f4	138
	113	f6	139
	114	f8	140

CHR\$ Character/Key	ASC Code	CHR\$ Character/Key	ASC Code
SHIFT-RETURN	141		165
switch to upper case	142		166
	143		167
BLK	144		168
CRSR ↑	145		169
RVS OFF	146		170
CLR/HOME	147		171
INST/DEL	148		172
	149		173
	150		174
	151		175
	152		176
	153		177
	154		178
	155		179
PUR	156		180
CRSR ←	157		181
YEL	158		182
CYN	159		183
SPACE	160		184
	161		185
	162		186
	163		187
	164		188



---

<b>CHR\$ Character/Key</b>	<b>ASC Code</b>
	189
	190
	191

Codes 192–223 will return the same characters as codes 96–127.  
Codes 224–254 will return the same characters as codes 160–190.  
Code 255 will return the same characters as code 126.

## APPENDIX 4

# Error Messages

Your Commodore 64's complaints assume the form of error messages that not only define the nature of the problem, but locate it, pinpointing the "bugged" program line number. The following is an alphabetical list of possible error messages with brief definitions of each.

**BAD DATA** String data received instead of anticipated numeric data.

**BAD SUBSCRIPT** Array number beyond specified **DIM** statement.

**CAN'T CONTINUE** **CONT** command will not work.

**DEVICE NOT PRESENT** Requested **IN/OUT** device not available.

**DIVISION BY ZERO** Operation not defined and not allowed.

**EXTRA IGNORED** Too many **INPUT** elements.

**FILE NOT FOUND** End-of-tape marker encountered on tape, or the requested file was not found on disk.

**FILE OPEN** Required file already **OPEN**.

**FORMULA TOO COMPLEX** System requires division of string expression, or there are too many parentheses.

**ILLEGAL DIRECT INPUT** entered in Immediate Mode.

**ILLEGAL QUANTITY** Function/Statement argument out of range.

**LOAD** Problem with taped program.

**NEXT WITHOUT FOR** Incorrectly nested **FOR/NEXT** loops or an extra **NEXT** statement with no corresponding **FOR** statement.

**NOT INPUT FILE** Cannot receive data (**INPUT** or **GET**) from output file.

**NOT OUTPUT FILE** Cannot **PRINT** data from **INPUT** file.

**OUT OF DATA** No more **DATA** to be **READ**.

**OUT OF MEMORY** No more **RAM** (Random Access Memory) available for program or variables; too many nested **FOR/NEXT** loops; too many **GOSUB** statements.

**OVERFLOW** Number too large.

**REDIM'D ARRAY** Same array **DIM**-ensioned more than once.

**REDO FROM START** Character data **INPUT** in response to numeric data request.

**RETURN WITHOUT GOSUB**  
**RETURN** statement encountered without preceding **GOSUB**.

**STRING TOO LONG** Maximum string length is 255 characters.

**SYNTAX ERROR** Unrecognized statement, missing or extra parentheses, misspelled statements, etc.

**TYPE MISMATCH DATA** elements mismatched against storage variable (string to numeric or the opposite).

**UNDEF'D FUNCTION** Request for an un-**DEF**ined function (**DEF FN** statement).

**UNDEF'D STATEMENT** Attempt to branch (**GOTO** or **GOSUB**) the program to a nonexistent program line.

**VERIFY** Program on tape or disk does not precisely match the current program in memory.

## APPENDIX 5

# PEEK/POKE Codes

Your Commodore 64 can generate two different character sets:

A. Uppercase—POKE 53272,21, and

B. Lowercase—POKE 53272,23

The following chart demonstrates the Character generated by each Character Set (A and B) for the corresponding Character Set Code Number:

---



























Character Set A	Character Set B	Character Set Code Number
<i>␣</i>		0
A	a	1
B	b	2
C	c	3
D	d	4
E	e	5
F	f	6
G	g	7
H	h	8
I	i	9
J	j	10
K	k	11
L	l	12




























---

<b>Character Set A</b>	<b>Character Set B</b>	<b>Character Set Code Number</b>
M	m	13
N	n	14
O	o	15
P	p	16
Q	q	17
R	r	18
S	s	19
T	t	20
U	u	21
V	v	22
W	w	23
X	x	24
Y	y	25
Z	z	26
		27
£		28
		29
↑		30
←		31
SPACE		32
!		33
"		34
#		35
\$		36
%		37















---

<b>Character Set A</b>	<b>Character Set B</b>	<b>Character Set Code Number</b>
&		38
'		39
(		40
)		41
*		42
+		43
,		44
-		45
.		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

Character Set A	Character Set B	Character Set Code Number
		64
	A	65
	B	66
	C	67
	D	68
	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

Character Set A	Character Set B	Character Set Code Number
	Z	90
		91
		92
		93
		94
		95
SPACE		96
		97
		98
		99
		100
		101
		102
		103
		104
		105
		106
		107
		108
		109
		110
		111
		112
		113
		114



Character Set A	Character Set B	Character Set Code Number
		115
		116
		117
		118
		119
		120
		121
		122
		123
		124
		125
		126
		127

Character set code numbers from 128 through 255 return inverse characters of codes 0 through 127.

## APPENDIX 6

# Musical Note Values

Here is a list of musical note values (8 octaves) with their corresponding high and low frequency register numbers:

---

<b>Octave</b>	<b>Note</b>	<b>High frequency register #</b>	<b>Low frequency register #</b>
0	C	1	12
0	C#	1	28
0	D	1	45
0	D#	1	62
0	E	1	81
0	F	1	102
0	F#	1	123
0	G	1	145
0	G#	1	169
0	A	1	195
0	A#	1	221
0	B	1	250
1	C	2	24
1	C#	2	56
1	D	2	90
1	D#	2	125
1	E	2	163

---

<b>Octave</b>	<b>Note</b>	<b>High frequency register #</b>	<b>Low frequency register #</b>
1	F	2	204
1	F#	2	246
1	G	3	35
1	G#	3	83
1	A	3	134
1	A#	3	187
1	B	3	224
2	C	4	48
2	C#	4	112
2	D	4	180
2	D#	4	251
2	E	5	71
2	F	5	152
2	F#	5	237
2	G	6	71
2	G#	6	167
2	A	7	12
2	A#	7	119
2	B	7	223
3	C	8	97
3	C#	8	225
3	D	9	104
3	D#	9	247
3	E	10	143
3	F	11	48

---

<b>Octave</b>	<b>Note</b>	<b>High frequency register #</b>	<b>Low frequency register #</b>
3	F#	11	218
3	G	12	143
3	G#	13	78
3	A	14	24
3	A#	14	239
3	B	15	210
4	C	16	195
4	C#	17	195
4	D	18	209
4	D#	19	239
4	E	21	31
4	F	22	96
4	F#	23	181
4	G	25	30
4	G#	26	49
4	A	28	49
4	A#	29	223
4	B	31	165
5	C	33	135
5	C#	35	134
5	D	37	162
5	D#	39	223
5	E	42	62
5	F	44	193
5	F#	47	107

---

Octave	Note	High frequency register #	Low frequency register #
5	G	50	60
5	G#	53	57
5	A	56	99
5	A#	59	190
5	B	63	75
6	C	67	15
6	C#	71	12
6	D	75	69
6	D#	79	191
6	E	84	125
6	F	89	131
6	F#	94	214
6	G	100	121
6	G#	106	115
6	A	112	199
6	A#	119	124
6	B	126	151
7	C	134	30
7	C#	142	24
7	D	150	139
7	D#	159	126
7	E	168	250
7	F	179	6
7	F#	189	172
7	G	200	243

---

<b>Octave</b>	<b>Note</b>	<b>High frequency register #</b>	<b>Low frequency register #</b>
7	G#	212	230
7	A	225	143
7	A#	238	248
7	B	253	46

## APPENDIX 7

---

# Sample Exercise Answers

Remember, each of the following programs represents just one of the many possible (and equally correct) solutions.

## CHAPTER 1

---

- 1-1. Write a program to **PRINT** your name, address, and phone number (like a business card) in the middle of the screen.

```
10 PRINT " (CLR/HOME) "  
20 PRINT: PRINT: PRINT: PRINT: PRINT  
30 PRINT: PRINT: PRINT: PRINT: PRINT  
40 PRINT TAB (12) ; "JEFFREY EINSTEIN"  
50 PRINT TAB (14) ; "#1 BYTE LANE"  
60 PRINT TAB (11) ; "NEW YORK, NY 11111"  
70 PRINT TAB (14) ; "212-123-4567"
```

Lines 20 and 30 **PRINT** 10 blank lines to position the cursor halfway down the screen. Notice how the **TAB** instructions are all separated from the literal strings (characters in quotes) by semicolons.

- 1-2. Write a program to **PRINT** HELLO in three separate columns down the screen. (*Hint*: Use a **GOTO** loop.)

```
10 PRINT " (CLR/HOME) "  
20 PRINT "HELLO"; TAB (16) ; "HELLO"; TAB (32) ; "HELLO"  
30 GOTO 20
```

- 1-3. Write a program to **PRINT** your name once at line 1, column 10; once again at line 11, column 10; and once more at line 21, column 10.

```
10 PRINT " (CLR/HOME) "  
20 PRINT TAB (10) ; "JEFFREY EINSTEIN"  
30 PRINT: PRINT: PRINT: PRINT: PRINT
```

```

40 PRINT:PRINT:PRINT:PRINT:PRINT
50 PRINT TAB(10); "JEFFREY EINSTEIN"
60 PRINT:PRINT:PRINT:PRINT:PRINT
70 PRINT:PRINT:PRINT:PRINT
80 PRINT TAB(10); "JEFFREY EINSTEIN"

```

## CHAPTER 3

---

- 3-1. Write a program to determine the area (in square inches) of rectangle X if side A is 20 inches long and side B is 25 times longer than side A. Assign variables to sides A and B.

*Explanation of variables*

A = Side A

B = Side B

```

10 PRINT " (CLR/HOME) "
20 A=20
30 B=A*25
40 PRINT "SIDE A = "; A; " INCHES LONG"
50 PRINT "SIDE B = "; B; " INCHES LONG"
60 PRINT:PRINT "RECTANGLE X = "; A*B; "SQ. INCHES"

```

- 3-2. Assume that your car gets 25 miles per gallon of gasoline (nice car!). Now assume that you must drive 555 miles to pick up your wife at her mother's house. (No, you can't leave her there, and no, she can't take a bus.) The gasoline will cost you \$1.15 per gallon. Write a program to determine how many gallons of gas the round trip would require and how much it would cost. Assign variables to the values 22, 555, and 1.15.

*Explanation of variables*

MPG = Miles Per Gallon

M = Miles

G = Cost of Gasoline

RT = Round Trip

```

10 PRINT " (CLR/HOME) "
20 MPG=25
30 M=555
40 G=1.15
50 PRINT "MILES PER GALLON = "; MPG
60 PRINT "MILES (ONE WAY) = "; M

```



```

70 PRINT "GASOLINE = sp$"; G; "PER GALLON"
80 RT = (M/MPG) * 2
90 PRINT: PRINT "TRIP = sp"; RT; "GALLONS OF GAS"
100 PRINT: PRINT "TOTAL COST = sp$"; RT * G

```

- 3-3. Write a program to **PRINT YOUR NAME + YOUR LOVER'S NAME** by assigning each name to a different string variable, then concatenating them.

```

10 PRINT " (CLR/HOME) "
20 HE$ = "JACKsp + sp"
30 SHE$ = "JILL"
40 PRINT HE$ + SHE$

```

## CHAPTER 4

---

- 4-1. Suppose you opened a savings account in 1980 with \$500 dollars at 11% annual interest. Write a program to **PRINT** your yearly balance (savings plus interest) for the next 15 years. Use a conditional statement to end the program at the right year.

*Explanation of variables*

Y = Year

D = Deposit

I = Interest

```

10 PRINT " (CLR/HOME) "
20 Y = 1980
30 D = 500
40 I = . 11
50 D = D + (D * I)
60 PRINT Y; "SAVINGS + INTEREST = "; D
70 Y = Y + 1
80 IF Y < 1996 THEN 50

```

Line 50 computes yearly savings plus interest.

Line 70 adds 1 to the value of Y with each loop.

Line 80 is a conditional statement to loop lines 50, 60, and 70 until Y is no longer less than 1996.

- 4-2. Now amend the above program to **PRINT** only the year in which your total savings surpass \$2000.

Amend line 60 as follows:

```
60 IF D>2000 THEN PRINT Y; "SAVINGS + INTEREST = ";
  D: END
```

Now line 60 will **PRINT** only **IF** variable D is greater than 2000. Note also how the **END** statement at the end of program line 60 **ENDs** the program the moment line 60 tests true.

## CHAPTER 5

---

- 5-1. Amend the program you wrote for Exercise 4-1 to allow you to **INPUT** both your initial deposit and the interest rate.

Amend lines 30 and 40 as follows:

```
30 INPUT "DEPOSIT = "; D
40 INPUT "INTEREST = "; I
```

- 5-2. Write a program to calculate the area of circle X in square inches. (*Hint: INPUT* will allow you to alter the radius length, and the equation Radius \* Radius \* PI, where PI equals 3.141 592 653, will calculate the area of circle X in square inches.

*Explanation of variables*

R = Radius

PI = 3.141 592 653

```
10 PRINT "(CLR/HOME) "
20 INPUT "RADIUS = "; R
30 PI = 3.141592653
40 PRINT: PRINT "CIRCLE X = "; R*R*PI; "SQ. INCHES"
```

Line 20 requests the **INPUT**. R\*R\*PI in line 40 calculates the area of circle X.

## CHAPTER 6

---

- 6-1. Write a program to list multiples of 25 from 0 to 1000.

```
10 PRINT "(CLR/HOME) "
20 FOR X=25 TO 1000 STEP 25
30 PRINT X,
40 NEXT
```

Line 20 sets the parameters of **FOR/NEXT** loop X from 25 **TO** 1000.

Line 40 recalls **FOR/NEXT** loop X at increments of 25 (**STEP** 25).

- 6-2. Amend your program from Exercise 4-1 using a **FOR/NEXT** loop to reveal your yearly balance from 1980 to the year 2000.

```

10 PRINT " (CLR/HOME) "
20 D=500
30 I=.11
40 FOR Y= 1980 TO 2000
50 D=D+(D*I)
60 PRINT Y; "SAVINGS + INTEREST = ";D
70 NEXT

```

Line 40 sets the parameters of **FOR/NEXT** loop Y from 1980 **TO** 2000.

## CHAPTER 7

---

- 7-1. Use **RND** to generate a random number in the range from 1 to 50. Use **INPUT** and a **FOR/NEXT** loop to give yourself 8 chances to guess the random number. Have the computer tell you whether each guess is too high, too low, or correct, and then tell you the correct number after 8 incorrect guesses.

### *Explanation of variables*

RN = Random Number

G = Guess

GG = **INPUT** Guess

```

10 PRINT " (CLR/HOME) "
20 PRINT "GUESS MY NUMBER (1-50) "
30 PRINT "YOU GET 8 GUESSES":PRINT
40 RN=INT(50*RND(1))+1
50 FOR G=1 TO 8
60 PRINT:PRINT "GUESS #";G;"="";:INPUT GG
70 IF GG=RN THEN 150
80 IF GG>RN THEN 110
90 PRINT GG;" IS TOO SMALL"
100 GOTO 120
110 PRINT GG;" IS TOO BIG"
120 NEXT
130 PRINT " (CLR/HOME)":PRINT "STUPE, MY NUMBER WAS";RN
140 GOTO 300

```

```

150 PRINT " (CLR/HOME) ":FOR X=1 TO 10:PRINT:NEXT
160 PRINT "BRAVO!! ";GG;" IS CORRECT":PRINT
170 PRINT "IT TOOK YOU";G;" GUESSES"
300 PRINT:PRINT:INPUT "PRESS RETURN TO GO AGAIN";A$
310 IF A$<>" " THEN 300
320 GOTO 10

```

Line 40 generates a random number between 1 and 50.

Line 50 sets the parameters of **FOR/NEXT** loop G from 1 **TO** 8 for a maximum of 8 guesses.

Lines 70 and 80 test your **INPUT** GG against variable RN.

- 7-2. Write a program to list the years 1980–2000, telling whether each year is or isn't a leap year. [*Hint*: Any year that is a multiple of 4 is a leap year (remember **INT**).]

```

10 PRINT " (CLR/HOME) "
20 FOR Y=1980 TO 2000
30 IF Y/4=INT(Y/4) THEN 60
40 PRINT Y;" IS NOT A LEAP YEAR"
50 GOTO 70
60 PRINT Y;" IS A LEAP YEAR"
70 NEXT

```

Line 20 sets the parameters of **FOR/NEXT** loop y (for year) from 1980 **TO** 2000.

Line 30 is a conditional test and will branch to line 60 only **IF**  $Y/4 = \text{INT}(Y/4)$ .

- 7-3. Amend Exercise 4-1 to round off your yearly savings total to the nearest cent.

Amend line 60 as follows:

```

60 PRINT Y;"SAVINGS + INTEREST = ";INT(D*100+.5)/100

```

## CHAPTER 8

---

- 8-1. Use a **FOR/NEXT** loop and a **READ/DATA** statement to **PRINT** the sentence THIS WAS EASIER THAN I THOUGHT, in which each word represents a single string **DATA** element.

```

10 PRINT " (CLR/HOME) "
20 FOR W=1 TO 6

```

```

30 READ D$
40 PRINT "sp";D$;
50 NEXT
60 DATA THIS, WAS, EASIER, THAN, I, THOUGHT

```

Line 20 sets the parameters of **FOR/NEXT** loop W (for Word) from 1 TO 6.

Line 30 **READS** the string **DATA** elements.

Line 40 **PRINTS** a blank space followed by the current string value of D\$.

Line 60 stores the string **DATA** elements.

- 8-2. Write a program that uses **DATA** statements to store the alphabet (A through Z), and a corresponding numerical value for each letter, for example,

```
DATA A, 1, B, 2, C, 3, D, 4, . . .
```

Have the program (a) request you to **INPUT** any letter, search the **DATA** statements for the requested letter, then **PRINT** both the letter and its numeric equivalent; (b) inform you when your **INPUT** was not a letter (a number or other character), and consequently not found in the **DATA** statements; (c) loop back to beginning to request and search for another letter (remember **RESTORE**).

```

1  REM CHAPTER EXERCISE 8-2
10 PRINT " (CLR/HOME) "
20 INPUT "ENTER ANY LETTERsp";LTR$
30 RESTORE
40 FOR X=1 TO 26
50 READ L$:READ N
60 IF L$=LTR$ THEN 110
70 NEXT
80 PRINT " (CLR/HOME) "
90 PRINT "CHARACTERsp";LTR$;"spWAS NOT FOUND"
100 FOR X=1 TO 3000:NEXT:GOTO 10
110 PRINT " (CLR/HOME) "
120 PRINT:PRINT "LETTERsp";LTR$;"spHAS THE NUMBER
    VALUE";N
130 FOR X=1 TO 3000:NEXT:GOTO 10
200 DATA A, 1, B, 2, C, 3, D, 4, E, 5, F, 6
210 DATA G, 7, H, 8, I, 9, J, 10, K, 11
220 DATA L, 12, M, 13, N, 14, O, 15, P, 16
230 DATA Q, 17, R, 18, S, 19, T, 20, U, 21
240 DATA V, 22, W, 23, X, 24, Y, 25, Z, 26

```

## CHAPTER 9

---

- 9-1. Write a single subroutine to flash heads or tails, depending on which phrase is chosen by a randomly generated number in the main program.

```

10 PRINT " (CLR/HOME) "
20 RN=INT(2*RND(1))+1
30 IF RN=1 THEN 50
40 N$="TAILS":GOTO 60
50 N$="HEADS"
60 GOSUB 100
70 GOTO 10
100 FOR A=1 TO 10
110 FOR B=1 TO 10:PRINT:NEXT
120 PRINT TAB(17);N$
130 FOR C=1 TO 50:NEXT
140 PRINT " (CLR/HOME) "
150 NEXT A
160 RETURN

```

Line 20 generates either 1 or 2.

Line 30 tests the current value of variable RN and will branch to line 50 only IF RN equals 1.

Line 60 recalls the subroutine at line 100.

**FOR/NEXT** loop A controls the flashing.

**FOR/NEXT** loop B moves the cursor down 10 lines.

**FOR/NEXT** loop C is an empty loop to slow the flashing.

- 9-2. In Exercise 7-1 you wrote a program to list the multiples of 25 from 0 to 1000 using a **FOR/NEXT** loop. Now write a program that requests you to **INPUT** any number from 5 to 25, then branches to a subroutine that **PRINTs** the multiples of your **INPUT** number to 1000 before **RETURNing** to request another **INPUT** number.

```

1 REM CHAPTER EXERCISE 9-2
10 PRINT " (CLR/HOME) "
20 INPUT "ENTER ANY NUMBER (5-25) ";N
30 IF N<5 OR N>25 THEN 10
40 GOSUB 100
50 GOTO 10
100 PRINT " (CLR/HOME) ":FOR X=N TO 1000 STEP N

```

```

110 PRINT X,
120 NEXT
130 FOR X=1 TO 3000:NEXT:RETURN

```

## CHAPTER 10

---

- 10-1. Write a program that requests you to **INPUT** two whole numbers, **PRINT**s them, and tells whether each is odd or even.

```

10 PRINT " (CLR/HOME) "
20 INPUT "ENTER ANY NUMBER";N1
30 PRINT:INPUT "ENTER ANOTHER NUMBER";N2
40 IF N1/2=INT(N1/2) THEN PRINT N1;" IS EVEN":GOTO 60
50 PRINT N1;" IS ODD"
60 IF N2/2<>INT(N2/2) THEN PRINT N2;" IS ODD":GOTO 80
70 PRINT N2;" IS EVEN"
80 FOR X=1 TO 2000:NEXT:GOTO 10

```

Line 40 tests **INPUT** N1.

Line 50 will **PRINT** only **IF** line 40 tests false— $N1/2 \neq \text{INT}(N1/2)$ .

Line 60 tests **INPUT** N2.

Line 70 will **PRINT** only **IF** line 60 tests false— $N2/2 = \text{INT}(N2/2)$ .

- 10-2. Write a program that generates a random number between 1 and 5, then requests you to **INPUT** any two numbers. Have your Commodore **PRINT** "BRAVO! YOU WIN!" only if the second **INPUT** number equals either the first **INPUT** number *plus* the random number or the first **INPUT** number *minus* the random number. Otherwise, have the program loop back to request two new **INPUT** numbers.

```

1 REM CHAPTER EXERCISE 10-2
10 PRINT " (CLR/HOME) "
20 RN=INT(5*RND(1))+1
30 INPUT "ENTER ANY NUMBER";N1
40 INPUT "ENTER ANOTHER NUMBER";N2
50 A=(N2=(N1+RN) OR N2=(N1-RN))
60 IF A=-1 THEN PRINT "BRAVO! YOU WIN!":STOP
70 PRINT "(CLR/HOME)":PRINT "TRY AGAIN"
80 FOR X=1 TO 3000:NEXT
90 PRINT "(CLR/HOME)":GOTO 30

```

## CHAPTER 11

---

- 11-1. Words that begin or end with the letters ASS are ASSes. Write a program that requests you to **INPUT** a word, then lists whether or not your **INPUT** word is an ASS according to the above definition.

```

10 PRINT " (CLR/HOME) "
20 INPUT "ENTER ANY WORDsp"; W$
30 IF LEFT$(W$, 3) = "ASS" OR RIGHT$(W$, 3) = "ASS" THEN
    PRINT W$; "spIS AN ASS": GOTO 50
40 PRINT W$; "spIS NOT AN ASS"
50 FOR X=1 TO 2000: NEXT
60 GOTO 10

```

Line 30 tests the first and last 3 letters of string variables W\$ (**LEFT\$** and **RIGHT\$**).

Line 40 will **PRINT** only **IF** line 30 tests false.

- 11-2. Write a program to alphabetize the last names of up to 10 of your friends.

```

10 PRINT " (CLR/HOME) "
20 FOR X=1 TO 10
30 INPUT "ENTER ANY NAMEsp"; W$(X)
40 NEXT
50 PRINT " (CLR/HOME) "
60 S=0
70 FOR X=1 TO 9
80 IF W$(X) <= W$(X+1) THEN 110
90 A$=W$(X) : W$(X)=W$(X+1) : W$(X+1)=A$
100 S=S+1
110 NEXT
120 IF S=1 THEN 60
130 FOR X=1 TO 10
140 IF W$(X) <> "" THEN PRINT W$(X)
150 NEXT

```

The above program is an example of a *bubble sort* routine. Line 80 tests each string against subsequent strings.



**CHAPTER 12**

- 12-1. Write a program that will (a) simulate the roll of a pair of dice 10 times; (b) display the individual face value of each die and the total for each roll; (c) display the total number of 1's, 2's, 3's, 4's, 5's and 6's rolled.

*Explanation of Variables*

T = Toss of Dice

R = Roll of One Die

A = Value of 2nd Die

```

10 PRINT " (CLR/HOME) "
20 DIM D(6)
30 FOR T=1 TO 10
40 GOSUB 200
50 PRINT TAB(10);R;
60 A=R
70 GOSUB 200
80 PRINT TAB(15);" + ";TAB(20);R;TAB(25);" = ";TAB(30);A+R
90 NEXT
100 FOR T=1 TO 6
110 PRINT TAB(17);D(T);TAB(22);T;"S"
120 NEXT
130 STOP
200 R=INT(6*RND(1))+1
210 D(R)=D(R)+1
220 RETURN

```

**DIM D(6)** reserves memory space for 6 subscript elements of array D. Lines 40 and 70 recall the subroutine beginning at line 200 to assign separate values to each die.

**note:** It is not technically necessary to assign a **DIM** statement to arrays of less than 11 elements, although it is good programming practice.

- 12-2. *Hit or Miss.* Write a program that will generate a four-digit number, with each digit having a value 1 to 4. Give yourself 8 chances to guess the number. Have the computer tell you how many HITS (right digits in the right sequence) you score for each guess and then tell you the right number after eight incorrect guesses.

*Explanation of variables*

N = Number  
 C = Chances  
 H = Hits  
 G\$ = Guess

```

10 PRINT " (CLR/HOME) "
20 DIM N(4)
30 PRINT "MY NUMBER IS 4 DIGITS, EACH 1-4"
40 PRINT "YOU HAVE 8 CHANCES TO GUESS IT":PRINT
50 FOR J=1 TO 4
60 N(J) = INT(4*RND(1)) + 1
70 NEXT
80 FOR C=1 TO 8
90 H=0
100 INPUT G$
110 FOR J=1 TO 4
120 IF N(J) = VAL(MID$(G$, J, 1)) THEN H=H+1
130 NEXT
140 IF H=4 THEN 200
150 PRINT C; " - sp"; G$; " = "; H; "HITS"
160 NEXT C
170 PRINT:PRINT:PRINT "STUPE, MY NUMBER WAS";N(1);N(2);
    N(3);N(4)
180 GOTO 210
200 PRINT:PRINT:PRINT C; " - sp"; G$; "spIS CORRECT"
210 PRINT:PRINT "PRESS ANY KEY TO GO AGAIN"
220 GET A$: IF A$="" THEN 220
230 PRINT " (CLR/HOME) ":GOTO 30

```

**DIM** N(4) reserves memory space for 4 separate elements of single-**DIM**ension array N.

The first **FOR/NEXT** loop J (lines 50–70) assigns different random digits (each 1–4) to the four separate elements of single-**DIM**ension array N.

The second **FOR/NEXT** loop J (lines 110–130) tests the individual digit values of your **INPUT** G\$ against the individual elements of single-**DIM**ension array N and adds the total number of **HITS** ( $H = H + 1$ ).

## CHAPTER 13

---

- 13-1. Write a program to **POKE** any **INPUT** number (between 0–255) into memory addresses 35001, 35002, and 35003.

```

1  REM CHAPTER EXERCISE 13-1
10 PRINT " (CLR/HOME) "
20 M=35000
30 FOR X=1 TO 3
40 INPUT "ENTER ANY NUMBER (0-255) ";N(X)
50 IF N(X)<0 OR N(X)>255 THEN 40
60 POKE M+X,N(X)
70 NEXT

```

- 13-2. Amend the above program to **PRINT** the characters corresponding to the numerical values now stored in memory addresses 35001, 35002, and 35003.

```

1  REM CHAPTER EXERCISE 13-2
2  REM ADD LINES 80-120
10 PRINT " (CLR/HOME) "
20 M=35000
30 FOR X=1 TO 3
40 INPUT "ENTER ANY NUMBER (0-255) ";N(X)
50 IF N(X)<0 OR N(X)>255 THEN 40
60 POKE M+X,N(X)
70 NEXT
80 PRINT:PRINT "PRESS ANY KEY"
90 GET A$:IF A$="" THEN 90
100 FOR X=1 TO 3
110 PRINT:PRINT M+X;" = sp";CHR$(PEEK(M+X))
120 NEXT

```

## CHAPTER 14

---

- 14-1. Write a program to construct a four-walled playing court, then send a ball bouncing all around the court.

*Explanation of variables*

S = Screen Memory Map Position

C = Color Memory Map Position

H = Horizontal Position Counter

V = Vertical Position Counter

```

10 PRINT CHR$(147)
20 POKE 53280,0:POKE 53281,1
30 S=1024:C=55296

```

```

40 H = 1 : V = 1
50 HH = 1 : VV = 1
60 POKE S + H + 40 * V, 81 : POKE C + H + 40 * V, 0
70 FOR X = 1 TO 25 : NEXT
80 POKE S + H + 40 * V, 32 : POKE C + H + 40 * V, 1
90 H = H + HH : V = V + VV
100 IF H <= 0 OR H >= 39 THEN HH = -HH
110 IF V <= 0 OR V >= 24 THEN VV = -VV
120 GOTO 60

```

Line 20 sets a black border and a white background.

Line 60 **POKE**s a ball into the current H/V coordinates.

Line 80 "erases" the ball at the current H/V coordinates.

Line 90 adds the current values of variables HH and VV (1 or -1) to the current values of variables H and V.

Lines 100 and 110 test the current values of variables H and V to determine when the ball must reverse directions (bounce) off a wall.

#### 14-2. Write a program to generate different random size and color rectangles.

##### *Explanation of variables*

CL = Color  
 HH = Horizontal  
 VV = Vertical  
 S = Screen Memory Map Position  
 C = Color Memory Map Position

```

10 PRINT CHR$(147)
20 POKE 53280, 0
30 POKE 53281, 0
40 S = 1024
50 C = 55296
60 CL = INT(16 * RND(1))
70 HH = INT(40 * RND(1))
80 VV = INT(25 * RND(1))
90 FOR V = 0 TO VV
100 FOR H = 0 TO HH
110 POKE S + H + V * 40, 160 : POKE C + H + V * 40, CL
120 NEXT H
130 NEXT V
140 GOTO 40

```

Lines 20 and 30 set both the border and background color to black.  
 Line 60 generates a random color.  
 Line 70 generates random horizontal parameters.  
 Line 80 generates random vertical parameters.

## CHAPTER 15

---

15-1. Amend the RUNNING NOSE program to change nose colors with each loop.

Amend lines 75 and 160 as follows:

```
75 POKE DC + 39 + 1, C1: POKE DC + 39 + 2, C2
160 GOTO 71
```

And add the following lines:

```
71 C1 = (15*RND(1)): C2 = (15*RND(1))
155 RESTORE
```

15-2. Amend the RUNNING NOSE program to display one expanded nose and one standard nose.

Just amend line 105 as follows:

```
105 POKE DC + 23, 4: POKE DC + 29, 4
```

## CHAPTER 17

---

17-1. Write a program that requests you to **INPUT** the radius of a circle, then calculates and **PRINTs** the area of a circle via the **DEF FN** function.

```
1 REM CHAPTER EXERCISE 17-1
10 PRINT " (CLR/HOME) "
20 INPUT "ENTER RADIUS"; R
30 DEF FN RADIUS (R) = R ^ 2 * 3.141592653
40 PRINT " (CLR/HOME) "
50 PRINT " IF RADIUSsp = "; R; " THEN AREAsp = "; FN RADIUS (R)
```

17-2. Write a program that requests you to **INPUT** the **TAN**gent of an angle, then calculates and **PRINTs** the value of the corresponding angle in degrees.

```
1  REM CHAPTER EXERCISE 17-2
10 PRINT " (CLR/HOME) "
20 INPUT "ENTER TANGENT";T
30 A=ATN(T)*180/3.141592653
40 PRINT " (CLR/HOME) "
50 PRINT:PRINT "ANGLEsp=";A
60 FOR X=1 TO 3000:NEXT:GOTO 10
```

## GLOSSARY

---

- Address** Location (a number) of a specific byte of memory.
- BASIC** Beginner's All-purpose Symbolic Instruction Code. A high-level computer language developed at Dartmouth College, New Hampshire.
- Binary** Base 2 counting system wherein all digits must be either 0 or 1.
- Bit** One-eighth of a byte containing either a 0 or a 1.
- BREAK** Interrupts a program.
- Bug** A program error.
- Byte** An 8-bit binary number.
- Character Code** A one-byte number that identifies a specific character.
- Command** An instruction either programmed or executed immediately.
- Concatenation** Linking together two or more strings.
- Conditional Statement** A statement executed only if a specific condition is satisfied.
- Crash** Breakdown of a program because of program or data error.
- Cursor** Indicates where the next character or space will be printed on the screen.
- Debug** To locate and remove errors from a program.
- Disk Drive** Random access storage device that reads and writes files from magnetic floppy disks.
- Edit** To modify any program line.
- Empty Loop** A loop that serves only to slow program execution.
- Empty String** A string with no characters.
- Error Messages** Computer-generated prompts that indicate the nature and location of a program or system bug.
- Floppy Disk** A magnetic storage disk for use in a disk drive.
- Function** A specific operation performed on a number or string.

**Hardware** The mechanical components of a computer and/or peripherals.

**High-Level Language** A language the computer must translate first to machine code before executing.

**Immediate Mode** Executes any command once.

**Integer Variable** Variable that stores only integers.

**K** 1K = 1024 bytes.

**Line Number** Determines the order in which program statements are executed.

**Literal String** Characters enclosed by quotation marks.

**Load** To transfer a program from storage medium (tape or disk) to the computer's memory.

**Loop** Part of a program that repeats itself.

**Low-Level Language** Machine code.

**Machine Code** Low-level language that executes all instructions at the speed of the microprocessor chip.

**Nested Loops** Loops within loops. Nested loops are executed before outer loops.

**Numeric Array** A set of numeric variables, each identified by the identical array name and a different subscript number.

**Numeric Variable** A variable assigned to store any given number or numeric expression.

**Peripherals** Satellite hardware (printers, monitors, keyboards, cassette tape recorders, disk drives, etc.) that translates specific computer instructions into specific functions.

**Printer** Device to provide hardcopy output (paper) from the computer.

**Priority** The established order or arithmetic or logical operations.

**Program** A numbered list of computer instructions or commands.

**Random Access Memory (RAM)** Computer memory accessed by the programmer for storage of programs and data. Specific RAM bytes may be altered at will (**PEEK/POKE** statements).

**Read Only Memory (ROM)** A fixed computer memory usually containing BASIC interpreter and operating system programs. May be read by programmer but not altered.

**Relational Operators** Symbols (=, >, <, etc.) used to compare numbers, expressions, or strings.



**Save** To transfer a program from computer memory to a storage medium (tape or disk).

**Software** Computer programs that drive the hardware.

**Statement** A computer instruction written into a program.

**String Array** A set of string variables, each identified by an identical array name and different subscript number.

**String Variable** A variable assigned to store a string of characters.

**Syntax** Required and proper structure of a program line.

## INDEX

### A

**ABS** 119–120  
 Addition 12  
 Address 128  
**AND** 58  
 Arithmetic Operators 12–13  
 Arrays  
   multidimensional 76  
   numeric 73  
   string 75  
**ASC** 86  
 ASCII 84–85  
**ATN** 125  
 Attack/Decay 113

### B

**BASIC** 130  
 Binary 127  
 Bit 104  
 Branching 54–55  
**BREAK** 6  
 Bug 128  
 Byte 104

### C

**CHR\$** 85  
**CLR** 142  
**CLR/HOME** 1  
 Comma 8

Colon 9–10  
 Color 90–94  
 Command 2–3  
**COMMODORE** Key 91  
 Concatenation 20  
 Conditional Statements/  
   Relations 23  
**CONT** 7  
**COPY** 7  
**COS** 125  
 Crash/Crashproofing 128  
**CTRL** 90  
 Cursor 1  
 Cursor Control Keys 6

### D

**DATA** 47  
 Debugging 128  
**DEF FN** 124  
**DELETE** 7  
**DIM** 75  
 Division 12  
 Down Cursor Arrow 5

### E

Edit Mode 4  
 Elements  
   array 72  
   data 47  
 Empty Loop 35

Empty String 32  
**END** 8  
 Error Messages 129  
**EXP** 121

### F

Flag 50  
 Floating Point Decimal  
   Variables 14  
**FN** 124  
**FOR/NEXT** 34  
 Function Keys 122

### G

**GET** 31  
**GOSUB** 53  
**GOTO** 6–7  
 Graphics 90

### H

Hardware 129  
 High Frequency 113

### I

**IF/THEN** 24  
 Immediate Mode 2  
**INPUT** 25  
**INST/DEL** 5

- INT 41
- Integer Variables 14
- Inverse Functions 121
- Inverse Letters 65
- L**
- Languages
  - high-level 130
  - low-level 130
- LEFT\$** 65
- Left Cursor Arrow 5
- LEN** 63
- Line Number 4
- LIST** 4
- Literal String 2
- LOAD** 153
- LOG** 121
- Logical Operators 58–59
- Loops 7
  - control variables 34
  - parameters 34
- Low Frequency 113
- M**
- Machine Code 182
- Memory 127–128
- Memory-mapped 94
- MID\$** 65
- Mnemonics 22
- Multidimensional Arrays 76
- Multiplication 12
- N**
- Negation 12
- Nested Loops 36
- NEW** 18
- NEXT** 37
- NOT** 59
- Numeric Arrays 73
- Numeric Notation 14–15
- O**
- ON** 55
- OR** 58
- Operators
  - arithmetic priority 12–13
  - logical 58–59
- P**
- Parentheses 13–14
- PEEK** 88
- Peripherals 129
- PI ( $\pi$ ) 125
- Pointer 47
- POKE** 88
- PRINT** 129
- PRINT#** 138
- Printer 129
- Priority
  - arithmetic 13–14
  - logical 59
- Program Mode 3
- Punctuation 8
- Q**
- Quotation Marks 2
- R**
- Radians 125
- Raised to the power of 12
- RAM (Random Access Memory) 128
- Relational Operators 60
- READ** 47
- Register 107
- REM** 16
- RESTORE** 7
- RETURN** 2
- RIGHT\$** 66
- Right Cursor Arrow 5
- RND** 90
- ROM (Read Only Memory) 128
- Rounding Numbers 44
- RUN** 3
- RUN/STOP** 6
- S**
- SAVE** 144
- Scientific Notation 14
- Semicolon 9
- SGN** 120
- SIN** 125
- Slicing Strings 64
- Software 129
- Sound 112
- SPC** 10
- Sprites 104
- SQR** 120
- STEP** 38
- STOP** 30
- Strings
  - string arrays 75
  - string variables 19
- STR\$** 68
- Subroutines 53
- Subscript 73
- Subtraction 12
- Sustain/Release 113
- Syntax 31
- T**
- TAB** 10
- TAN** 125
- Tape 131–132
- THEN** 24
- TI\$** 70
- TO** 34
- Trigonometric Functions 124–125
- U**
- Up Cursor Arrow 5
- USR** 145
- V**
- VAL** 68
- Variables
  - floating point numeric 16–17
  - integer numeric 17
  - string 19
- Voice 113
- W**
- Waveforms 113



# EINSTEIN'S BEGINNERS' GUIDE TO THE COMMODORE 64™

- is an entertaining and informative introduction to computing, written in plain English—not computerese
- offers a wide variety of easy-to-learn practical programming techniques
- requires no prior experience with computers or programming
- provides detailed explanations of modes, print displays, mathematical operations, and data storage/retrieval
- explains how to create animation using sprite graphics
- teaches you how to use Commodore V2 BASIC instructions for your own programming applications

**HARCOURT BRACE JOVANOVIICH, PUBLISHERS**

1250 SIXTH AVENUE, SAN DIEGO, CA 92101  
111 FIFTH AVENUE, NEW YORK, NY 10003

>>> \$7.95

Cover design by Bob Silverman

ISBN 0-15-600413-5