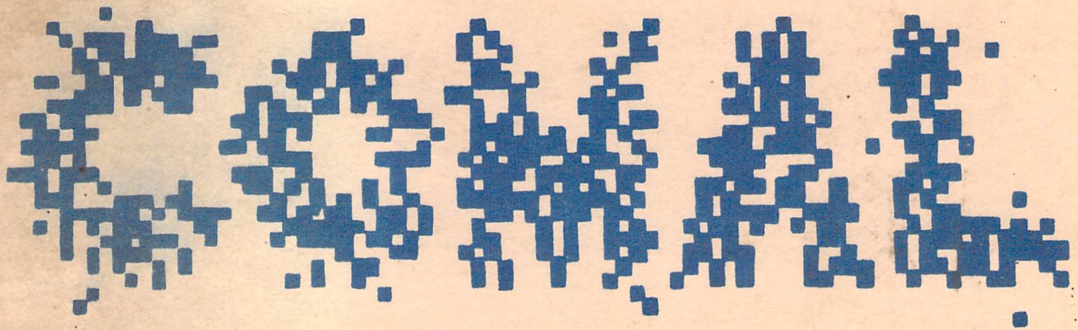
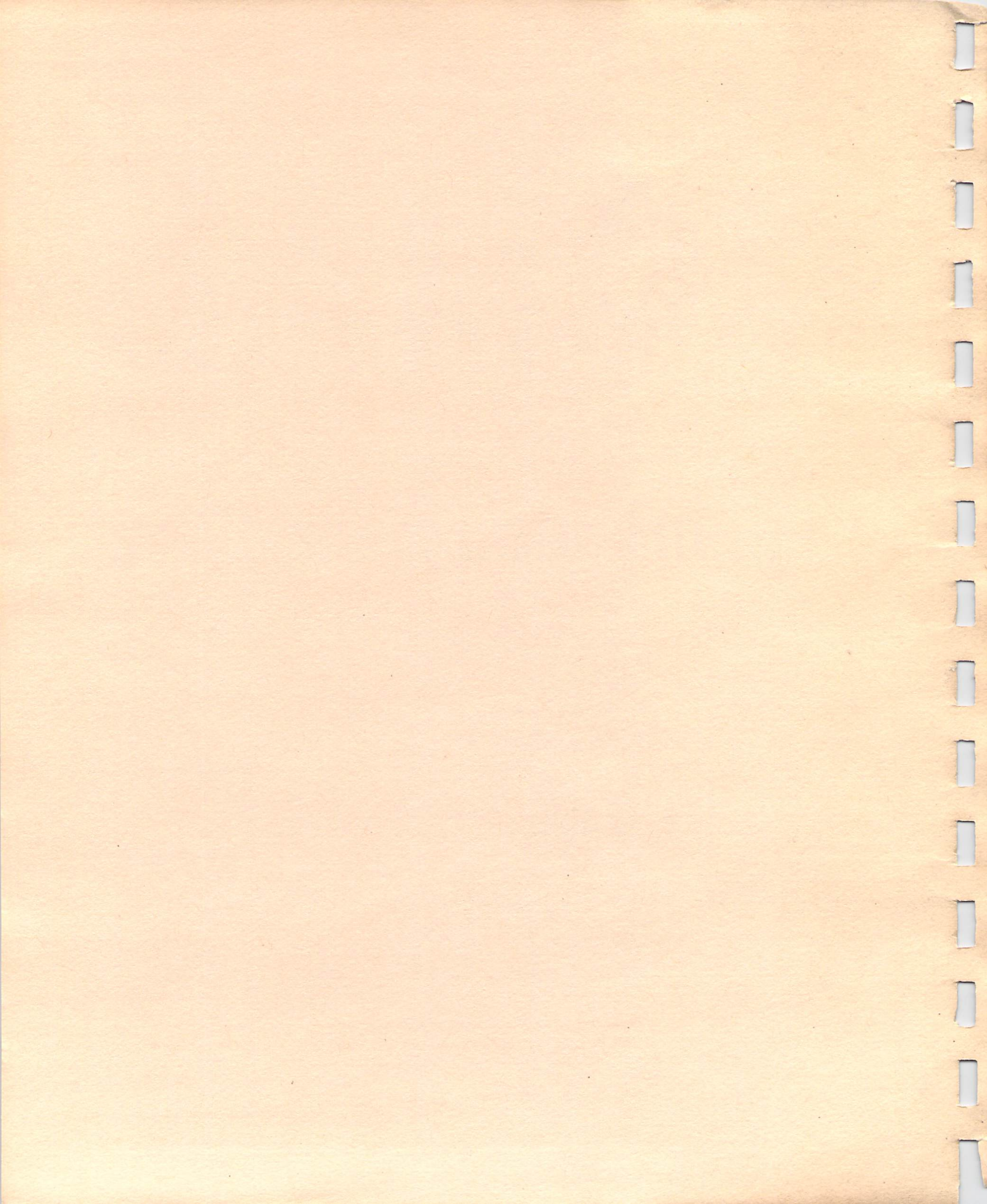


introduction to computer programming

with

comal 80 and the commodore 64/128

J. william leary



introduction to computer programming

with

COMAL 80

and the

commodore 64/128

j. william leary, ed.d.

Cover design, illustrations and drawings:

Roselyn Stewart Leary

Acknowledgements

Without the help and support of the following, this book still would be an idea to pursue "tomorrow":

Dr. C. Holland, Superintendent of Schools, Pocahontas County, W.V.; Kenneth E. Vance, Principal, Pocahontas County High School; Dan Curry, Curriculum Development, Pocahontas County Schools; Cora Lee Wyatt, Treasurer; Gwennie Friel, Secretary; and, to these students for those tedious but essential tasks of proofing, duplicating and collating, Tammy Armstrong (first draft), Mary Kay Irvine (Girl Friday, final copy), Tina Roach, Carl Seielstad, Carol Arbogast, Angie Matheny, Kathy Roach and Delana Irvine.

A special thank you to Len Lindsay, Editor, COMAL TODAY, for his supportive critique of the manuscript. (Any errors or obfuscous explanations remain the responsibility of the author.)

The author is also indebted to Robert L. Poland, Richmond High School (Indiana) --an unusually gifted teacher, for introducing him to COMAL in the Summer of 1984.

============

Printed in the United States of America. Published by:

COMAL Users Group, U.S.A., Limited
6041 Monona Drive
Madison, WI 53716

Please note the following trademarks:

Captain COMAL of COMAL Users Group, U.S.A., Limited; Commodore 64 of Commodore Electronics Limited; IBM of International Business Machines; Apple of Apple Computer;

Copyright 1985 by J. William Leary
Copyright 1986 by COMAL Users Group, U.S.A., Limited

All rights reserved. No part of this book may be reproduced in any way or by any means, without permission in writing from the publisher.

CONTENTS

****

1. introduction 1
2. mathematical operations 12
3. introduction to programming 21

4. computer management techniques 30
5. LIST and EDIT commands 39
6. introduction to $trings 46

7. more mathematical operations 54
8. more on $trings 67
9. FOR / ENDFOR loop 72

10. counting and summing 80
11. binary decisions: IF / THEN / ELIF / ELSE / ENDIF 87
12. binary numbering system 97

13. logic operators 106
14. READ - DATA 111
15. INPUT - numerics and $trings 126

16. REPEAT - UNTIL 135
17. nested loops 142
18. WHILE - DO 152

19. CASE - OF 158
20. procedures 163
21. arrays 181

22. $tring arrays 196
23. two-dimensional arrays 204
24. sorting 217

25. introduction to files 231
26. the graphic turtle 248
27. for further study 259

index 262

to goose

and each faith matheny

This book is written for the COMAL 80 cartridge.

It may be obtained from:

COMAL Users Group
USA, Limited
6041 Monona Drive
Madison, WI 53716

phone: 608 222-4432

CHAPTER ONE

introduction
****

objectives

At the completion of the chapter the student will be able to:

1. State what the acronym COMAL represents;
2. Tell what the difference is between a ROM and a RAM;
3. Place the cursor at any place on the monitor screen;
4. Show the difference between tapping the CLR HOME key alone and tapping it along with the SHIFT key;
5. Tell what the function of the RUN/STOP key is;
6. Define what a system reserved word is;
7. Use the operands +, -, /, * and ↑ in simple programs;
8. Know how to implement the AUTO function for numbering lines;
9. Know how to use RENUM to change the numbering of lines; and,
10. Write a simple program to find the area of a polygon or circle.

1.1 COMAL --COMmon Algorithmic Language-- is a powerful computer language. Though it has striking similarities with BASIC, it is an over simplification to dismiss COMAL as "just structured BASIC." The painful short comings of BASIC began to surface in the 70's and though there have been many (excellent) revisions, they remain just that --revisions.

On the other hand, to say COMAL is a pre-Pascal language is to imply it lacks the power and sophistication of that computer language. COMAL, in

every aspect, can match (and exceed, in the author's opinion) Pascal in clarity, structure and power.

At this writing (1985) COMAL has been adopted as the official school language in seven European countries, the latest being Scotland. In Denmark, COMAL is used in the operation of its airports.

1.2 THE MARVELOUS TECHNOLOGY of the computer will be left to the engineers. The user, however, should understand two primary features: 1. The ROM; and, 2. The RAM.

The ROM chip(s) cannot be changed by the user. ROM is an acronym for Read Only Memory. These chips have been programmed by the manufacturer and they are, essentially, what makes each computer distinctive from any other. ROMs are not interchangeable from one brand of computer to another.

RAMs (Random Access Memory), on the other hand, can be programmed by the user --as long as he follows the sequence of logic built into the ROM by the designer. Many RAMs are interchangeable and can be purchased from any of several electronic companies.

When the computer is turned off, what ever memory the user has "built into" the RAMS is lost. This is not true for the ROMs unless, of course, they are destroyed by a power surge or static electricity.

1.3 WHAT IS TYPED on the KEYBOARD is changed into machine language (a language based on binary numbers) by the CENTRAL PROCESSING UNIT. Once this has been accomplished, it is "operated" on by the instructions we have programmed into the RAMS. When this task has been completed, the final results are then changed back and displayed on the monitor screen or the printer.

1.4 BEFORE STARTING, THERE are some things we need to become familiar with on the C=64 keyboard.

When we turn on the computer we will note, at the end of the printed message on the screen, a flashing square light. This is called the CURSOR. By tapping certain keys, we can move it about the screen.

In the lower right of the keyboard there are two keys. One has arrows pointing up and down with the letters CRSR in the middle. The other has the same letters only the arrows point left and right. By using these keys in conjunction with a SHIFT key we can place the cursor in any position on the screen. If we wish to move the cursor DOWN or to the RIGHT we do so without employing a SHIFT key; to move the cursor up or to the left we must keep the SHIFT key depressed.

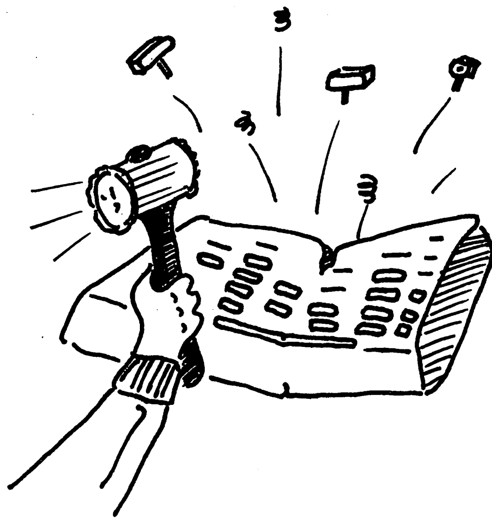
Because the CURSOR is so important, practice for several minutes moving it about the screen. Put it into the four corners, half way down one

side, then the other side, in the middle and, finally, middle top and middle bottom rows.

The cursor is the best servant we have. Commodore has designed their ROMs so that we can edit programs (using the cursor) while they are still on the screen --even to changing just one letter or one number. More on this later.

Every time we finish typing a line we must tap the RETURN key. By doing so we are 'telling' the computer, "This is what I want. Do it." It may not always agree. We may have asked it to do something it is not programmed by the ROM(s) to do; in which case, it will flash an error message on the screen before allowing us to type further. This is an outstanding strength of COMAL! And before we have finished with this book we will come to appreciate it more and more.

There are other ways of moving the cursor. If we tap the INST/DEL key the cursor will move one position to the left. If there is a character there, it will be erased (a form of editing). If we wish to move the cursor forward with the INST/DEL key, then we use it in conjunction with a SHIFT key. We can also use these two procedures to move text back and to push text forward.



Tapping the CLR/HOME key will move the cursor to the upper left hand corner of the screen, WITHOUT ERASING TEXT. Tapping this key ALONG with a SHIFT key will move the cursor to the upper left hand corner A N D will clear the screen at the same time!

The RUN/STOP key does just that. When a program is running and we wish to stop it, then tapping that key will do so. There are times when the computer "locks up" and depressing the RUN/STOP key WHILE tapping the RESTORE key will often UNLOCK it. Computers are "funny", though, and no matter what we try (short of a sledge hammar), nothing short of turning the computer off and then on seems to work in the effort to "unlock" them. This is why we should save what we are doing after every few lines.

Another important key is the long one at the bottom of the keyboard. It is the space bar. It functions the same as the space bar on a typewriter --with one nice exception, it can erase! Type a few characters on the screen and WITHOUT tapping the RETURN key, move the cursor back to the beginning of the line using the SHIFT key in conjunction with the left and right arrow CRSR key. Now, tap the space bar three times. Finally, depress the space bar and keep it there until all the letters are erased.

1.5 THE COMPUTER, THOUGH not designed to be a calculator, can be used as one and will allow some further introduction to the basics of the keyboard and COMAL.

First, an explanation of the mathematical symbols used in COMAL.

+ operand for addition

  example: 6 + 4 = 10

- operand for subtraction

  example: 6 - 4 = 2

* operand for multiplication

  example: 6 * 4 = 24

/ operand for division

  example: 6 / 4 = 1.5

↑ operand for raising to a power

  example: 6↑4 = 6*6*6*6 = 1296

< less than

  example: 4 < 6 = 4 less than 6

> greater than

  example: 6 > 4 = 6 greater than 4

<> not equal to

  example: 6 <> 4 = 6 not equal to 4

Type the following statement

```
6 + 4
```

and then tap the RETURN key.

On the screen, the CURSOR should be flashing over the "+" sign and there should be the message, "not a statement." This sentence will most likely be in lower case letters. If it isn't, depress the Commodore logo key (C=) and the SHIFT key simultaneously. Another outstanding feature of COMAL is the system-forced capitalizing of reserved words, PLUS a system-forced indentation of program lines. This increases the ease of reading, editing and debugging of programs.

What the computer needed was some instruction. All we entered was "6 + 4". The verb was missing.

If we want the computer to process the data and give an answer, then we must ask for the output. The key word is **PRINT**.

If the cursor is still flashing over the "+" sign, move it down by tapping, four or five times, the **CRSR** key --the one with the up and down arrows on it.

Tap the RETURN key.

Now type this

```
print 6 + 4
```

And tap the RETURN key.

The answer, 10, appears on the screen.

Verify the answers for 6-4; 6*4; 6/4; and, 6↑4.

All of this is very fine if the statements to be processed are not so complicated that they call for the use of more than one line; or, if we do not wish to use the results in another part of a program; or, if we do not want to save the work for another time. In other words, if all the work were of the order "6 + 4" in difficulty, then there would no need for computers. Of course,the truth lies elsewhere.

The heart of computer technology is the ability to design ALGORITHMS (programs) --within the restriction of its design, to solve problems. After writing these programs we often wish to save them, either for reference or for future use.

1.6 THOUGH COMAL INGORES line numbers when a program is RUN, it does allow them as a help or aid to the programmer. The valid line numbers in COMAL range from 1 to 9999 and can be used in any sequence as long there is successive logic in the design of the program. COMAL also has built into its ROM the characteristics of allowing the user to renumber (RENUM) the lines AT ANY TIME (!), RENUM by specified interval, and to have the computer, as it were, to provide the next line number(s) AUTOmatically after the current line is 'entered' by tapping the RETURN key.

To illustrate the latter, type the following

```
auto
```

and tap the RETURN key. The screen responds with

```
0010
```

Tap RETURN again. The screen displays

0020

Do this several times, noting the screen printout.

Tap the RUN/STOP key. Doing this will get us out of the AUTO mode.

Now, type the following

auto 1000

and tap the RETURN key. The screen shows

1000

Tap RETURN once more, and then several more times. Note the printout.

When finished, tap the RUN/STOP key.

Type this

auto 1000,3

and tap the RETURN key once. Now do it several more times noting, each time, the printout on the screen.

Try several beginning numbers and several different sequences. Observe that the default sequence is 10.

This writer prefers starting all programs at 1000. With 9000 lines still available, this is more than ample for almost any program design.

1.7 ONE FURTHER OBSERVATION of importance. Before starting a program always do two things:

1. Clear the screen; and,
2. Type NEW.

NEW is a reserved system word that clears the memory of the computer before starting another program. If WE don't type NEW, we could very well end up with an algorithm containing lines from a previous program. The end result is predictable.

===== For a few programs the notation /R/ will be used to indicate that we are to tap the RETURN key. =====

1.8 ENTER THE FOLLOWING program into the computer (do not use AUTO command)

```
clearscreen (SHIFT & CLR/HOME keys)
NEW                 /R/
```

```
1000 print 6 + 4  /R/
```

Nothing happened. And nothing was supposed to have happened. What we did, **by typing line numbers**, was to tell the computer, "Look, I'm designing a program here. Do not do anything with it until I tell you to." And how do we tell the computer to start working on the program and produce, what we trust, will be the correct result? By typing

```
RUN     /R/
```

And now we have the answer (10) to our first program.

Do this one.

```
clearscreen
NEW                /R/
1000 print 6↑4     /R/
RUN                /R/
```

The printout should be 1296.

Before typing NEW, type

```
list  /R/
```

Note that "print" is now capitalized. This is system-forced (a nice feature!) for all words reserved by COMAL in its operation. (To this point we have introduced four reserved words: NEW, RUN, PRINT; and, LIST.)

Type the following program, exactly as it appears.

```
clearscreen
NEW                 /R/
1000 width := 6     /R/
1010 print width    /R/
RUN                 /R/
```

The printout on the screen is 6.

Let's take a look at each line.

```
1000 width := 6
```

The use of " : = " has the meaning REPLACE 'WIDTH' with the value of 6. This does **not** mean that 'WIDTH' is **EQUAL** to 6; it is stating that the variable, WIDTH, is being **REPLACED** with the value of 6. The importance of this concept cannot be over emphasized.

In computer programs, variables are **dynamic**, that is, their value may change within the algorithm as many times as the programmer wishes --and without causing any problems in logic!!

The other type of variable is STATIC. Consider the following algebraic

equation.

$$2X + 4 = 15 - 3$$

The program (algorithm) to solve for the value of **X** is

$$2X + 4 = 12$$

$$2X = 12 - 4$$

$$2X = 8$$

$$X = 8/2$$

$$X = 4$$

The only value of **X** that will satisfy this equation is the value of 4. No other value will work --hence, we say the variable is STATIC, not dynamic.

However, in our computer program we specified the WIDTH to be 6. And when we asked the computer (in line 1010) to print WIDTH, it printed the value 6. If we specified (in line 1000) that WIDTH was 8 or 10.11 or -6.95 then those values would have been printed. After more experience, we will learn how to change the value of a given variable many times within a single program.

Now, to extend the logic, let's consider the formula for the area of a rectangle.

```
area := width*height
```

What we are saying is, take the variable **AREA** and REPLACE (!!) it with the value of WIDTH times HEIGHT.

One more look at this concept. Consider the classic counting loop that is used so widely in several programming languages (as we will see soon, COMAL's counting loop structure is far more logical).

```
count := count + 1
```

If we were to say "count equals count plus 1" then we find ourselves saying 1 = 2; because, in extending the logic of "count equals count plus 1", and assigning the value of 1 to COUNT, then we have

1 equals 1 + 1

or 1 = 2. Not so!!

But by stating, conceptually, "count is replaced by count plus 1", then we have 'count' taking on successive values of 1, 2, 3 and so forth as the computer 'clicks' by the statement each time.

Enter the following

```
clearscreen
NEW                                  /R/
auto 1000                            /R/

1000 width  := 7                     /R/
1010 height := 8                     /R/
1020 area   := width*height          /R/
1030 print area                      /R/

tap  RUN/STOP key                    /R/
RUN                                  /R/
```

What happened was this: The computer started the program at line 1000 after we typed "RUN." It took the value of 'width', ran down to its storage cells, labeled one 'width' and placed the value of '7' in it; it then went back, fell through to 1010 and repeated what it did on 1000, only this time it labeled a storage cell "height" and put the value '8' in it; went back, fell through to 1020, ran down to the storage cells and labeled another one "area" and then zoomed over to the cells 'width' and 'height', noted what values were in them, multiplied them together and then put that value (56) into the storage cell 'area'; it went back, fell through to 1030 --where we told the computer to print the value of area-- zoooooooommmmmeeeedd back to the storage cell 'area', saw that it had '56' in it (it has poor recall, doesn't it!) and obliged by printing the '56' on the screen.

	HEIGHT:= 8	
WIDTH:= 7		
		AREA:= 56

1.9 WHILE THE ALOGRITHM (program) is on the screen type,

```
renum 1000,5     /R/
```

type

```
LIST  /R/
```

The same lines are there, but now they have the numeration of 1000, 1005, 1010, 1015. Try several other RENUM formats. Is there an upper limit to the sequence number that we may type after the comma? If so, what is it?

RECAP
*****

ROM chips are Read Only Memory chips and cannot be erased.

RAM chips are Random Access Memory chips and may be programmed by the user. They are erased when the computer is turned off or if the user types NEW.

RUN is the command that causes the computer to process a program.

LIST will display the program on the screen.

AUTO will print, automatically, line numbers on the screen.

RUN/STOP will stop the AUTO command and also the RUNning of a program.

NEW clears the computer's memory and prepares it for a program.

RENUM is the command that renumbers the lines of an algorithm (program).

problems
****

Using the format found in 1 . 8 write programs to solve the following.

1. Find the area of a rectangle with a height of 19 and width of 20.

2. Find the area of a rectangle with height = 14.67, width = 35.67.

3. Find the area of a square with a side of 16.

4. Find the area of a square with a side of 15.008.

5. Find the area of a square with a side of 1.0809.

The area of a triangle is

area = base X height

2

In COMAL, this would read

```
area:=base*height/2
```

6. Find the area of a triangle with base = 4 and height of 6.

7. Find the area of a triangle with base = 4.02 and height = 8.

The area of a circle is

area = pi X radius X radius

In COMAL, this would read

```
area:=pi*radius↑2
```

Where pi takes on the value (for now) of 3.14.

So the formula to use is

```
area:=3.14*radius*radius
```

By the way

radius = diameter/2.

8. Find the area of a circle with a radius of 6.

9. Find the area of a circle with a radius of 10.

10. Find the area of a circle with a diameter of 40.

answers:

```
 1.   380
 2.   523.2789
 3.   256
 4.   225.240064
 5.     1.16834481
 6.    12
 7.    16.08
 8.   113.04
 9.   314
10.  1256
```

CHAPTER TWO

mathematical operations

****

objectives

At the completion of the chapter the student will be able to:

1. Take a mathematical expression and rewrite it in accordance with the proper rules, for the computer to calculate;
2. State, in the proper order, the sequence in which the computer will operate on mathematical expressions;
3. Know that in such expressions as x↑y↑z, COMAL calculates x↑y --the result being q-- and then calculates q↑z;
4. Express whole numbers in scientific notation;
5. Express decimals in scientific notation;
6. Take a number expressed in scientific notation and rewrite it as a whole number and/or a decimal; and,
7. Use parentheses, when appropriate, to combine expressions of more than one term.

```
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
c                                                               c
o  SETTING MONITOR COLORS                                       o
m  ----------------------                                       m
a                                                               a
l  THE SCREEN.  While pressing down the CTRL key, tap the "Y"   l
c    key and then any number from 1 - 8 to  select  a  color.   c
o                                                               o
m  THE BORDER.  While pressing down the CTRL key, tap the "X"   m
a    key and then any number from 1 - 8 to  select  a  color.   a
l                                                               l
c  THE CURSOR.  While pressing down the CTRL key, tapping any   c
o    number from  1 - 8  will change the color of the cursor.   o
m                                                               m
a  Pressing the CTRL key and tapping the letter "Z" will make   a
l    these choices  the  default colors should  the  computer   l
c    be reset by using the RUN/STOP and RESTORE keys.           c
o                                                               o
m  For non-color monitors,  the best combination  of  BORDER,   m
a    SCREEN and CURSOR is  to press the CTRL key and tap "W".   a
l                                                               l
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
```

2.1 **CONSIDER THE MATHEMATICAL** statement

6*5 - 2

Is the answer 30 - 2 = 28 or

6*3 = 18?

In other words, is the 2 subtracted from the 5 BEFORE multiplying, or do we multiply FIRST and THEN subtract the 2?

2.2 **THE RULES COVERING** the order in which mathematical operations are performed are very specific. In table form they are:

operation order	function performed
0	remove parentheses first
.....	.....
1	exponents (↑)
2	multiplication (*) OR division (/), in order, from left to right

3 addition (+) OR subtraction (-),
in order, from left to right

The reason for putting "remove parentheses first" as 0, is that once having located an arithmetic expression enclosed by parentheses, then the order of procedure within those parentheses follows the sequence of 1, 2 and 3.

Study these two expressions

[a] 6*5 - 2

[b] 6*(5 - 2)

First, though, observe the "*" between the 6 and (5 - 2). In algebra the expression 6(5 - 2) is understood to be six times the quantity (5 - 2). The computer, being literal, cannot read 6(5 - 2) the same way. There must be a "*" between the 6 and the "(" for the computer to understand that we are multiplying!

In [a] the computer scans the statement and seeing that there are no parentheses, proceeds to operation order 1. Finding no exponential notation (↑) it then goes to operation order 2. This tells it to multiply and divide in the order of left to right. 6*5 becomes 30. The computer stores that value.

The computer now looks for operation order 3, and finding a "-" sign, subtracts 2 from 30 and ends with the final total of 28.

In [b] the computer, finding a parentheses, handles that part of the statement first. While inside the parentheses, the computer looks for exponents (↑), multiplication (*), division (/) and then, finally, addition (+) and sutraction (-) operands. It finds subtraction only, and so it does 5 - 2 and stores the resultant of 3. Having removed the parentheses the expression now is

6*3

The answer, of course, is 6*3 = 18.

Work out the following before studying the explanations.

....

[c] 6*5-2+(4+9)*(3+2)-(5-2)-2↑3

....

step

1 6*5-2+(4+9)*(3+2)-(5-2)-8

2 6*5-2+ 13 * 5 - 3 -8

```
     3            30-2+      65     - 3 - 8

     4            28         93      90   82
```

In step 4, the computer subtracts 2 from 30 to get 28, adds 65 for 93, subtracts 3 for 90 and, finally, subtracts 8 for the answer of 82.

```
..................................

[d]  3↑2*((4-10)-2*5 + (4*3+8))-26

..................................
```

```
step

 1            3↑2*(   -6  -2*5  + (12+8)-26

 2            3↑2*(   -6  -2*5  +   20)- 26

 3            3↑2*(   -6  -10   + 20) - 26

 4            3↑2*(       4     ) - 26

 5            3↑2*4-26

 6            9*4-26

 7            36-26

 8            10 --which is the answer.
```

2.3 THE RULE OF order from left to right also applies to exponential notation.

3↑2 means 3*3 and 3*3 = 9. 2↑3 means 2*2*2 and 2*2*2 = 8. However, what about 2↑3↑2?

To make certain that the left to right order is followed, we rewrite 2↑3↑2 as

```
(2↑3)↑2  this gives--

8↑2 which equals 64.
```

Remember, in COMAL, the order of operation for expressions like this is

```
3↑2↑3 =

(3↑2)↑3 =

(3*3)↑3 =

9↑3 =
```

9*9*9 = 729

2.4 SUPPOSE WE WANTED to divide 30 by 2 + 4 --how would we write this for the computer?

Would we write

```
print 30/2+4 , or

print 30/(2+4)?
```

If we did these on the computer, we would find the first answer to be 19 and the second to be 5. As pointed out earlier, the computer is literal and has a certain order to follow. In the first expression, the computer would divide 30 by 2 and THEN add 4 to the result. In the second, the computer would have added the 2 and 4 (for 6) because IT DOES PARENTHESES FIRST(!) then it would have divided the 6 into the 30.

2.5 CALCULATE THIS ON the computer

2↑29

We should have a readout of 536870912.

And this

2↑30

The readout is now 1.07374182e+09.

If we multiplied the answer by hand of 2↑29 (536870912) by 2 --to get 2↑30, it would be

1073741824

instead of

1.07374182e+09

As we can see, the numerals DO match --except that the 4 on the end is missing.

Beyond a certain length (and count to see what that is) the computer will give numeric answers in exponential notation. In this instance the "e" stands for a base of 10; the "+09" means to find the decimal and count 9 places to the right --and add 0's if we run out of numerals. So

1.07374182e+.09

would give

1073741820 as the answer to 2↑30.

We know the actual answer is 1073741824, so the best the computer can do is approximate answers beyond 9 digits. Considering the size of the computers we are working with, this is a staggering accomplishment! In this calculation, the answer is only off 4 units in 1 billion 73 million plus!!! Let's not quibble!

=== Do this one on the computer

    2↑(-6)

We need the parentheses around the "-6" because the computer, remember, does things literally. The expression 2↑-6 would make no sense to it.

The answer should be

    .015625

By the way, 2↑(-6) means 1/2↑6 or 1/(2*2*2*2*2*2) or 1/64.

=== 1/128 = 1/2↑7 = 1/(2*2*2*2*2*2*2) = 2↑(-7). Do that on the computer. Here we get an answer that reads

    7.8125e-03

"e", again, stands for base 10. The "-03" means to start at the decimal and count three places TO THE LEFT(!), placing 0's where needed. So the final answer here is

    .0078125

If we were to do this out by hand, the answer would be .0078125 also!

=== Express 88456900211 in scientific notation.

The first thing to do is to rewrite this as

    8.8456900211

then count the number of digits after the decimal point --which, in this instance, is 10. The final answer would be

    8.8456900211e+10

=== Express .000308551 in scientific notation.

Again, the first thing to do is to rewrite this as

    0003.08551

(Note that the decimal goes after the first non-zero digit!) Next, count the number of digits it takes to get back to the beginning of the number --in this case it is 4. The final answer would be

    3.08551e-04

=== Rewrite 890.05673 in scientific notation.

Rewriting and moving the decimal --

8.9005673

But this time, the decimal is only TWO places from the 8, so the final answer is

8.9005673e+02

=== Write a statement for the computer to calculate the answer to

```
6 + 4     8(5 - 2)       27
----- - --------- + ------
  5       2 + 6       7 + 2
```

Please observe, carefully, how parentheses are used so that in the first fraction, the ENTIRE numerator is divided by 5; that 8 and not JUST the 2 divides into 8(5 - 2) in the second fraction; and, for the third fraction, 27 is divided by 9 and not JUST the 7!.

A first step

```
(6 + 4)/5 - 8*(5 - 2)/(2 + 6) + 27/(7 + 2)
```

The second step calls for putting PRINT before the expression

```
print (6 + 4)/5 - 8*(5 - 2)/(2 + 6) + 27/(7 + 2)
```

The answer is 2.

RECAP
*****

EXPRESSIONS inside of parentheses are handled first by the computer.

↑, *, /, + and - are evaluated in mathematical expressions --and in that order.

"e" represents the base 10.

THE number following "e" is the exponent.

problems
****

First, solve the following without the aid of the computer.

Then do them on the computer (do not combine the numbers, type them in as they appear). If the answers do not agree, check the work and then the way they were entered into the computer.

1. $3 \uparrow 2 \uparrow 2$

2. $4(3 + 4)$

3. $$\frac{6(2 + 1)}{9}$$

4. $6(3) - 2(4 + 1) + 3 \uparrow 2$

5. $$\frac{6(3)}{9} - \frac{2(4) + 2}{10} + \frac{4 \uparrow 3}{2 \uparrow 3}$$

6. $$\frac{17 + 7}{3 + 5} - \frac{3 + 9}{2 + 4} - \frac{2 \uparrow 5}{4 \uparrow 2}$$

7. $$\frac{2(6(3+1))}{8} - \frac{14(3-1)}{2 \uparrow 2} + \frac{6 \uparrow 3}{9}$$

8. $4((3)(5)) + (2 \uparrow 3)(3-1) - 6$

9. $$\frac{3((4 - 1)(18 - 5))}{(10 - 7)(6 + 7)} - \frac{(2 \uparrow 4)(6)(3 - 1)}{3 \uparrow 2 + 3}$$

10. $$\frac{7(4 + 9)(6 - 1)}{13(5)} - \frac{3(2)(6)}{2 \uparrow 4 + 2} - \frac{15}{5} \uparrow 4$$

11. Express in scientific notation 86711100053

12. Express in scientific notation 4967.3569013

13. Express in scientific notation .0168531

14. Rewrite 3.194624701e+.04

15. Rewrite 4.00561e-05

answers

1.	81	2.	28	3.	2	4.	17	5.	9
6.	-1	7.	23	8.	70	9.	-13	10.	16

11. 8.6711100053e+10 12. 4.9673569013e+03

13. 1.6831e-02 14. 31946.24701

15. .0000400561

CHAPTER THREE

introduction to programming

****

objectives

At the completion of the chapter the student will be able to:

1. Write a program statement with the unknown variable on the left;
2. Assign values to variables within a program paradigm;
3. Use // to write documentation within a program;
4. Insert lines within a program that has already been written;
5. Use the function DEL to remove a line or lines from a program;
6. Edit (on the screen) existing programs with the CURSOR keys; and,
7. Translate arithmetic formulas into algorithms understood by the computer.

```
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
c                                                               c
o  THE VALUE OF PI                                              o
m  ---------------                                              m
a                                                               a
l  There are many features built into  the COMAL 80 cartridge.  l
c                                                               c
o  One of them is the value of pi.  Instead of having  to sup-  o
m    ply 3.14, 22/7  or 3.1416  --to list some  of  the common  m
a    ones, COMAL has programmed it as the constant 3.14159266.  a
l                                                               l
c  Therefore, in problems with pi as part of a formula,  COMAL  c
o    will substitute that value  in  its  place automatically.  o
m                                                               m
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
```

3.1 DESIGNING A SOLUTION to a problem and being able to program a computer to accept it and produce a solution, or a series of possible solutions for varying sets of parameters, is the essential strength of modern computer technology. In this chapter we shall introduce some elementary techniques of programming, using the information we already know.

3.2 EVERY PROGRAM HAS variables. When we were solving for the area of a rectangle in chapter one, we used the formula,

```
area = width*height
```

That particular formula has three variables: area; width; and, height. If we know any two, we can solve for the third.

In COMAL, variable names do not have to be abbreviated to a single letter, two letters, or a letter and a number; instead, we may have up to 78 characters as the name of a variable. This is very helpful. If we can use the full name of the variable, it will assist us in following the program design, as well as helping in locating errors.

In an equation, variables exist on both sides. It is essential in computer programming that the unknown variable be ON THE LEFT SIDE of the equation. For example, if we know the HEIGHT and the WIDTH of a rectangle and want to find the AREA, then this statement **is not valid**

```
1000 width*height:=area
```

The unknown, area, is on the right side of the equation. The computer is not designed to handle this, even though we could on paper.

The proper way to write the statement is

```
1000 area:=width*height
```

If the width and area are known, the height could be found by writing

```
1000 height:=area/width
```

3.3 IN PROGRAMMING, THERE are rules we must follow. Most are an extension of just plain common sense. For example, if we are going to use the formula for a rectangle to find its area, then we must first tell the computer what the values are for the width and the height **before** we ask it to compute the area. Certainly, this is logical.

Let's assign the value of 6 to the width and 13 to the height. We now have

```
width = 6

height = 13
```

We know area = width*height. We want the computer to multiply the width times the height and give us an answer for the area. We are now in a position to write our program.

First, type

```
auto 1000   /R/
```

After we tap RETURN, 1000 appears on the screen with the cursor flashing two spaces to the right of the last "0."

Before the computer can calculate the area it must know the values for the width and the height. So, on line 1000 type

```
1000 width:=6  /R/
```

After tapping RETURN, the next program line number --1010-- appears. Type

```
1010 height:=13  /R/
```

Now that the computer has been told what the values are, it can calculate the area. So, on line 1020, enter

```
1020 area:=width*height  /R/
```

To get the value for area printed on the screen, we must use the verb "print"; therefore, on 1030 type

```
1030 print area
```

The final program now looks like this

```
1000 width:=6
1010 height:=13
1020 area:=width*height
1030 print area
```

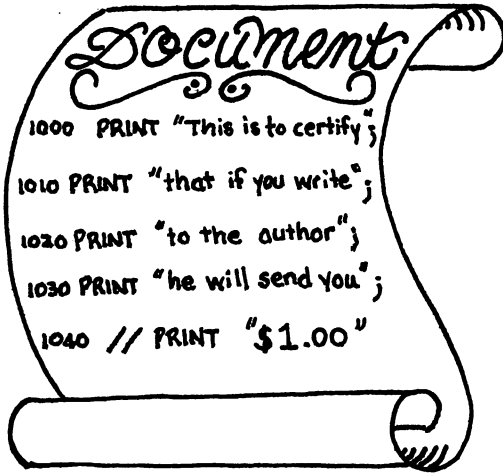
After tapping the RUN/STOP key to end the auto mode, type RUN and tap RETURN. The result (78) will be printed on the screen.

3.4 IF SOMEONE WALKED into the room and looked at the program, what meaning would it have?

Very little, actually. What does the number represent?

All programs should be documented (!!) --and do not be stingy on documentation. Time has a way of making things more and more hazy. To be sure, this is an elementary program and remembering what it was about

would not be difficult. But as programs become more and more involved, the details of what we were designing in any given section have a way of slipping by.



COMAL provides a nice way for the user to put in remarks --or notes, so that they do not interfere with the actual program. By using two slashes // next to a line number, we can enter a note or remark. When the computer is running the program, it will ignore any lines that begin with //.

First, clear the screen by tapping the SHIFT and CLR HOME keys simultaneously. Then display the program by typing LIST.

Generally, remarks are put at the beginning of a program or at the beginning of sub-sequences within a program (as they get longer and longer). Because 1000 is the first line, we will need to put it before that. To do this type 990. BEFORE tapping the RETURN key, complete the line as follows:

```
990 // program to find the area of a rectangle  /R/
```

Now, LIST the program again. We will see that the computer has put in 990 before line 1000. We can always insert lines like this. If we have left something out, all we have to do is type a number just before the line we want it to precede, and the computer will do the rest! This is why we allow the default value of 10 between each line. And, remember(!), in COMAL we can RENUM the lines at any time!! --and with any sequence.

Let's do just that. Type

```
renum 1000,1  /R/

LIST. (Note the line numbers.)
```

Type

```
renum 1000,5   /R/
```

Then

```
LIST
```

again.

The program now reads

```
1000 // finding area of rectangle
1005 width:=6
1010 height:=13
1015 area:=width*height
1020 PRINT area
```

Suppose, we wanted to do away with (delete) line 1005. How would we do it? Again, in COMAL, this is very simple. Type

```
del 1005   /R/
```

LIST the program and we see that line 1005 is gone. Now, put it back once more by typing

```
1005 width:=6 /R/
```

What if the width was 7, instead of 6. That correction is also made quite easily. Using the CURSOR keys along with the SHIFT key, move the cursor so that it is directly over the "6" on line 1005. Type

```
7     /R/
```

It's that simple! RUN the program again and now the answer is 91.

We can replace as many characters as we wish by using the cursor keys. This is called on-screen editing, a very useful tool that Commodore has built into its computers.

The problems at the end of this chapter are quite simple. Use them as an opportunity to practice LISTing, DELeting, inserting lines and changing values by using the cursor keys. Don't think the technique has been mastered until these routines have been done dozens of times. We will use these skills throughout the rest of the book --time and time again!

3.5 **DELETING IS NOT** restricted to DELeting just one line at a time. If, in the program on the screen, we wished to delete lines 1005, 1010 and 1015, we would type

```
DEL 1005 - 1015  /R/
```

3.6 **HERE ARE TWO** more examples to demonstrate how to write simple programs.

(Don't forget to clearscreen /R/, and type NEW /R/ before copying each of the following programs!)

=== In a triangle we are given the length of the base as 4 and the area as 25. Write a program to find the height.

First, the formula for the area of a triangle is

```
area = base*height
       -----------
            2
```

We must get the height on the left side of the equation. Simple algebra gives us the next step.

```
2*area = base*height
```

Switching sides, we have

```
base*height = 2*area
```

Dividing both sides by BASE gives

```
base*height = 2*area
-----------   ------
base            base
```

Cancelling "base" on the left side of the equation, now gives

```
height = 2*area
         ------
          base
```

We now can write our program. First, we <clearscreen>, and then type

```
new                                              /R/
```

then

```
auto 1000                                        /R/

1000 // program to find height of triangle       /R/
1010 area:=25                                    /R/
1020 base:=4                                     /R/
1030 height:=2*area/base                         /R/
1040 print height                                /R/

     (the answer: 12.5)
```

=== Write a program to find the radius of a circle whose area is 189.

(Don't forget CLEARSCREEN, NEW and AUTO 1000.)

The formula for the area of a circle is

```
area = pi*(radius)↑2

radius↑2 = area/pi
```

To find the radius only, we must take the square root of both sides, so we now have

```
--------         --------
radius↑2   =     (area/pi


radius   =   (area/pi)↑.5


1000 // program to find radius of circle   /R/
1010 area:=189                             /R/
1020 radius:=(area/pi)↑.5                  /R/
1030 print radius                          /R/

     (answer: 7.75632443)
```

RECAP
*****

ONLY one variable can be on the left side of an equation in a computer statement.

VARIABLES can be assigned values within the program.

THE double "//" is used to enter comments in a program.

DEL is a system word that is used to delete lines or a line from a program.

ON the screen editing is accomplished by use of the cursor key.

problems
****

Write programs to solve the following. Be certain to include a statement of documentation in each (//).

1. Rectangle. Width = 6; height = 12. Find area.
2. Rectangle. Area = 48; width = 6. Find height.

3. Triangle. Base = 4; height = 7. Find area.

4. Triangle. Area = 12; height = 9. Find base.

5. Circle. Radius = 10. Find area.

6. Circle. Area = 74. Find radius.

7. Circle. Area = 212. Find diameter.

8. Circle. Radius = 10. Find the circumference.

The formula for the circumference of a circle is

```
circumference = 2*pi*radius
```

9. Circle. Area = 97.08. Find the circumference.

From a previous discussion, we know the radius of a circle, given the area, is

```
radius = (area/pi)↑.5
```

if

```
circumference = 2*pi*radius
```

then, substituting, we have

```
circumference = 2*pi*((area/pi)↑.5)
```

10. Circle. Circumference = 42.34. Find the radius.

11. Circle. Circumference = 56.038. Find the area.

The formula for the area of a trapezoid is

```
area = height*(base1 + base2)
       ----------------------
                  2
```

12. Trapezoid. base1 = 10; base2 = 20; height = 6. Find area.

13. Trapezoid. area = 166; height = 14. Find the sum of the two bases.

[Hint: let

sumbases = (base1 + base2)

and solve for sumbases.]

14. Trapezoid. area = 231; height = 19; base1 = 4. Find base2. [Hint: put in the givens and work out formula by hand first.]

The algebraic formulas to convert temperatures from one measurement to another are

$$\text{celsius} = \frac{5}{9}(\text{fahrenheit} - 32)$$

and

$$\text{fahrenheit} = \frac{9}{5}(\text{celsius}) + 32$$

15. In celsius, what is 77 degrees fahrenheit?

16. In fahrenheit, what is -40 degrees celsius?

answers:

1.	72	9.	34.9276862
2.	8	10.	6.73862029
3.	14	11.	249.893747
4.	2.66666667	12.	90
5.	314.159216	13.	23.7142857
6.	4.85334231	14.	20.3157895
7.	16.4294487	15.	25
8.	62.8318531	16.	-40

CHAPTER FOUR

computer management techniques

****

objectives

At the completion of the chapter the student will be able to:

1. Format a disc by using the command

   pass "n0:<identifier>,86"

2. Store files on a disc as a program with

   save "<file name>"

3. List the files on a disc to the monitor by typing

   cat

4. Over-write a previous program stored on a disc by

   save "@0:<file name>"

5. Take a program off a disc and put it into the computer with

   load "<file name>"

6. Take a program off a disc and have the computer RUN it with

   chain "<file name>"

7. Run a program on a printer or get a copy of a program with

   select output "lp:"

----

comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal

RENUM FUNCTION KEY

Another method built into the COMAL 80 cartridge for renumbering lines in an alogorithm (program) is defined by the **f1** key.

By tapping the **f1** key and then the RETURN key, all of the lines in the program will be renumbered automatically.

The lines will start with

```
0010
0020
et cetera
```

The USER must employ the command specified in chapter three to get any other line number and/or sequence interval.

comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal

4.1 **EACH TIME WE** completed one of the problems at the end of Chapter Three, we had to type NEW before starting the next. Once we typed NEW, the program was erased from the computer's memory and there was no way to recall it. There is a way to SAVE programs so they can be looked at later --or, can be recalled to be MERGEd as part of a future program.

The fastest method of saving and recalling programs is to put them on a floppy disc. But before we can do that the disc must be prepared --"made ready." This is called FORMATTING a disc.

Turn on the disc drive and carefully(!) insert a floppy disc. Close the cover. Type the following command putting some identification name in place of the words FILE NAME --do not include the symbols --"<" ">"-- found immediately before and after the words "file name." Use any number you wish in place of the "86"; just be certain it contains two digits. (Leave no spaces inside the quotation marks.)

```
pass "n0:<file name>,86"  /R/
```

Wait a few seconds until the drive has stopped running, the red light is out and the green one comes on.

To view what is on the disc, type

```
cat      /R/
```

(From now on, the reminder to tap RETURN --/R/-- will be left out. Every time a line is completed, always finish it by tapping the RETURN key.)

CAT is an abbreviation of "catalogue." Typing CAT will automatically

recall and list, on the screen, all programs that are on the disc. If we wish to get a hard copy (a list printed on paper) there is a command for that. It will be covered later in the chapter.

When we typed "cat" the only listing on the screen was the identification of the disc. Because we have not saved any programs, there was a statement that read

"664 blocks free."

This means we have 664 blocks of space left on the disc to store programs. Each program uses up blocks of space; and the longer the program, the more blocks required.

If this is the first computer language studied, another outstanding characteristic of COMAL cannot be fully appreciated.

> No matter where we are in developing (writing) a program, we can call for a CAT listing **WITHOUT ERASING** the program we are working on!!!

Never forget, COMAL was written for the programmer!! And it's these seemingly small features that make it such an outstanding language.

PASS is a command that among other things, formats a disc. After formatting, a disc can accept and store programs. **PASS** may also be used in VALIDATING and DUPLICATING discs, as well as RENAMING or COPYING FILES. In this introductory text, we shall restrict our discussion of PASS to FORMATTING and VALIDATING discs. (For a further discussion on PASS, see Len Lindsay's, THE COMAL HANDBOOK.)

4.2 TO DEMONSTRATE HOW to store programs on a disc, enter the following.

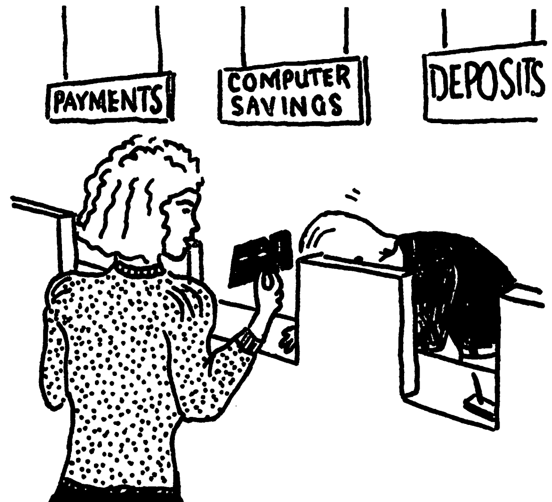
```
<clearscreen>
new
auto 1000

1000 // program to find base2 of a trapezoid
1010
1020 area:=231
1030 height:=19
1040 base1:=4
1050
1060 // formula for base2 of a trapezoid
1070
1080 base2:=2*area/height - base1
1090
1100 print base2
```

RUN the program to make certain it works. The answer should be 20.3157895.

There are two basic methods for storing programs on the disc. The first

uses the command SAVE; the second uses the command LIST. Each serves a different purpose. For this program we will use SAVE. (LIST will be covered in a later chapter.)



Before saving a program, we need to give it a name. Again, common sense should prevail. We should stay away from esoteric or obscure codes and names (unless, of course, we have written a program that will change lead to gold). Since we are dealing with trapezoids, let's give the program that name. (When storing programs relating to the problems at the end of each chapter, it would seem logical to use names such as, ch1-pr1; ch9-pr11; and so forth.)

Type

```
save "trapezoidbase"
```

After the computer has stopped running, type

```
CAT
```

(Don't type NEW!.) The screen should read (after the heading)

```
1   "trapezoidbase"   prg
663 blocks left
```

This is our first program to disc. It took up one block of space, and there are 663 blocks left for storage.

Now, type

```
LIST
```

And we will see that our program has not been erased!

Type

```
NEW
```

This has erased the program from the memory of the computer. Type

```
LIST
```

and see that this is so. Clear the screen by pressing the SHIFT and CLR/HOME keys simultaneously.

4.3 THE NEXT STEP

4.3 THE NEXT STEP is to retrieve the program from the disc. There are four commands we can use to do this: LOAD; CHAIN; RUN; and, MERGE. We will deal with the first three

for now.

When calling for a program on a disc, we must type it exactly as it appears in the CATalogue. If it is misspelled, then we must do the same; if has a space or two inside the name, we must do the same; and, if it uses symbols --such as punctuation-- we must do the same. If we are not sure how the program was stored, we can type

```
CAT
```

and have it listed on the screen. If there are too many program names to fit on the screen and they are going by too fast to read, tap the SPACE BAR. This will stop the listing. To continue the listing, tap the SPACE BAR again. Check the spelling and type the command.

To retrieve the program on trapezoids, type

```
load "trapezoidbase"
```

When the flashing cursor appears on the monitor, type

```
LIST
```

And there is our program! We can RUN it if we wish or even make changes by typing in new lines, or by using the on-screen editing feature of the cursor.

Suppose, for that matter, we do wish to alter the program by changing the area to 619.5. Must it be saved under another name? We could if we wished to. But we can save in under the same name too.

First, use the cursor controls and go to line 1020 and change the 231 to 619.5. Tap the RETURN key, use the down CRSR key to get to the bottom of the listing, and type

```
RUN
```

The answer should be 61.2105263.

To store the new program with the same name as the old, type

```
save "@0:trapezoidbase"
```

The "@" is next to the letter "P" and the "0" is a zero. This command over-writes the old program with any changes we have made or lines we might have added or deleted.

There is another advantage to this command too. Suppose we must interrupt our work before finishing a program. We can save the program as it is; then when we come back and have the opportunity to finish, this command cam be used again --and as many times as we want, for that matter-- to update our previous work.

There may be times when we wish to take a program from the disc and run

it without taking the time to type in the word RUN. COMAL has added this advantage too.

First, type

NEW

to clear the present TRAPEZOIDBASE program from the computer's memory. Clear the screen. Type LIST to make certain the memory is clear.

Two commands to take a program from the disc and have it RUN automatically, are

CHAIN and RUN

Type either of the following statements

```
chain "trapezoidbase"

run "trapezoidbase"
```

The program is not listed but the answer is --61.2105263.

The computer, recognizing the command CHAIN or the command RUN, searched the disc, loaded the program into its memory and ran it. It then printed the answer on the screen. If we wish to see the program lines we can do so by typing LIST.

4.4 UP UNTIL NOW, before we ran a program, we had to clear the screen first. We did this by tapping the CLR/HOME key while pressing the SHIFT key.

COMAL has two commands which can be incorporated into programs to do this for us. Here, we shall introduce one of them. It is

PAGE

If PAGE is output to the screen, it will clear the screen. By putting PAGE on the first line of an algorithm (program), the screen will be cleared before anything else is done.

For the program on the trapezoidbase, type

```
990 page
```

or, if using the **f1** key for RENUMbering, type

```
1 page
```

then, either type

renum 1000 or, tap **f1**

LIST

to make certain everything is o.k. Then type

```
RUN
```

The program will be executed --and note that the screen is cleared first, before the answer is printed.

PAGE can be used as many times as wanted in a program. Any time we wish to clear the screen, we can incorporate the command PAGE in the algorithm design.

Because we have added a line to the program, we need to reSAVE the program by typing

```
save "@0:trapezoidbase"
```

4.5

THERE IS ONE more output statement we should consider. The output to a printer --called, getting a "hard copy."

The first thing, of course, is that we must be connected to a printer via the serial port on the back of the disc drive. Another factor to keep in mind is that some printers will "lock up" if there is a PAGE command in the program (others will execute a form feed --or page eject). If your printer does either, then delete PAGE or edit the line by putting "//" before the command.

Now, having taken care of that and with the program listed on the screen, type

```
990 select output "lp:"
```

"lp" stands for **L**ine **P**rinter. RENUM the program by typing

```
RENUM 1000
```

If connected to a printer, RUN this program and get a "hard-copy" answer to the problem.

Another advantage of hard copies is the assistance it provides in debugging programs for errors. When we want to find errors, we often want just a copy of the program rather than a printout of the results. One method of getting this is to type

```
LIST "LP:"
```

without a line number. That is, type the printer command outside of the program proper.

When the RETURN key is tapped, the printer will type a copy of the progam. If the program has been RUN, typing LIST immediately after, will also give a printed copy of the program. Once a program has been RUN and LISTed, an output to the screen can be had by typing RUN again.

To get a printout of the directory (CATALOGUE of programs SAVEd on the disc), type

```
select output "lp:"
cat
```

(Be certain there is no current program in the computer before doing it.)

RECAP
*****

COMMAND	WHAT IT DOES
pass "n0:<file name>,86"	to format a disc to get it ready to accept and save files
cat	lists file names on a disc.
save "<file name>"	stores files on a disc as a prg.
save "@0:<file name>"	over-writes a previous prg stored on the disc
load "<file name>"	takes prg file off disc and puts it into the computer
chain "<file name>" and run "<file name>"	takes prg file off the disc and puts it into the computer and then RUNs it
select output "lp:"	chooses the printer as the output location
page	clears the screen

problems
****

Enter the following program

```
1000 page
1010
1020 // program to find volume of a rectangular solid
1030
1040 width:=6.453
1050 length:=10.157
1060 height:=13.166
1070 volume:=width*length*height
1080 print volume
```

1. RUN the program.

2. Use the cursor to change LENGTH in line 1050 to 8.979. RUN the program.

3. DELete line 1020.

   DELete line 1060, 1070 and 1080 with one command.

   Type

   ```
   1011 // program to find height of rectangular solid

   1051 volume:=629.84

   1052 height:=volume/(width*length)

   1053 print height
   ```

   RENUMber the program to start at 2100 with each succeeding line increased by 3. (Last line should be 2124.)

   RUN the program.

4. Replace line 2100 with a print command to

   RUN the program on a printer.

5. LIST the program on a printer.

6. RUN and LIST problem 7, chapter 3 on a printer.

7. Make a hard copy of problem 10, chapter 3.

8. Make a hard copy of problem 13, chapter 3.

9. Make a hard copy of problem 14, chapter 3.

10. Make a hard copy of problem 16, chapter 3.

answers:

```
1.  862.940731
2.  762.857618
3.   10.8702768
```

CHAPTER FIVE

list and edit commands

****

objectives

At the completion of the chapter the student will be able to:

1. Use the command LIST to bring up on the screen:

   1.1 All lines in a program;

   1.2 Any single line in a program;

   1.3 A series of lines beginning with a given line number and ending with another line number;

   1.4 All lines from the beginning of a program and ending with another specified line; and,

   1.5 All lines beginning with a specified line through to the last line of a program.

2. Use the command EDIT to bring up on the screen:

   2.1 All lines in a program, one at a time;

   2.2 Any single line in a program;

   2.3 A series of lines, one at a time, beginning with a given line number and ending with another specified line;

   2.4 All lines, one at a time, from the beginning of a program and ending with another specified line; and,

   2.5 All lines, one at a time, beginning with a specified line through to the last line of a program.

3. State the advantages of EDIT over LIST when debugging or correcting programs.

```
comalcomalcomalcomalcomalcomalcomalcomalcomalcomal
c                                                c
o  BLANK LINES IN AN ALGORITHM                   o
m  ---------------------------                   m
a                                                a
l  COMAL allows the use of blank lines.  It fa-  l
c    cilitates tracing programs. To be sure, it  c
o    is  only  "cosmetic"  as far as the actual  o
m    program is concerned;  but we must not ig-  m
a    nore  our  obligation to the USER.  To il-  a
l    lustrate:                                   l
c              1000 page                         c
o              1010                              o
m              1020 // an illustration           m
a                                                a
l  HARD COPY TO A PRINTER                        l
c  ----------------------                        c
o                                                o
m  Another way to get  a  printed  copy of what  m
a    is ACTUALLY ON THE SCREEN(!)  --whether it  a
l    be the results from  RUNning a program, or  l
c    the program itself, is to                   c
o                                                o
m              press down the CTRL key           m
a              and tap the letter "P".           a
l                                                l
comalcomalcomalcomalcomalcomalcomalcomalcomalcomal
```

5.1 BY NOW, AS with all beginners, we have made error after error after error in typing. Using the CURSOR key along with the SHIFT keys to correct these errors is generally acceptable for short programs, but it becomes more and more unwieldly as programs become 24 lines or longer in length.

In this chapter we shall discuss two COMAL functions that will assist us in debugging and making corrections in our programs.

Before we begin, enter the following algorithm (program). There is a mathematical error in 1130, but type that line just as it appears. (Don't forget NEW, LIST, AUTO 1000.)

```
1000 page
1010
1020 // program to find area and perimeter of rectangle
1030
1040 width:=10
1050 height:=8
1060
1070 // formula for perimeter
1080
1090 perimeter:=2*width + 2*height
1100
```

```
1110 // formula for area
1120
1130 area:=width/height
1140
1150 print perimeter
1160
1170 print area
```

RUN the program to be certain it is operating. The printout should be 36 and 1.25.

There are many reasons to call up on the screen a certain line or lines. Perhaps the program RUNs and though there are no mechanical program errors, the output just "doesn't make sense." For example, in the above program, the AREA is 1.25. Now, common sense tells us there is something "wrong" --the answer doesn't "jive" with the result we have for the perimeter. Still, the program ran and there were no error messages.

Perhaps we made a typing error in the formula for the AREA. We need to check what we did.

One way to see that particular line is to LIST the whole program. This would work as long as the program is 24 lines or less in length because it would "fit" entirely on the screen. But if it longer than that then we will have to tap the RUN/STOP key when the line we want scrolls on to the monitor --hardly an efficient manner to get what we would like!

Even with the program off the screen (as ours is because of the PAGE command) we could type (and please do so)

```
LIST 1130
```

Immediately, the following line appears

```
1130 area:=width/height
```

The error in logic is obvious. We have DIVIDED height into width instead of MULTIPLYING the two together. The computer did what WE told it to but we didn't want the computer to divide, we wanted it to MULTIPLY. The error is ours!

We can take the cursor to the "/", type "*", tap RETURN --and the correction is made. RUNning the program again gives the correct area --80.

Still, we might argue, how would we know that the area was defined on line 1130! The truth is --and particulary so, as programs become longer and longer-- we don't.

There are several ways to handle this. We could have typed

```
list 1100 - 1150
```

if we had a "feeling" the line we were looking for was in that range.

(In this case it was.) The computer would list on the screen

```
1100
1110 formula for area
1120
1130 area := width/height
1140
1150 print perimeter
```

We could still make the correction, with the help of the CURSOR key, on line 1130.

If we had typed

```
list -1100
```

the computer would put on the screen all lines FROM THE BEGINNING of the program which in this instance is 1000, through to line 1100.

On the other hand, if we had typed

```
list 1100-
```

the computer would list on the screen all lines from 1100 THROUGH TO THE END of the program which, in this algorithm, would have been from 1100 through 1170.

5.2 COMAL HAS PROVIDED another convenience for the programmer that is even better than LIST when corrections are to be made. It is the EDIT command.

If, to follow the above discussion on LIST, we want to look at and possibly correct line 1130 we would type

```
edit 1130
```

This time when the line is listed on the screen, instead of the CURSOR being **below** the line, it is **over** the first character AFTER the line number. All we have to do is move the CURSOR through the line, make the correction, tap RETURN, --and the work is done!

If we are not certain where the line for the AREA is located we can type

```
edit 1100-1150
```

1100 will appear on the screen with the cursor one space past the last digit in the line number.

This line has no error, so we tap RETURN and 1110 appears(!!). No error again, tap RETURN. 1120, no error, tap RETURN. (Remember, we fixed the error the first time around when we covered the LIST command. If we hadn't, we could now move the cursor over on 1130 and make the correction.) Tap RETURN (no error on 1140) and RETURN once more to get 1150. One more RETURN takes us out of the sequence. Certainly, EDIT is

a great time saver and a very convenient command for the programmer!

EDIT can also be used the same way as LIST.

Enter the following

```
edit
```

This will start us at the first line of our program and each time we tap RETURN, the next line will appear. We can do the whole program this way. If we wish to stop before then, we can do so by tapping the RUN/STOP key.

```
edit -1100
```

This command will list the lines, one at a time after each RETURN, starting at the first line of the program through 1100.

```
edit 1100-
```

This command will list the lines, one at a time after each RETURN, starting at 1100 through the last line of the program.

RECAP

*****

COMMAND	WHAT IT DOES
list	lists all lines in a program; if the program is longer than 24 lines, the lines will scroll on and then off the screen.
list 1130	will list only one line --and, in this particular command-- just line 1130.
list 100-200	will list all lines beginning at 100 and ending at 200.
list -900	will list all lines starting at 1, or starting at the beginning line of the actual program, if the first number is greater than 1 --through line 900.
list 900-	will list all lines starting at 900 through 9999 -or through to the end of the program, if its ending line is less than 9999.

RECAP
*****

COMMAND	WHAT IT DOES
edit	lists all lines, one at a time after each RETURN, and places the CURSOR at the first position in the line.
edit 1130	will list only one line --and, in this command-- just line 1130, with the CURSOR at the first position in the line.
edit 100-200	will list all lines, one at a time, beginning with 100 and ending at 200; again, the CURSOR will be at the first position in each line as it appears
edit -900	will list all lines, one at a time, starting at the beginning of the actual program, through line 900; again, with the CURSOR at the first position in each line as it appears.
edit 900-	will list all lines, one at a time, starting at 900 through the end of the actual program; again, with the CURSOR at the first position in each line.

problems
****

There are errors in lines 1090, 1130, 1170 and 1220 in the following program. Enter the program in the computer exactly as below. Do not change them during typing. Any practice now will pay rich dividends in the future. After typing the program, use EDIT to debug and make corrections.

```
new

auto 1000

1000 page
1010
1020 // program on a rectangle
1030
1040 width:=6
1050 length:=14
1060
1070 // formula for area
1080
1090 area:=width/length
1100
1110 // formula for half the perimeter
1120
1130 halfperimeter:=width + width
1140
1150 // formula for total perimeter
1160
1170 perimeter:=2*width*2*lenth
1180
1190 // formula to get side of a square with
1200 // the same area as a rectangle
1210
1220 sideofsquare:=(area)↑1/2
1230
1240 print area
1250
1260 print halfperimeter
1270
1280 print perimeter
1290
1300 print sideofsquare
```

answers:

```
area = 84
halfperimeter = 20
perimeter = 40
sideofsquare = 9.16515139
```

CHAPTER SIX

introduction to $trings

****

objectives

At the completion of the chapter the student will be able to:

1. Enter the proper format to the computer to print $trings;
2. Use the trailing punctuation ";" and "," to get the desired printout;
3. Move text forward and backward by using, in proper sequence, the SHIFT key and the INST/DEL keys; and,
4. Employ the ZONE function to enhance the results put on the screen or to a printer.

```
comalcomalcomalcomalcomalcomalcomalcomalcomalcomal
c                                                c
o  ERASING                                       o
m  -------                                       m
a                                                a
l  Any portion of a program line --to its end--  l
c    can  be  erased by using the CTRL key  and  c
o    the letter "K."                             o
m                                                m
a  Place the CURSOR  over  the  first character  a
l    to be erased;  while pressing down on  the  l
c    CTRL key, tap the letter "K."  The balance  c
o    of the line  will be erased and the CURSOR  o
m    will remain so corrections can be typed.    m
a                                                a
comalcomalcomalcomalcomalcomalcomalcomalcomalcomal
```

6.1 **THE PRINTOUTS TO** this point have consisted only of numbers. Areas and perimeters of rectangles were given as numbers with no identification before or after the figure. This, of course, is not satisfactory; it doesn't give enough of a printout to make sense to another person without a great deal of work on their part --which, as we look at it, defeats one major strength of a computer.

In the last chapter, the area of one rectangle was 84 and its perimeter was 40. A printout similar to the following would have been better:

```
rectangle: 6 by 14

area = 84

perimeter = 40
```

The words "rectangle", "by", "area" and "perimeter" are called $trings. There are specific rules for getting the computer to print them.

6.2 **CLEAR THE SCREEN** and type NEW. Now, type

```
print comal
```

The computer does not oblige; instead we get an error message that reads

```
comal: unknown variable
```

Yet, by the previous discussion, "comal" is a $tring. Type

```
print "comal"
```

The computer does oblige this time and prints "comal."

Hence, our first observation: To get the computer to print $trings, we MUST USE QUOTATION MARKS around what we want printed.

Type

```
print 6
```

The computer does.

Type

```
print /
```

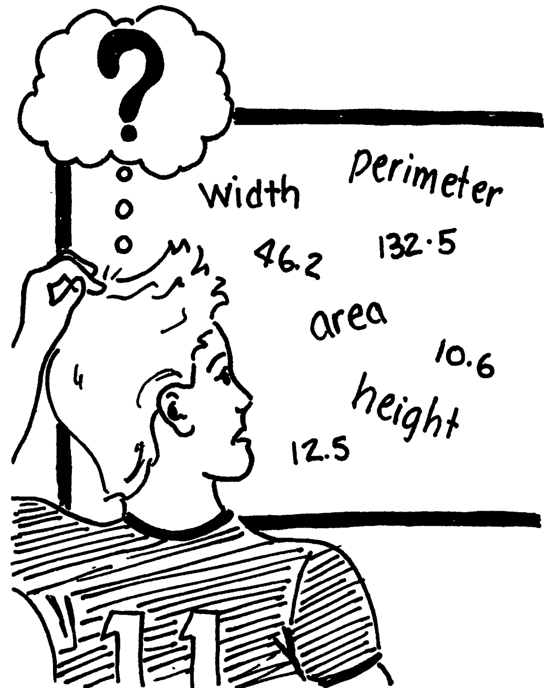
The computer responds with

```
expression expected, not "/"
```

To the computer, any non-numeral is a $tring and if it is to be printed, it must be enclosed in quotation marks.

Enter the following program into the computer

```
1000 page
1010
1020 // program for area & perimeter of rectangle
1030
1040 width:=12.5
1050 height:=10.6
1060
1070 // formula for area
1080
1090 area:=width*height
1100
1110 // formula for perimeter
1120
1130 perimeter:=2*width + 2*height
1140
1150 print area
1160
1170 print perimeter
1180
```



RUN the program. The answers should be 132.5 and 46.2. But what these represent is known only to us --unless someone takes the program and figures out what is going on.

(Before going further, SAVE the program in case we make a mistake and need to start over again.)

LIST the program to screen. Look at line 1150. It says

```
1150 PRINT area
```

This is the place where we want to put in a $tring identifying the area of the rectangle. Type

```
edit 1150
```

The line appears on the bottom of the screen (if the program is still listed on the monitor), with the cursor above the "P". Move the cursor over to the first letter of the word "area." Using the SHIFT key (keep it depressed) tap the INST/DEL key to open up several spaces. When this is done, and keeping the cursor right where it is, type --complete with the quotation marks **and** the spaces before and AFTER the "=" sign--

```
"area = ";
```

The line should now read

```
1150 PRINT "area = ";area
```

If the second "area" is not next to the semicolon, move the cursor so that it is over the "a" and then by tapping the INST/DEL key --**without** depressing the SHIFT key-- bring it back so that it is juxtaposed (adjacent) to the semicolon. RUN the program. We should have

```
area =  132.5
46.2
```

This is better! Now type

```
edit 1170
```

The screen will display

```
1170 PRINT perimeter
```

Again, move the cursor key to the first letter of "perimeter" and while pressing the SHIFT key and tapping the INST/DEL key, open up several spaces. Keeping the cursor right where it is type

```
"perimeter = ";
```

If the second "perimeter" is not juxtaposed to the semicolon, move the cursor over to the "p" and by tapping INST/DEL, bring it back.

RUN the program. The screen now reads

```
area =  132.5
perimeter =  46.2
```

Perhaps it would look better if there were a space between the two lines. We can accomplish this quite simply by inserting a line in the program between the two requested printouts. In this program type

```
1155 print
```

LIST it to see that it is there. After doing that, RUN it. The printout now shows

```
area =  132.5

perimeter =  46.2
```

And it does look better!

6.3 **THE USE OF** the semicolon after a PRINT statement is called TRAILING PUNCTUATION. It tells the computer that what ever is to be printed next --WHETHER IT BE A $TRING OR

A VARIABLE-- is to be printed one space after the end of the expression or variable which precedes the semicolon. In this instance it told the computer, "After you print 'area =', print the numeric value of the variable, area." The computer did just that.

There was no trailing punctuation after the variable AREA so the computer shifted or "returned the carriage" (as in a typewriter). When it fell into line 1155 the PRINT command there "returned the carriage" once more. At 1170, it did the same as it did on 1150.

It is essential to master trailing punctuation. And the very best way to learn it, is to experiment with it often.

Type

```
edit 1150 - 1170
```

On 1150 and 1170 change the ";" to commas. After the changes, RUN the program. The printout will be

```
area = 132.5

perimeter = 46.2
```

If the two different printouts are compared, it will be noted that the 132.5 and the 46.2 are CLOSER to the two $trings. In fact, if we hadn't put the space after the "=" sign in the $tring request, the 132.5 and 46.2 would have been immediately adjacent to the "=" signs. (Try it.)

This is the essential difference between the two. The ";" allows one space before printing out the next character, whereas the "," allows no spaces.

6.4 THERE IS ANOTHER advantage in using commas as trailing punctuation. It allows us to use another function to enhance the appearance of the output. It is called

ZONE.

In most other computer languages, it is a tedious and time-consuming task to get a balanced printout. But, in COMAL, the simple use of ZONE with commas as trailing punctuation (the ";" overrides the ZONE command) we can easily design the printout so that it works just the same on the printer as it does on the screen (something unheard of in some other languages!!).

Add the following lines to the program

```
1135 print "rectangle:";width;"by";height
1136 print
1137 zone 14
```

RUN the program. The printout is

```
rectangle: 12.5 by 10.6

area =      132.5

perimeter = 46.2
```

If we wish to move the 132.5 and 46.2 over farther then we can simply increase the value of ZONE.

In line 1135 we do not need to leave a space inside of the quotation marks for either "rectangle:" or the word "by" because the trailing punctuation, ";", does that automatically. **Width** and **height** are not in quotes because they are variables and have been assigned values in lines 1040 and 1050.

The first ";" after the $tring "rectangle", (line 1135). assures us that the value of the variable **width** will be printed immediately after it. The ";" after **width** keeps the $tring "by" on the same line as does the next ";" after "by" does for the variable **height**.

The **print** on line 1136 will cause a blank to be created between the print statement on line 1135 and the print command on line 1150.

A further comment on **ZONE** is in order. The numeric value after ZONE determines the placement of the output. The position of the first character in the output can be determined by counting the number of spaces from the first character in the **$tring** that precedes it.

Now, the $tring

```
"perimeter = "
```

contains 12 spaces, so we cannot call for a ZONE value less than 13 (one more than the number of spaces in the $tring); because the computer cannot handle the command, the printout becomes most irregular.

If we wish to line up the decimals in the printout, then we can add another line --1156, which will read

```
1156 zone 15
```

RUN the program.

With the changes and insertions we have made, it is time to

```
renum 1000
```

SAVE the program.

Unless we put in another ZONE command, the last one entered will cause all further output to be calculated with the same parameters. Our last ZONE command was

ZONE 15

All printouts requested after that, will be ruled by THAT command. If we wish to align other parts of the program differently, we will have to put in another ZONE value on another line. There is no restriction on the number of times ZONE can be placed in a program.

RECAP
*****

TO GET	ENTER
A $tring to print	Enclose the $tring in quotation marks
A space between lines in printout	Use a PRINT statement on a separate program line
Printouts on one line	Use trailing punctuation, either ";" or ","
Formatted printouts	Use the ZONE command

problems
****

Go back to the problems in Chapter Three and redo them. This time put in the PAGE command, documentation (//) and have the printouts labeled with $trings.

SAVE each problem to disc. Give consideration to the SAVE identification name being --say

chap 6 prob 1

and so forth.

For problems 1 and 2, add the perimeter; for 5, 6 and 7, add the circumference.

A typical printout (problem 5) might be:

```
circle:
radius =   10
area =     314.159266
circum =   62.8318531
```

answers

1. area = 72 perimeter = 36
2. height = 8 perimeter = 28
3. area = 14
4. base = 2.66666667
5. area = 314.159266 circumference = 62.8318531
6. radius = 4.85334231 circumference = 30.4944491
7. diameter = 16.4294487 circumference = 51.6146353
8. circumference = 62.8318531
9. circumference = 34.9276862
10. radius = 6.73862029
11. area = 249.893748
12. area = 90
13. sumbases = 23.7142857
14. base2 = 20.3157895
15. celsius = 25
16. fahrenheit = -40

CHAPTER SEVEN

more mathematical operations
****

objectives

At the completion of the chapter the student will be able to:

1. Identify the following as either functions or operators:

   abs; div; mod; sqr; int; and, rnd

2. Use the appropriate command to find the absolute value of a number;

3. Use the proper command to divide two numbers so that the answer is an integer;

4. Use the proper command to determine if the divisor is a factor of the dividend;

5. Use the appropriate command to find the square root of a number;

6. Write a correct expression to round numbers off to the nearest whole number, tenth, hundredths and thousandths;

7. Use the proper command to generate random numbers between 0 and 1;

8. Use the proper command to generate random numbers within any specified range; and,

9. "Seed" the computer with the command RANDOMIZE to generate a true random number.

```
comalcomalcomalcomalcomalcomalcomalcomalcomalcomal
c                                                c
o  CURSOR MOVEMENT                               o
m  ---------------                               m
a                                                a
l  To move the CURSOR forward one word tap  the  l
c    letter "F" while pressing the CTRL key.     c
o                                                o
m  To move  the  CURSOR back one word press the  m
a    the CTRL key and tap the letter "B".        a
l                                                l
comalcomalcomalcomalcomalcomalcomalcomalcomalcomal
```

7.1 IN THIS CHAPTER

IN THIS CHAPTER we will cover the following commands and operators:

abs

div

mod

sqr

int

rnd

7.2 THE ABSOLUTE VALUE

THE ABSOLUTE VALUE of a number (real or integer) is THAT number with a positive value. The absolute value of 6, is 6; the absolute value of -6, is 6.

To be sure, the use of this function is rather limited except in advanced mathematics. If we were comparing a range of test scores to the class average, we might not want the differences of the scores BELOW the average to be listed as negative values --particularly so, if we had a chart similar to the following.

average = 60

test scores	points above average	points below average
72	12	
85	25	
50		10
90	30	
45		15

The statement appears as

```
abs(-5.66)
```

Here, the answer is 5.66. The "-5.66" is the ARGUMENT of the function. In COMAL, the argument also may be a functional expression. For example

```
abs(8 - 10)
```

is valid.

```
abs(8 - 10) =

abs(-2) =

2
```

The argument may contain variables also.

```
abs(score - average)
```

In the chart above, the third score was 50 (average = 60). The computer would substitute those values.

```
abs(score - average) =

abs(50 - 60) =

abs(-10) =

10
```

If we put all these scores in a program with a loop (loops will be introduced starting with Chapter Nine), we could read each score, have the computer make the appropriate substitutions EACH TIME AROUND, and then get a printout similar to that of the chart.

Another ARGUMENT that can be used is

```
abs(int(num))
```

We haven't discussed integer (int) yet, but note that the ARGUMENT of ABS --int(num)-- is a function AND another argument. We will come back to this later.

7.3 **DIV IS AN** operator and, as such, returns an INTEGER answer to a division problem.

```
13 div 4
```

will return 3 only. The remainder, 1/4 or .25, is ignored. It's very useful when we wish only whole number answers in particular applications.

Division by 0 (zero) is not allowed.

7.4 MOD IS AN operator and, as such, returns an answer that is only the REMAINDER in a division problem.

    13 mod 4

will return 1 from the "1" in 1/4. (13/4 = 3 1/4 .)

    14 mod 4

will return 2 from the "2" in 2/4. (14/4 = 3 2/4 .)

MOD is a extremely useful operator. For example, if we wish to know whether the divisor --say, 5 (as an example), will "go into" the dividend --say, 15, evenly (the MOD operator would return a value of 0 if such were the case) then

    15 mod 5

would be THE method for determining that.

    15/5 = 3 0/5

In this case the returned value IS 0, so we can conclude, with certainty, 5 is, indeed, a factor of 15.

Here is a direct program to illustrate the principle.

```
1000 page
1010
1020 dividend:=15
1030 divisor:=5
1040 remainder:=dividend MOD divisor
1050
1060 PRINT "remainder =";remainder
```

RUNning the program gives

    remainder = 0

And we now know that 5 is a factor of 15.

When we use MOD later in our studies, it will be used as part of a more sophisticated program.

Again, as in DIV, division by 0 results in an error.

7.5 VERY OFTEN THERE is a need to find the square root of a number. Certainly, this function would have been useful for problems 6 - 9 in Chapters 3 and 6. It is written as

    sqr(num)

where "num" is the argument.

The argument can never be negative; if it is, the computer will return an error message.

```
sqr(16)
```

will return the positive root of 4.

```
sqr(-16)
```

will return the error message

```
ARGUMENT ERROR.
```

The ARGUMENT can also be a variable. Consider this simple program.

```
1000 number:=15
1010 result:=sqr(number)
1020 print "the square rooot of";number;"=";result
```

RUNning the program gives

```
the square root of 15 = 3.87298335
```

7.6 **IN PROGRAMMING, THE** integer (int) function is used over and over. Getting answers to the nearest penny or rounding answers to the nearest thousandths or tenths, are but two examples. And again, the argument of INT can be a real number, an integer, a variable or an mathematical function.

As a real number

```
int(6.89)
```

will return 6.

It might be argued that .89 is greater than one-half, so

```
int(6.89)
```

should return a 7. But keep in mind that in using the INT function, ANYTHING AFTER THE DECIMAL --.89 in this instance-- is **TRUNCATED**, or CUT OFF from the whole number.

The returned value is always --**ALWAYS**-- THE SMALLER NUMBER!! Hence

```
int(121.99999999)
```

returns 121.

```
int(0.9876)
```

returns 0.

```
int(-2.50)
```

will return -3!!

Why minus three?

Recall the number line; it looks, in part, like this:

```
---.---.---.---.---.---.---.---.---.---
   -4  -3  -2  -1   0   1   2   3   4
         ↑                   ↑
      -2.50                 2.50
```

Note that +2.50 (indicated by the RIGHT-SIDE arrow) lies between 2 and 3. Because

```
int(2.50)
```

TRUNCATES the .50, and leaves the **SMALLER** value --between 2 and 3-- the answer is 2.

However, -2.50 (indicated by the LEFT-SIDE arrow), lies between -3 and -2. -3 is **SMALLER** (less than) -2; hence, when

```
-2.50
```

is TRUNCATED

```
int(-2.50)
```

it will give the value of the SMALLER NUMBER which, in this example, is -3!!

7.7 **THERE IS A** method for rounding off real numbers to take into account the value of decimals .5 or greater. With this, numbers like 2.89 can be rounded to 3.00.

In **7.6** we observed that the argument for the function INT could be a mathematical expression. With that in mind, suppose, in a calculation involving money we were to take $16.78943 and express it to the nearest penny ($16.79).

As it now stands ($16.78943) is foolish. How would we read it? --As, sixteen dollars and seventy-eight nine-four-three cents!! What would that mean!

The first thing to do is to write $16.78943 as the ARGUMENT for an INT function.

```
int(16.78943)
```

If we did nothing further, the computer would return 16. But we want it

to be 16.79 (the "9" is greater than 4 so the 78 cents should round to .79).

The next thing to do is to multiply the argument by 100. Remember, pennies are two places after the decimal --which is the hundredths place.

```
int(16.78943*100)
```

This gives

```
int(1678.943)
```

If we stop here, the computer would return 1678 --certainly, not the correct answer!

Our next step is to add .5.

```
int(1678.943 + .5)
```

Which gives

```
int(1679.443)
```

If we stop here, the computer would return 1679 --still not the correct answer.

However, if we divide by 100! ...

```
int(1679.443)/100
```

then we would have

```
1679/100 =

16.79!!
```

All of these steps, of course, are combined into one initial statement

```
int(16.78943*100 + .5)/100
```

Here is an elementary program to illustrate the principle.

```
1000 page
1010
1020 // weight in pounds
1030
1040 weight:=6.7993
1050
1060 // price per pound
1070
1080 price:=1.29
1090
1100 // total cost
1110 totcost:=weight*price
1120 print "$",int(totcost*100 +.5)/100
```

At line 1110, totcost = 8.771097. But because of the ARGUMENT for INT in line 1120, the printout is $8.77. Take note that in the ARGUMENT of the function INT (totcost*100 + .5) we have used a VARIABLE (totcost) **AND** a MATHEMATICAL EXPRESSION (*100 + .5).

Onefurther comment on line 1120.

The use of a comma (,) as trailing punctuation assures us that the value of the INT function will be juxtaposed (placed) IMMEDIATELY after the $tring, $ --which, as it should be, is ENCLOSED in QUOTATOON marks so that the computer will print it as such.

Before concluding our discussion, let's look at this from another perspective. What would happen if the $16.78**9**43 had been $16.78**3**43 --a "3" instead of the "9" after the .78? Would it, had we applied the same procedures, have rounded out to $16.78, seeing that 3 is less than 5?

Consider

```
int(16.78343*100 +.5)/100
```

Multiplying by 100 would give

```
int(1678.343 + .5)/100
```

Adding the .5

```
int(1678.843)/100
```

Thus giving

```
1678/100 =

16.78
```

If we want answers to the nearest tenth, then we would mulitply by 10, add .5 and divide the result by 10. If we want an answer to the nearest thousandths (.001), then we would multiply by 1000, add .5 and divide the result by 1000.

7.8 **THE USE OF** random (rnd) numbers pervades a good deal of computer programming.

By random, we mean the selection of a number without regard to any specifio pattern or prejudice --i.e, any given number in a range of numbers has as much chance of showing up as any other number.

For example, if we want a random number from 1 through 6 (the values on the faces of a die!), 4 could just as easily appear as --say, 1.

If we wanted to survey 10 high school Juniors how would we pick them? Choose the officers? This would not be a random selection. Arrange them alphabetically and pick every fifth student from that list? Again, this

would not be a random selection.

One thing we could do is go to a book that has a TABLE OF RANDOM NUMBERS. We could take the first 10 numbers from that table, and if none are higher than the total membership of the class (in which case we choose the eleventh number, the twelfth, et cetera, until we get the ten numbers), we would count down the alphabetical list and choose the students by the position that the numbers indicated.

Say the first three numbers in the table are 71, 49, 3. Having arranged the class alphabetically, we would survey the seventh-first student, the forty-ninth student, and the third student.

When we generate a random number on the computer we have two choices

1. generate decimal numbers => 0 but < 1

or

2. generate whole numbers (positive and/or negative)

To generate a random number => 0 and < 1, type the following

```
print rnd
```

Do that a few times and note the printout. Here are some typical outputs:

```
.877953646

.088050889

.525670939

.628792926

.498504036
```

If, on the other hand, we want whole numbers as a printout, then we have to specify the range from which we wish the computer to generate them.

If we would like to simulate the throwing of a die, the random expression would be

```
print rnd(1,6)
```

The 1 and the 6 are inclusive.

Do that a few times. Here is a typical printout for six "throws":

```
4, 6, 5, 1, 1, 2
```

Because we are working with microcomputers, they need to be "seeded" when dealing with random numbers. This way we do not get the same random number sequence from program to program. If we did --that is, have the same random number sequence generated from one program to the next-- then we have the contradiction of a PREDETERMINED FINITE SEQUENCE OF RANDOM

NUMBERS!

To deal with this, we use the command

```
randomize
```

Using the TIME function built into the computer (which works on "jiffies" --60 jiffies in 1 second) RANDOMIZE initializes ("seeds") the first random number. Then when we call for the random number, we get a true one.

This program will print one "throw" of a pair of dice.

```
1000 page
1010
1020 // seeding the random number generator
1030
1040 randomize
1050
1060 // a pair of dice
1070
1070 die1:=rnd(1,6)
1080 die2:=rnd(1,6)
1090
1100 dice:=die1 + die2
1110
1120 print dice
```

When RUN, the program will generate a number from 2 through 12 --perhaps, the first time, a 10.

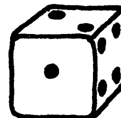
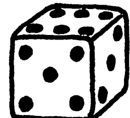
RUN the program several times, noting the printouts.

The beginning and ending numbers in the range can differ in value. If we wish to generate random numbers from 39 to 72, we enter

```
print rnd(39,72)
```

for 117 through 209

```
print rnd(117,209)
```

DICE =  + 

Quite straight forward!

DICE:= RND(1,6) + RND(1,6)

DICE:= DIE1 + DIE2

RECAP
*****

COMMAND	WHAT IT DOES
abs	returns the + value of an integer or real number
div	returns the integer value of a division operation
mod	returns the remainder of a division operation
sqr	returns the square root value of a positive number
int	returns the truncated lower value of a number
rnd	generates a random number, decimal or integer
randomize	"seeds" the computer so that it will generate a true random number

problems
****

Work the following by hand first. After that check the work on the computer.

1. Determine the absolute value of the following.

1.1) 6.8403 1.2) 0.401 1.3) -14.6 1.4) -0.001

1.5) -1/3 1.6) (-2)↑3 1.7) -8/2 1.8) 2↑(-2)

2. div

2.1) 16 div 4 2.2) 18 div (-8) 2.3) 14.2 div 3

2.4) .0463 div 5   2.5) -17.109 div .3   2.6) -5 div .1 (why?)

3. mod

3.1) 14 mod (-3)   3.2) 256 mod 16   3.3) 1 mod 9

3.4) 19 mod 4   3.5) -6.01 mod 2 (why?)

4. sqr

4.1) sqr(16)   4.2) sqr(25)   4.3) sqr(-49)

4.4) sqr((-2)↑2)   4.5) sqr(32/2)   4.6) sqr(4↑4)

5. int

5.1) int(6.991)   5.2) int(-0.991)   5.3) int(7.89*10+.5)/10

5.4) int(2*3.1)   5.5) int(84.635*100+.5)/100

5.6) int(-19.07499*100+.5)/100   5.7) int(-19.5)

6. Write the range for the following:

6.1) rnd   6.2) rnd(16,21)   6.3) rnd(48,49)

6.4) rnd(18.2,26.2)   6.5) rnd(101,263)

6.6) 6*rnd(1,6)   6.7) rnd(-4, -1)   6.8) rnd(-1,-4)

6.9) rnd(1,6) + rnd(1,6)

answers
****

1.1 6
1.2 0.401
1.3 14.6
1.4 0.001
1.5 .333333333
1.6 8
1.7 4
1.8 .25

2.1 4
2.2 -2
2.3 4
2.4 0
2.5 -57
2.6 -49 (Computer limitation when changing .1 to binary notation)

3.1 2
3.2 0
3.3 1
3.4 3
3.5 -9.99999978e-.03 (Which is -.00999999978. MOD, of course, should give a single digit answer; however, as the computer works in binary notation, these errors "work" their way in, in unusual commands.)

4.1 4
4.2 5
4.3 error message: argument error
4.4 2
4.5 4
4.6 16

5.1 6
5.2 -1
5.3 7.9
5.4 6
5.5 84.64
5.6 -19.07
5.7 -20

6.1 .0 through .999999999
6.2 16 through 21
6.3 48 or 49; (48 through 49)
6.4 18.2 through 26.2
6.5 101 through 263
6.6 6 through 36
6.7 -4 through -1
6.8 0 or -1. If, in the ARGUMENT for RND, the lowest number is not put first, the printout will not be consistent. There is an incorrect ARGUMENT here; -4 is lower than -1.
6.9 2 through 12

CHAPTER EIGHT

more on $trings
****

objectives

At the completion of the chapter the student will be able to:

1. Demonstrate the proper way to write a $tring variable;
2. DIMension a $tring variable;
3. Use two methods to get an output for the $trings of two or more $tring variables;
4. Output, using ASCII values, any letter of the alphabet;
5. Generate letters of the alphabet using a combination of CHR$ and (RND) statements; and,
6. Define and explain chr$.

```
comalcomalcomalcomalcomalcomalcomalcomalcomalcomal
c                                                c
o  RUNning a program                             o
m  -----------------                             m
a                                                a
l    After entering a program into the computer  l
c    it may be RUN by tapping the f7 key.  It    c
o    eliminates the need of typing RUN and tap-  o
m    ping the RETURN key.                        m
a                                                a
comalcomalcomalcomalcomalcomalcomalcomalcomalcomal
```

8.1 IN PREVIOUS CHAPTERS we have stored numbers in the computer's memory bank. We gave those numbers, variable names such as "width", "area", "radius", et cetera. We also can store $trings in the same manner. But in doing so, we must attach to the name of the variable we use to represent the $tring, the $ symbol. Hence

```
name:="roselyn"
```

would **not** be correct. Instead, it must be

```
name$:="roselyn"
```

8.2 IN ADDITION, THOUGH optional in COMAL through 40 characters, it is a good practice to DIMension all $tring variables. Most other languages (Pascal, e. g.) require DIMensioning regardless of the length of the $tring variable(s)!

$trings are made up of CHARACTERS. The word "COMAL" has five characters; hence, if it were set to a $tring variable, that variable would have to be DIMensioned to at least 5. This program should help in understanding the concept.

```
1000 page
1010
1020 // demonstration of $tring
       DIMensioning
1030
1040 dim first$ of 7, second$ of 3
1050 dim third$ of 8, fourth$ of 20
1060
1070 first$:="loretta"
1080 second$:="has"
1090 third$:="earrings"
1100
1110 fourth$:=first$ + " " + second$
     + " " + third$
1120
1130 print fourth$
```

RUNning the program gives

```
loretta has earrings
```

In line 1110 the "+" is **not** to be construed to mean ADDITION, as in arithmetic; rather, it simply means (when used in $trings) "to join to" or "put next to".

IT IS IMPORTANT to note how $tring variable names are DIMensioned. Look at line 1040. **dim** is the system-reserved word that alerts the computer to what we want to do. Once **dim** has been typed, succeeding $tring variable names to be dimensioned can be put on the same lines as

long as they are separated by commas. To be sure, what we typed on lines 1040 and 1050 could have been put on one line; however, not only does it "look" better to use two, but the program would be easier to debug, should there be errors.

Fourth$, which we DIMensioned at 20, MUST HAVE A DIMension VALUE that **includes** the total number of characters in FIRST$ plus SECOND$ plus THIRD$ string **plus** two for the spaces used between first$ and second$, AND between second$ and third$.

If we had DIMensioned fourth$ at 18, then

loretta has earrings

would become

loretta has earrin

The $tring variable, fourth$, may be DIMensioned at a larger value --say, 23. This will not cause havoc with the program, and it does save making tedious counts to determine minimum values. Be generous --but not foolish.

8.3 **WE CAN AVOID** the use of spaces, " ", when joining strings together. (They are used, by the way, to keep the words in the sentence separated.) We do this by including the spaces WITHIN the strings themselves. Here is a program for "loretta has earrings", that does just that. Make a careful comparison between the two algorithms --particularly, line 1080.

```
1000 page
1010
1020 // demonstration of spaces within $trings
1030
1040 dim first$ of 7, second$ of 5
1050 dim third$ of 8, fourth$ of 24
1060
1070 first$:="loretta"
1080 second$:=" has "
1090 third$:="earrings"
1100
1110 fourth$:=first$ + second$ + third$
1120
1130 print fourth$
```

RUNning

loretta has earrings

In line 1080 we put a space **before** and **after** the word **has** --and, inside the quotation marks!! Always remember:

INSIDE OF QUOTATION MARKS, THE COMPUTER
COUNTS A SPACE AS A CHARACTER!!!

Therefore, we DIMensioned " has " at 5 in line 1040.

Fourth$ was DIMensioned at 24, even though 20 would have been sufficient.

8.4 IN THE ASCII code, the chr$ value (CHR$ represents CHARACTER $TRING) for the letter "a" is 65 and for "z" it is 90. The rest of the alphabet lies between those two values.

If we

print chr$(65), the computer will return

a

print chr$(90), the computer will return

z

print chr$(78), the computer will return

n

If we wish the computer to generate letters RANDOMly (rnd), then the statement would be

```
print chr$(rnd(65,90))
```

Observe that the ARGUMENT for CHR$ is rnd(65,90) and, in turn, the ARGUMENT for RND is 65,90.

As in programs calling for a numeric printout, precede the statement with RANDOMIZE in order to "seed" the random generator.

> RECAP
> *****
>
> TO print $tring variables, they must first be DIMensioned.
>
> SPACES count as characters inside of quotes.
>
> THE + sign, when used with $trings, means "join to" or "put next to".
>
> CHR$ stands for CHARACTER $TRING.

problems
****

1. Write programs to print these sentences:

    1.1 i like computers

    1.2 yea! yea! football team!

    1.3 go, warriors, go

2. Write a program to generate, at random, any of the first three letters of the alphabet.

3. Write a program to generate, at random, any of the last three letters of the alphabet.

4. Write a program and put on three distinct lines, the following:

```
print chr$(rnd(77,79));
```

    Keep RUNning the program until the printout is

    m o n

    Count how many times it takes.

5. Do problem 4 three times and take an average of the number of times it takes.

6. Put on four distinct program lines the following:

```
print chr$(rnd(82,85));
```

    Keep RUNning it until the printout is

    r u t s

    Count how many times it takes.

****

Notation:

4. Mathematically, it would be once in 27 times.

5. Like 4, it should be 27.

6. Mathematically, it would be once in 256 times.

CHAPTER NINE

for / endfor loop
****

objectives

At the completion of the chapter the student will be able to:

1. Write and save a personal logo in such a way that it can be MERGEd with each new program written;

3. Write multi-line FOR/ENDFOR loops that call for complex instructions;

4. Write ONE-LINER FOR loops for single statement instructions;

5. Write and embed a ONE-LINER FOR time-delay loop in a program;

6. Use the visual technique for diagnosing possible errors in programming sequences having system-forced indentation;

7. Explain the NULL statement in COMAL; and,

8. Write time-delay loops of varying lengths, from 1/4 second up to 5 seconds.

```
comalcomalcomalcomalcomalcomalcomalcomalcomalcomal
c                                                c
o  DELETING PROGRAMS FROM THE DISC               o
m  ------------------------------                m
a                                                a
l  In a one disc drive system, the drive is la-  l
c    beled 0; if two drives are used, the other  c
o    is known as 1.                              o
m                                                m
a    To erase a file off the disc --say, it has  a
l    the name  RECTANGLE,  enter  this  command  l
c                                                c
o               delete "0:rectangle"             o
m                                                m
comalcomalcomalcomalcomalcomalcomalcomalcomalcomal
```

9.1 BEFORE A DISCUSSION of the for/endfor loop, there is another housekeeping item we need to look at.

Every algorithm should identify:

the programmer;
the name of the program; and,
the date written.

It would be time consuming to type this every time a program was written. With the use of the save command LIST and load command MERGE, this task can be accomplished quite easily.

Heading styles will vary with the programmer. The following is a guide. Type

```
9000 // select output "lp:"
9010
9020 //    ******************************
9030 //    * leary, j. wm / 1 sept 1986 *
9040 //    ******************************
9050 //    *       chap 9  prob 1       *
9060 //    ******************************
9070
```

Having double-checked for accuracy, enter

```
list "title.l"
```

Now, before we begin a new program, we can type

```
merge "title.l"
```

The command **merge** will send the computer to the disc, recover the file, and place it in the computer's memory. Further, it will ignore the 9000-9070 line numbering and, instead, give line numbers beginning with 0010. Before doing more on the program, a

```
renum 1000
```

would be in order, if that is to be our first line in the algorithm.

If we do a CAT listing (do so) we will note our program is saved as **seq** instead of **prg**.

The command

```
list
```

in

```
list "title.l"
```

places the algorithm on the disc as a SEQuential file. Because of the

way COMAL is designed, we easily can take SEQuential files off a disc and put (merge) them into any program. To remind us that this (or any program, for that matter) is a seq file, we attach the ".l" to its name. (Another way to take sequential files off a disc is to use the command ENTER. We will cover this in another chapter.)

The reason line 9000 is "//" is to allow running the programs on the monitor until all bugs in the algorithm have been worked out. Once we have the desired output then we can remove the "//" and RUN and LIST the program on the printer. If we are NOT going to include a copy of the program along with the output, then we can replace all of the //s with **print** and have the logo appear before the final output.

Though we have put in a specific date and an identification of a particular problem, it is easy enough --with the use of the CURSOR key(s)-- to make the necessary changes. Having a date and an identification already centered, will help putting in the new ones.

9.2 **TO THIS POINT,** we have had to EDIT a program to change the values of any variable(s) in order to have the computer run a program with different numbers. Certainly, this is most unsatisfactory; and if that is the only way computers could be made to function, the market would have dried up long ago. The very strength of computers is their ability to do repetitive work tirelessly and in a short period of time.

If we have a **KNOWN** number of variables then, most often, the **for/endfor** loop is the best way to write a program to process that data.

FOR loops may be multi-line or one line (not infrequently called, "one-liner fors").

9.3 **MULTI-LINE FORS** have a general format similar to

```
FOR  controlvariable:=start   TO   end   DO
 PRINT    instruction 1
 PRINT    instruction 2
 ............
 ............
ENDFOR
```

=== EXAMPLE. Print a name and address four times, leaving a space between each complete entry. One program might look like this

```
1000 for address:=1 to 4 do
1010 print "heather blaire"
1020 print "630 kimlin drive"
1030 print "glendale, ca.  91206"
1040 print
1050 endfor address
```

After LISTing the program, it would be shown on the monitor as

```
1000 FOR address:=1 TO 4 DO
1010  PRINT "heather blaire"
1020  PRINT "630 kimlin drive"
1030  PRINT "glendale, ca.  91206"
1040  PRINT
1050 ENDFOR address
```

The SYSTEM WORDS are capitalized. But more important to the programmer is the INDENTATION used in the loop. If the identation is not "closed" --that is, the "F" in FOR of line 1000 does not have the "E" in ENDFOR in the same relative position, then the programmer knows there is an error!!

Later, when we deal with more complex programming techniques, this becomes an indispensible visual debugging assist. And please note, this is system-forced!!

The control variable "address" in line 1000 is not "sacred." We just as easily could have used

```
for gumball:=1 to 4 do
```

The PRINT in line 1040 "kicks" the printer down one line after printing the previous three. This is what gives the spacing between the four adddresses when the program is RUN.

=== PROBLEM. Print the numbers from 10 through 99.

```
1000 for numbers:=10 to 99 do
1010  print numbers;
1020 endfor numbers
```

The trailing punctuation (;) in line 1010 prints the numbers across on the line (with a space between them) instead of straight down one column. A partial printout would be

```
10 11 12 13 14 15 16 ......
```

Replace the semicolon with a comma.

The numbers still are printed across the line, but in this instance there is no spacing between each one.

```
1011121314151617I8 ........
```

Add this line to the program

```
990 zone 6
```

BEFORE we RUN the program, we first must be certain there is a COMMA on line 1010. (Remember? The semicolon overrides the ZONE function.) RUN the program and note the printout.

It will be

```
11     12     13     14     15     16 .........
```

Though we could put the ZONE statement inside the loop, putting it just before accomplishes the same thing --and it does make it easier to debug. Unless we change the ZONE statement, anything that is printed after it will always have the same formatted printout. As noted earlier (Chapter 6), we can change the ZONE value within a program as often as we wish. Experiment with different values for ZONE.

9.4 **WE ALSO CAN** increment the control variable by a **STEP** value. Here is an example.

```
1000 zone 6
1010 for numbers:=10 to 99 step 2 do
1020  print numbers,
1030 endfor numbers
```

The printout begins as

```
10     12     14     16     18     20 ........
```

The **STEP 2** causes the computer to add 2 to the preceding number. The last number to be printed is **98**!! Though the control variable asked for 10 to 99, 98 + 2 is 100 which, being larger than 99, cannot be executed. (1, by the way, is the default value of STEP.)

Control variable parameters do not have to be integers --nor do STEP values. The following could be put in line 1010 above

```
for numbers:=1.04 to 33.56 step .91 do
```

The printout begins as

```
1.04  1.95  2.86  3.77  4.68  5.59 .......
```

STEP may also be negative. In so, then the START value must be larger than the END value.

```
1010 for numbers:=76 to 54 step -1 do
```

The printout begins as

```
76     75     74     73 .......
```

9.5 **ONE-LINER FORS** are just that, one line in length.

Some examples (type these and RUN them)

```
1000 for blanko:=1 to 39 do print " ",
```

another

```
1000 for dashes:=1 to 39 do print "-",
```

another

```
1000 for pause:=1 to 1000 do null
```

In one-line FORS, the **ENDFOR** statement must be **LEFT OFF**.

Any time we do not have a complex set of instructions for the END/ENDFOR loop to execute, always use a ONE-LINER FOR.

Throughout this course we will use variations of the last example

```
1000 for pause:=1 to 1000 do null
```

time and again in writing programs. It is called a time-delay loop.

COMAL recognizes NULL. Actually, nothing is done by the computer. We might vizualize the computer, when it reaches this line, as having to stop (pause) and count to 1000, by ones, before it can go on to the next line. (Remember the old hide-and-seek games we played as kids! When "it" we covered our eyes with our arms while leaning against a telephone pole or building, counted to 100 as fast as we could --faster than a computer does, even now?-- then shouted something like, ". . . 99, 100! Here I come, ready or not!!")

It takes about one second for the computer to "count" to a thousand. 1 to 500 takes approximately one-half second.

Later on, we will find many uses for a time-delay loop, and one-liner fors is an excellent way to execute them.

RECAP
*****

COMMAND	WHAT IT DOES
merge	takes a SEQuential file off the disc, renumbers the lines and puts it into the computer's memory.
for/endfor loop	used when a known set of complex data is to be processed.
step	determines the amount the control variable is to increased each time a loop is executed.
null	used in time-delay loops.

problems

****

For the following problems use only FOR/ENDFOR or ONE-LINER FOR loops.

1. Write a three-line cheer for a football team, with one space between each line. Print the cheer three times, with 2 blank lines between each printout.

2. Write the numbers from 1 to 10 with increments of .25:

   2.1 In a row; and,

   2.2 In a column.

3. Write the numbers from 9 to 4 with decrements of .25:

   3.1 In a row; and,

   3.2 In a column.

4. Print a line of 40 "*"s.

5. Print a 40-character line that alternates between a "*" and a blank space.

6. Using the CHR$ codes (see chapter 8 for help) print the alphabet:

   6.1 On a line with a space between each letter;

   6.2 On a line with no space between the letters;

   6.3 On a line with 5 spaces between each letter; and,

   6.4 In a column.

7. Write a program to produce the following printout:

```
a has ascii code 65
b has ascii code 66
c has ascii code 67
d has ascii code 68
.......
 (and so forth to)
          .........
y has ascii code 89
z has ascii code 90
```

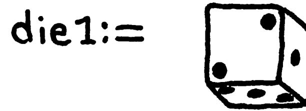
   Use a ONE-LINER FOR time-delay loop of 1/2 second inside the FOR loop.

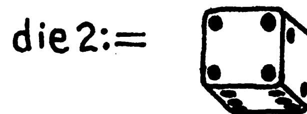
8. Throw a pair of dice 96 times and have the printout in columns --with 4 spaces between each column.

9. Starting with E --chr$(69)-- and ending with S, --chr$(83)-- have the computer select, at random, any 25 letters and print them in one row with a space between each.

10. Starting at 391 and ending at 617 select, at random, 50 numbers and have the computer print them in columns with appropriate spacing, so they can be read easily.

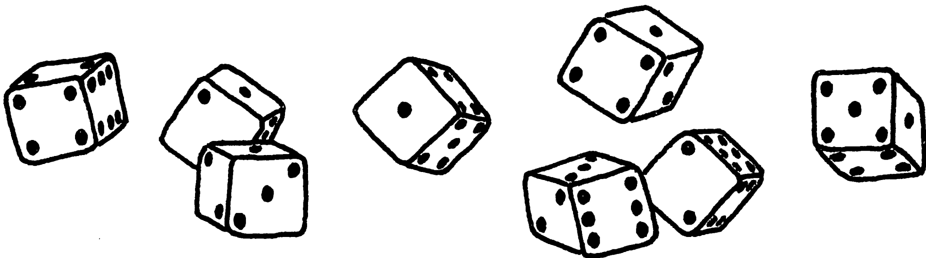
FOR dembones :=1 TO 4 DO

die1:= 

die2:= 

PRINT die1; die2

ENDFOR dembones



CHAPTER TEN

counting and summing
****

objectives

At the completion of the chapter the student will be able to:

1. Include a counting statement in a program;

2. Include a summation statement in a program;

3. Explain why it is important to initialize a counting and a summation statement outside of a loop;

4. State what the "computerese" for

   :+

   is in a counting statement;

5. Show, with a series of mathematical statements, the progression of a summation statement in a program; and,

6. Write a program to strike an average of several numbers;

```
comalcomalcomalcomalcomalcomalcomalcomalcomalcomal
c                                                c
o  RUNNING PROGRAMS DIRECTLY FROM CAT LISTING     o
m  ------------------------------------------     m
a                                                a
l  After listing the directory to the screen by  l
c    the CAT command, any program (prg) can  be  c
o    RUN by  moving the CURSOR to the beginning  o
m    of the listing and then tapping the f7 key. m
a                                                a
comalcomalcomalcomalcomalcomalcomalcomalcomalcomal
```

10.1 THERE ARE TIMES

THERE ARE TIMES when it seems every program we work on has a counting and/or summation loop. And it is not too far off to say that counting and summing are almost as generic to computing as letters are to the alphabet.

In COMAL, the counting loop reads

```
count:+1
```

Notice the absence of spacing in the statement.

The word "count" is not a reserved or system word. Any of the following are valid counting loops.

```
kount:+1

countdice:+1

kounthens:+1

counteggs:+1

gumballs:+1
```

One thing that is **very important** in COMAL is the absolute necessity of initializing the count variable --i.e., it must be set to zero (or another value, if it is appropriate) **before** the count loop is embedded in the program. Failure to do this results either in an error message or "funny" answers.

Here is a program to illustrate a counting loop.

```
1000 count:=0
1010 //
1020 for dye:=1 to 15 do
1030 die:=rnd(1,6)
1040 count:+1
1050 print die;
1060 endfor dye
1070 //
1080 print "# of times die tossed =";count
```

(In order to avoid "turmoil" and program breakdown, the variable **dye** in line 1020, is SPELLED DIFFERENTLY than the variable "die" in 1030. We must do this or the computer becomes confused because it cannot keep the two variables separated --and chaos becomes the order of the day!)

We already know from line 1020 that the program is going to loop 15 times; hence, the count loop in 1040 is superfluous, to be sure. However, at this stage of our programming knowledge, the only loop we have at our disposal is the FOR/ENDFOR one.

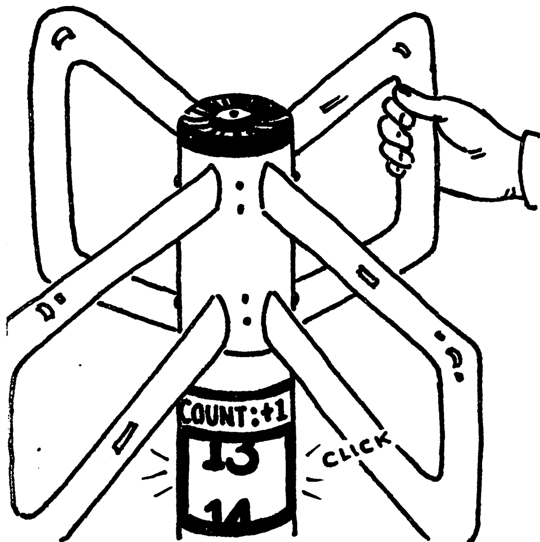
Note that the COUNT statement was initialized to 0 in line 1000 AND OUTSIDE THE LOOP!! In line 1040

```
count:+1
```

is the counting statement.

It might help us to grasp what is happening if we imagined THE ENTIRE STATEMENT, including the line number

```
1040 count:+1
```



as a TURNSTILE. As the computer, starting at line 1000, progresses through the program, it falls through from line to line. When it falls through line 1040 it "clicks" the TURNSTILE one turn. Because line 1040 is within the for/endfor loop, the computer will "click" through 15 times (in this program). As it does, COUNT is incremented by 1 each time.

Indeed, as the computer "clicks" through the TURNSTILE (line 1040), the

```
:+
```

is "computerese" for

"whatever value COUNT has, increment (add) ---"

and in this instance, ". . . add 1."

Because, on line 1040 we set (initialized) COUNT to 0, the first time through the TURNSTILE, COUNT is 0 and the statement

```
count:+1
```

becomes

```
0:+1 =  1
```

So COUNT, then, is 1 the first time through. On the second trip

```
count:+1
```

becomes

```
1:+1 =  2
```

Counting loops can DECREMENT as well as INCREMENT and situations arise when programs need to be written this way.

NEVER INITIALIZE A COUNTING VARIABLE WITHIN A LOOP! Always initialize OUTSIDE the loop; otherwise, every time the computer falls through

that line, it will set the counter back to 0.

The following program demonstrates this error.

```
1000 page
1010
1020 randomize
1030
1040 sumnums:=0
1050
1060 for errordemo:=1 to 10 do
1070
1080  count:=0
1090  nums:=rnd(54,76)
1100  count:+1
1110
1120 sumnums:+nums
1130
1140 endfor errordemo
1150
1160 averageofnums:=sumnums/count
1170
1180 print "average =";averageofnums
```

Because

1080 count:=0

is **WITHIN** the for/endfor loop, each time the computer falls through line 1080, "count" is set back to 0; therefore, at the end of the 10th loop, COUNT will only be **1**. And the average taken in line 1160 will be the total sum divided by 1 --not 10!

10.2 **IN THE PRECEDING** program, line 1120 is a SUMMATION STATEMENT. The format is similar to counting statements, but instead of incrementing by 1 it increments by a variable amount.

Like COUNT, summation statements also must be initialized and, again, **OUTSIDE** the loop (except for those "delightful" detours we will confront later in the text). See line 1040 in the above program.

Summation statements, as can be seen from line 1120 above, take the form

```
sumnums:+<variable>
```

As in COUNT, there are no "sacred" words. Any of the following are valid.

```
sumdie:+die1

sumnumbers:+integers

sumeggs:+eggs
```

This program illustrates a summation statement.

```
1000 page
1010
1020 randomize
1030
1040 sumdie:=0
1050
1060 for dye:=1 to 6 do
1070
1080  die:=rnd(1,6)
1090
1100  sumdie:+die
1110  print die;
1120
1130 endfor dye
1140
1150 print "sum of die tosses =";sumdie
```

For the sake of illustration, let's assume the 6 tosses of the die came out to be

4 4 1 6 3 3

On the 6 trips through line 1100, this is how it would be processed by the computer.

trip	die	summation
1	4	0:+4 = 4
2	4	4:+4 = 8
3	1	8:+1 = 9
4	6	9:+6 = 15
5	3	15:+3 = 18
6	3	17:+3 = 21

And the printout for line 1150

sum of die tosses = 21

There are times when we may wish to set up a summation loop that DECREMENTS by a variable, until 0 or another predetermined value has been reached.

10.3 **IF WE WISHED** to add the values of the six "tosses" of the die in the preceding program and take

an average-per-toss, we could add a count statement to the algorithm.

```
1000 page
1010
1020 countdie:=0
1030 totaldie:=0
1040
1050 randomize
1060
1070 for dye:=1 to 6 do
1080
1090  die:=rnd(1,6)
1100  totaldie:+die
1110  countdie:+1
1120  print die;
1130
1140 endfor dye
1150
1160 print "sum of die =";totaldie
1170
1180 ave:=totaldie/countdie
1190
1200 print "ave value of each throw =";ave
```

The loop lies between lines 1070 and 1140. The count statement is on line 1110 and was initialized at 1020; line 1100 is the summation statement which was initialized on line 1030.

Line 1090 generates the value of each "toss" of the die; line 1120 prints that value --in a row, because of the trailing punctuation. Outside of the loop, the average (ave) is struck by dividing the sum (totaldie) by the number of tosses (countdie). (ave value of each throw = 3.5)

After entering the program, RUN it four or five times. If the program is not clear, be certain not to go on until the chapter has been studied again.

```
.................................................
.                                               .
.  RECAP                                        .
.  *****                                        .
.                                               .
.  COUNT statements take the general form       .
.                                               .
.              count:+1                         .
.                                               .
.  SUMMATION statements take the general form   .
.                                               .
.              sum:+<variable>                  .
.                                               .
.  BOTH must be initialized first.              .
.                                               .
.................................................
```

problems

****

(The student should understand that the use of a count statement inside of a for/endfor loop is superfluous for the following problems. Generally, when such a loop is started at 1, the ending value (the BEGINNING value if the loop decrements to 1) would be the same as the final value of an included count statement. However, because the chapter is addressing only COUNTING and SUMMATION STATEMENTS, this becomes the primary objective.)

Use a FOR/ENDFOR loop in each of the following exercises.

1. Toss a die 10 times. Have the printout show the value of each toss, the sum of the tosses and the average value per toss.

2. Throw a pair of dice 20 times. Have the printout show the value of each throw, the sum of the throws and the average value per throw.

3. Throw a pair of dice 100 times. Have the printout in ten columns. Print the value of the sum and the average value per throw. Use another ZONE value to line up the sum and average.

4. Generate 50 random numbers from 103 through 341. Print their values in five columns. Print the sum of the numbers and their average. Use ZONE to align all printouts.

5. Generate 30 random numbers >0 and <1. Line up their values in columns. Print the sum of the numbers and their average. Use ZONE to align all printouts.

CHAPTER ELEVEN

binary decisions: if/then/else

objectives

At the completion of the chapter the student will be able to:

1. Write down and illustrate, mathematically, the five relational comparative symbols;

2. Write a program where the predefined constants for Boolean variables of 0 and 1, represent FALSE and TRUE, respectively

3. Write an IF/THEN/ELSE program where the variable is evaluated by Boolean logic, and the predefined constant is generated by a random statement;

4. Write programs involving the following five IF structures:

   11.1 IF ... THEN ...

   11.2 IF ... THEN ... ENDIF

   11.3 IF ... THEN ... ELSE ... ENDIF

   11.4 IF ... THEN ... ELIF ... ENDIF

   11.5 IF ... THEN ... ELIF ... ELSE ... ENDIF

   and,

5. Write a program with any of the five IF structures embedded within a FOR/ENDFOR loop.

comalcomalcomalcomalcomalcomalcomalcomalcomalcomal

EDITING WITH THE f5 KEY

Tapping the f5 key places the computer in the EDIT mode.

After tapping f5, tapping the RETURN key will list, line by line, the entire program.

To get a specific line or lines, enter them immediately after tapping f5. Tap the RETURN key.

comalcomalcomalcomalcomalcomalcomalcomalcomalcomal

11.1 IN CHAPTER ONE we introduced relational comparisons --or inequalities, as they are known.

- > meaning, greater than
- < meaning, less than
- <> meaning, not equal to

At this point we will expand those to include the following complete table.

statement	illustration	translation
>	6>4	6 greater than 4
=>	4=>4 6=>4	4 equal to **or** greater than 4 6 equal to **or** greater than 4
<	4<6	4 less than 6
=<	4=<4 4=<6	4 equal to **or** less than 4 4 equal to **or** less than 6
<>	4<>6 6<>4	4 not equal to 6 6 not equal to 4

We will begin using these in programs.

11.2 A TYPICAL FORM of a binary decision is

```
IF <condition> THEN
 statement(s)
ELSE
 statement(s)
ENDIF
```

(Note the system-forced indentation --a debugging aid-- and the capitalization of the system words.)

Here is an elementary program to illustrate the IF/THEN/ELSE/ENDIF paradigm.

```
1000 PAGE
1010
1020 RANDOMIZE
1030
1040 die:=RND(1,6)
1050
1060 IF die =< 3 THEN
1070  PRINT "you tossed a";die
1080  PRINT "you win $5"
1090 ELSE
1100  PRINT "you lose $6"
1110 ENDIF
```

If, in line 1040, we "throw" a 1 or 2 or 3 then the **condition** in line 1060 is true --and lines 1070 and 1080 will be executed. Lines 1090 and 1100 will be ignored, and line 1110 will terminate the sequence.

On the other hand, if we "throw" a 4 or 5 or 6 then the **condition** in line 1060 is FALSE --**and** lines 1070 and 1080 will be ignored. Lines 1090 and 1100 will be executed and, again, line 1110 will terminate the sequence.

11.3 BOOLEAN VARIABLES ARE named after the 19th century mathematician George Boole. He had a great deal to do with developing the principles of mathematical logic.

Boolean variables deal with the logic concept of FALSE and TRUE. In COMAL, **FALSE** has the predefined constant that is always equal to 0. **TRUE** is also predefined and is any value <> 0 --and 1 is the accepted value to use.

In the previous program on tossing a die, if the condition was met then, explicitly, it was TRUE; and the statements associated with that truth (lines 1070 - 1080) were executed, and the sequence was terminated. If the condition was **not** met then, explicity, it was FALSE; and the statements associated with it (lines 1090 and 1100) were executed, and the sequence terminated.

Consider these lines taken out of another program.

```
1050 paint:=rnd(0,1)
1060 IF paint THEN
1070  PRINT "buy white paint"
1080  PRINT "paint kitchen wall"
1090 ELSE
1100  PRINT "kitchen walls ok, go to beach"
1110 ENDIF
```

The random number generated in line 1050 will be either 0 or 1.

FALSE is predefined as 0.

TRUE is predefined as 1.

```
..................................................................
..                                                              ..
.. Rewriting the program as the computer "sees" it--            ..
..                                                              ..
..         1050 paint:=rnd(0,1)                                 ..
..         1060 IF paint comes out to be 1 THEN (I will)        ..
..         1070          "buy white paint" (and)                ..
..         1080          "paint kitchen wall"                   ..
..         1110 ENDIF                                           ..
..................................................................
```

(It skips over lines 1090 and 1100.)

```
..................................................................
..                                                              ..
.. The other condition would be--                               ..
..                                                              ..
..         1050 paint:=rnd(0,1)                                 ..
..         1060 If paint comes out to be 0 THEN                 ..
..         1090         (instead, I find the)                   ..
..         1100         "kitchen walls ok, go to beach"         ..
..         1110 ENDIF                                           ..
..................................................................
```

(It skips over lines 1070 and 1080.)

What we are saying is, that the IF condition is implicitly **true** and the ELSE is the contravention of that --hence, **false**.

11.4 **WE ARE NOT** restricted to one ELIF. There may be a multiple set of them. Very often, when this condition exists, another statement **CASE** can be used (which we will study in a later chapter).

To illustrate, let's write a program for tossing a die

```
1000 page
1010
1020 randomize
1030
1040 die:=rnd(1,6)
1050
1060 if die=1 then
1070 print "1 was thrown"
1080 elif die=2 then
1090 print "2 was thrown"
1100 elif die=3 then
```

```
1110 print "3 was thrown"
1120 elif die=4 then
1130 print "4 was thrown"
1140 elif die=5 then
1150 print "5 was thrown"
1160 elif die=6 then
1170 print "6 was thrown"
1180 else
1190 print "check your program"
1200 endif
```

When LISTed, the program appears on the screen as

```
1000 PAGE
1010
1020 RANDOMIZE
1030
1040 die:=RND(1,6)
1050
1060 IF die=1 THEN
1070  PRINT "1 was thrown"
1080 ELIF die=2 THEN
1090  PRINT "2 was thrown"
1100 ELIF die=3 THEN
1110  PRINT "3 was thrown"
1120 ELIF die=4 THEN
1130  PRINT "4 was thrown"
1140 ELIF die=5 THEN
1150  PRINT "5 was thrown"
1160 ELIF die=6 THEN
1170  PRINT "6 was thrown"
1180 ELSE
1190  PRINT "check your program"
1200 ENDIF
```

All the conditions are TRUE except line 1180. It would seem, we really do not need line 1180 or line 1190. After all, a die has only six faces and one of them has to show up on any given toss. On the other hand, suppose we had made an error in writing the algorithm. What if we had written, by chance, the random number as

```
rnd(1,7)
```

or

```
rnd(0,6)
```

Lines 1180 and 1190 would have "caught" the error --gooooooooood programming technique!!-- if, in the first instance, the "7" had been generated or, in the second instance, the "0" had been generated.

11.5 WE ALSO CAN have one-line binary decision statements.

They take the general form of

```
if die<>6 then count:+1
```

LISTed, it appears as

```
IF die<>6 THEN count:+1
```

There is no ENDIF with a one-line IF/THEN.

11.6 SINGLE- OR MULTI- line binary decision statements can be embedded within FOR/ENDFOR loops.

Here is an example.

```
1000 PAGE
1010
1020 // initializing two count statements
1030
1040 count:=0
1050 count7:=0
1060
1070 RANDOMIZE
1080
1090 FOR dye:=1 to 10 DO
1100
1110  die1:=rnd(1,6)
1120  die2:=rnd(1,6)
1130  dice:=die1+die2
1140  PRINT dice;
1150
1160  IF dice<>7 THEN
1170   count:+1
1180  ELSE
1190   count7:+1
1200  ENDIF
1210
1220 ENDFOR dye
1230
1240 PRINT "# of throws not equal to 7 =";count
1250 PRINT
1260 PRINT "# of throws equal to 7 =";count7
```

Type in the program and after studying it, RUN it a few times.

By the way, a one-line IF/THEN may call, as multi-line ones do, for print statements of even PROCEDURES --something we also will be studying later. An example of the former would be

```
IF dice=7 THEN PRINT dice;
```

This is one way of getting just the 7s printed --or any other sequence of numbers we might wish.

==== problem

Generate 20 random numbers from 341 through 619. Print only the odd ones. Count how many there are, sum them and take an average per number.

```
1000 page
1010
1020 // program to generate 20 rnd #s
1030
1040 // initializing variables
1050
1060 countodd:=0
1070 sumodd:=0
1080
1090 randomize
1100
1110 for figures:=1 to 20 do
1120
1130  numbers:=rnd(341,619)
1140
1150  if numbers mod 2 <> 0 then
1160   print numbers;
1170   sumodd:+numbers
1180   countodd:+1
1190  endif
1200
1210 endfor figures
1220
1230 average:=sumodd/countodd
1240
1250 print " number of odd integers =";countodd
1260 print
1270 print "  total of odd integers =";sumodd
1280 print
1290 print "average of odd integers =";average
```

Some comments:

Line 1140 MOD gives the remainder of a division problem. If we divide --say, 316 by 2, the answer is 158 and no remainder, because 2 "goes evenly" into 316. If, on the other hand, we divide --say 437 by 2, the answer is 218 1/2; a remainder of 1. Therefore, stating--

IF numbers mod 2 NOT EQUAL TO 0 THEN
[IF numbers MOD 2 <> 0 THEN]

Is a good method for determining if an integer is even or odd. If there is a remainder, it is odd; if there is no remainder, then it is even.

This same logic can be extended to any other condition we might want. For example, if we want to know if an integer is divisible by 7, then the

following statement would do it.

```
if integer mod 7 = 0 then print "7 is a factor of";integer
```

If there IS a remainder, it means the integer was NOT divisible by 7.

```
RECAP
*****

BOOLEAN variables are pre-defined.

            0 = false
            1 = true

There are five structures for IF/THEN

            IF/THEN
            IF/THEN/ENDIF
            IF/THEN/ELSE/ENDIF
            IF/THEN/ELIF/ENDIF
            IF/THEN/ELIF/ELSE/ENDIF
```

problems

****

For the first four problems use the predefined values for the Boolean variables --and RND to generate them-- then write a program:

1. For the variable "rain"

   If true, then
   it is raining
   put shovel away

   If false, then
   get the shovel
   dig a 6x6x6 foot hole

2. For the variable "test"

   If true, then
   study like mad
   computer test tomorrow

   If false, then
   look at tv
   snow day tomorrow

3. For the variable "girlfriend"

   If true, then
      buy flowers
      you get father's car

   If false, then
      buy audio tape
      girlfriend fickle

4. For the variable "boyfriend"

   If true, then
      put on long earrings
      he's taking me out

   If false, then
      call up bill
      he likes long earrings

5. Throw a pair of dice 100 times. If a 7 comes up, print it; otherwise, print a period.

   Count the number of 7's.

   The final printout should be:

   the number of 7s tossed = --

6. Redo 5 and put a printout on the screen so that it is in eight columns. Have the same final printout.

7. Throw a pair of dice 100 times. Print them in columns.

   Count the number of times a 1, a 7 or an 8 appears.

   Have the final printout show the results for each number.

8. Throw a pair of dice 100 times. Print only the tosses equal to or greater than 8. The other printout should be a blank space. Put the printout in columns. Have the final statement tell how many of each number.

9. Throw a pair of dice 20 times. Add the face value of all throws =>7. Use a count statement inside the IF/THEN loop.

   The final printout should read

   number of tosses => 7 --

   sum of tosses => 7 --

   ave of these tosses = --

Use ZONE to get a "decent" print format.

10. Redo 9 with the same first two final printouts. The third printout, however, must have an integer answer. It must read

    ave of the tosses was --

OR

    ave of the tosses was about --

(HINT: Consider use of MOD and INT.)

11. Generate 17 random numbers between 119 and 231. Print them in columns. Add them. Strike an integer average (as in 10).

    Final printout should reflect mature programming skills.

12. Generate 42 random numbers between 213 and 695. Print, in columns, only those divisible by 3. Add them and strike an actual average.

    The final printout should be complete.

    (HINT: IF <variable> MOD 3 = ?? THEN .....)

13. Throw a pair of dice 10 times. Have the printout in the form of

```
throw  shows  total
-----  -----  -----

1      6      6
2      9      15
3      11     26
4
```

and so forth

14. Throw a pair of dice 100 times. Count the number of 2's, 3's, 4's .... 12's tossed. Have the final printout reflect the total for each of the eleven numbers.

15. Throw a pair of dice 100 times.

    Sum the value of the dice for 8 or less.
    Sum the value of the dice for 9 or more.
    Have the final printout compare the sums.

16. Generate 20 random numbers from 601 through 918.

    Add the odd ones and compare that with the sum of the even ones.

    Be certain to print the numbers in columns across the screen.

CHAPTER TWELVE

binary numbering system
****

objectives

At the completion of the chapter the student will be able to:

1. Define the acronym ASCII;
2. Define the acronym BIT;
3. Tell how many bits make a byte;
4. Write out the numerical progression for a single byte;
5. Write integers in binary code; and,
6. Change binary code into integers;

```
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
c                                                               c
o  PROGRAM SCROLLING                                            o
m  -----------------                                            m
a                                                               a
l  As programs become longer and longer,  it becomes a problem  l
c    to follow them on the screen after issuing a LISTing com-  c
o    mand.                                                      o
m                                                               m
a  The listing can be stopped and started by tapping the SPACE  a
l    BAR. The SPACE BAR, in effect, becomes  a  switch, alter-  l
c    nately  stopping and starting the scrolling of the lines.  c
o                                                               o
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
```

12.1 TYPE THE FOLLOWING command.

use dansk

Tap the RETURN key.

Having done that, type "h" and then tap the RETURN key. The following will appear on the screen

h: ukendt saetning eller procedure

Because we do not have a command of Danish, the message following "h:" has no meaning --it fails to **communicate**.

Type

use english

Tap the RETURN key.

Once more, type "h" and then tap the RETURN key. The following will appear on the screen

h: unknown statement or procedure

The message now has been **successfully** communicated!! Why? Because we have a mastery of English. We have read the same message twice, but we only were able to **translate** one of them.

The same is true for the computer. In order for it to process our typed-in messages, these commands first must be handled by the Central Processing Unit. The CPU then **translates** our messages into binary code which, in turn, is directed to and handled by, the rest of the computer. Once the computer has completed the task(s) we have assigned, the results are sent back through the CPU which **translates** this back into English and outputs the results to the screen and/or the printer.

It is a historic fact that language has been a barrier to the development of many Asian countries. Because they had no alphabet and used, instead, "pictures" for each word, communication and, in particular, communication of ideas, was severely hampered.

If computers were developed around a language using $tring alphabets or "word" pictures, they would be so cumbersome as to be totally ineffective --especially in view of the horrendous investment of capital it would take to make them, as measured against the value of their output.

Therefore, the engineers employed a numeric language --and a language with a base two, instead of the base ten we use.

(10 units = 10
10 tens = 100

10 hundreds = 1000)

In a further stroke of genius, they used 0 and 1 as the two numbers. By doing this, electronic "toggling" could be employed. Either a circuit was closed or open --0 or 1; hence, **binary digits**. Each binary digit is called a **bit** --an acronym for BInary digiT; 8 bits make a **byte**. Each byte represents a character. The character may be a space, a letter, a number, a punctuation mark, a command or even a special manufacturer-designed symbol.

12.2 **IT SOON BECAME** apparent that there needed to be some standard agreement among manufactures of computers, as to what the values of these bytes would represent. Unless there was, then one manufacturer might assign the letter "A" a 65, another, 13 and a third 112. To avoid this possibility, a code was developed. It was named the American Standard Code for Information Interchange --or, **ASCII** (as-kee'), as it is "affectionately" called. Most of the computer systems in the world use it.

In ASCII, the value of an "a" is 65, the value of "z" is 90, the value of "5", 53, and the value of "+", 43 --to name a few.

So, when the computer is asked to store an "a", it stores a "65" because it is easier to do so. When the CPU, in turn, reprocesses a 65, it outputs an "a" --either to the screen or to the printer. And because it can arrive at 65 very quickly (due to binary digit processing) it makes, not only what otherwise would be an impossible task possible, but it does so with singular dispatch!

12.3 **IN ORDER TO** make provision for all the numerals, letters, symbols and commands, it takes a combination of eight (8) bits. In binary notation the scale appears as

128 64 32 16 8 4 2 1

Take note, that

$2^0 = 1$

$2^1 = 2$

$2^2 = 4$

$2^3 = 8$

$2^4 = 16$

$$2^5 = 32$$

$$2^6 = 64$$

$$2^7 = 128$$

With this scale --128, 64, 32, 16, 8, 4, 2 ,1-- we can write the numeric value of any ASCII code we choose, from 0 through 255. However, before doing so, we must be able to either TURN OFF or TURN ON each of the 8 bits --and, in any sequence! The engineers have so designed the computers that this is done by the CPU. All we have to do is type --say, the letter "a" and the CPU will turn on (or off) the appropriate bits. Remembering, the binary digits are 0 and 1 ("off" and "on"), the eight-bit byte for the letter "a" would look like this

```
                 128 64 32 16 8 4 2 1
                   ↑  ↑  ↑  ↑ ↑ ↑ ↑ ↑
bits: off/on--     0  1  0  0 0 0 0 1
                      ↑             ↑
                     64            +1= 65
```

The ASCII code for "j" is 74. Through the electronic "toggling" --done, by the way, with electrical impules, 74 is

```
                 128 64 32 16 8 4 2 1
                   ↑  ↑  ↑  ↑ ↑ ↑ ↑ ↑
bits: off/on--     0  1  0  0 1 0 1 0
                      ↑       ↑   ↑
                     64      +8  +2  = 74
```

By sending the appropriate signal, the CPU "switched" ON the bits for 64, 8 and 2. These were added, and the proper storage of the letter "j" was completed. Because of the speed of electricity, this is accomplished, for all practical purposes, immediately.

"Z" has the ASCII code of 90.

```
                 128 64 32 16 8 4 2 1
                   ↑  ↑  ↑  ↑ ↑ ↑ ↑ ↑
bits: off/on--     0  1  0  1 1 0 1 0
                      ↑     ↑ ↑   ↑
                     64   +16+8  +2  = 90
```

The plus symbol (+) is 43.

```
                 128 64 32 16 8 4 2 1
                   ↑  ↑  ↑  ↑ ↑ ↑ ↑ ↑
bits: off/on--     0  0  1  0 1 0 1 1
                         ↑    ↑   ↑ ↑
                        32   +8  +2+1= 43
```

The "$" is 36.

```
                         128 64 32 16 8 4 2 1
                           ↑  ↑  ↑  ↑ ↑ ↑ ↑ ↑
    bits: off/on--         0  0  1  0 0 1 0 0
                                 ↑      ↑
                                32     +4     = 36
```

The value of the "3" is 51.

```
                         128 64 32 16 8 4 2 1
                           ↑  ↑  ↑  ↑ ↑ ↑ ↑ ↑
    bits: off/on--         0  0  1  1 0 0 1 0
                                 ↑  ↑     ↑ ↑
                                32+16    +2+1= 51
```

Here is a table of some of the more common ASCII Codes.

CHAR	CODE	CHAR	CODE
!	33	>	62
"	34	?	63
#	35	@	64
$	36	a	65
%	37	b	66
&	38	c	67
'	39	d	68
(	40	e	69
)	41	f	70
*	42	g	71
+	43	h	72
,	44	i	73
-	45	j	74
.	46	k	75
/	47	l	76
0	48	m	77
1	49	n	78
2	50	o	79
3	51	p	80
4	52	q	81
5	53	r	82
6	54	s	83
7	55	t	84
8	56	u	85
9	57	v	86
:	58	w	87
;	59	x	88
<	60	y	89
=	61	z	90

====problem

Write, in binary code, the ASCII value for the letter "w."

A look at the table tells us that the ASCII Code for "w" is 87; therefore, we write the chart as follows.

```
                    128 64 32 16 8 4 2 1
                      ↑   ↑  ↑  ↑ ↑ ↑ ↑ ↑
    bits: off/on--    0   1  0  1 0 1 1 1
                         64   +16  +4+2+1= 87
```

And the binary code becomes

```
0 1 0 1 0 1 1 1
```

=====problem

What character does the binary code

```
0 0 1 0 0 1 1 1
```

represent?

Working in reverse of the above procedure we have

```
  0   0  1  0 0 1 1 1
  ↑   ↑  ↑  ↑ ↑ ↑ ↑ ↑
128 64 32 16 8 4 2 1
         ↑    ↑ ↑ ↑ ↑
        32     +4+2+1= 39
```

A look at the table shows us that ASCII Code 39 represents the apostrophe (').

12.4 WHAT, THEN, IS the COMAL statement to get these characters as an output?

From our experience in earlier chapters we probably know it is

```
chr$
```

Type the following and note the printout on the screen (indicated in parentheses here).

```
print chr$(42) -- (*)

print chr$(81) -- (q)

print chr$(38) -- (&)

print chr$(35) -- (#)

print chr$(62) -- (>)
```

To demonstrate further, type this command.

```
print chr$(67),chr$(79),chr$(77),chr$(65),chr$(76)
```

Tap the RETURN key and the word

comal

will be printed on the screen.

The following program will allow us to see what some of the characters are for ASCII codes not listed in the table. Be certain to type it exactly as it appears. In line 1060, PRINT CHR$(144) makes certain that the cursor remains black in the printout (some of the ASCII Codes cause the cursor to change color and, as a consequence, the printout to the screen cannot be read).

```
1000 page
1010
1020 // a look at ascii codes from 0 - 255
1030
1040 for looky:=0 to 255 do
1050  print "ascii";looky;" ";"char is"; chr$(looky)
1060  print chr$(144)
1070  for pause:=1 to 200 do null
1080 endfor looky
```

Two of the more interesting Codes were chr$(19) and chr$(147). Run them as separate PRINT commands and note what they do.

Here are some ASCII codes and the action they provide.

CHR$ code	Action
5	white characters
13	return
17	cursor down
18	reverse on
19	home
28	red
29	cursor right
30	green
31	blue
129	orange
144	black
146	reverse off
149	brown
150	light red
151	dark grey
152	grey
153	light green
154	light blue
155	light grey
156	purple
157	cursor left
158	yellow
159	cyan

We can add additional interest to our programs by incorporating them into

the algorithm. If, in throwing a pair of dice, we wish the 7's to be printed in --say, red and in reverse, this program would do it. (Note the commas immediately after the chr$ statements.) The CHR$(146),CHR$(144) turns "off" the reverse and changes the red to black for the ".".

```
1000 page
1010
1020 randomize
1030
1040 for dyce:=1 to 100 do
1050
1060  die1:=rnd(1,6)
1070  die2:=rnd(1,6)
1080  dice:=die1+die2
1090
1100  if dice=7 then
1110   print chr$(18),chr$(28),dice;
1120  else
1130   print chr$(146),chr$(144),".";
1140  endif
1150
1160 endfor dyce
```

RECAP
*****

1 bit equals a binary number --0 or 1.

0 is an "off" position; 1, an "on" position.

It takes 8 bits to make a byte.

A byte is a letter, number, symbol, command, punctuation mark or manufacturer-desiigned symbol.

The ASCII code is a standard for most of the computer-generated commands used by world-wide manufacturers of computer systems.

problems

****

For problems 1 - 7, identify the chr$ code number and the character or command output that would occur in a --PRINT CHR$( )-- statement.

1. 1 0 0 1 1 1 0 0

2. 0 0 0 1 0 0 1 0

3. 0 0 1 0 0 1 1 1

4. 0 1 0 1 1 1 1 0

5. 1 0 0 0 0 0 0 1

6. 1 0 0 1 0 1 1 1

7. 0 0 1 1 1 1 1 1

For problems 8 - 14 take the given numbers and write them in binary code.

8. 127

9. 254

10. 5

11. 40

12. 209

13. 191

14. 172

15. Throw a pair of dice 100 times. If a 7 is tossed, print it in reverse green; otherwise, print a "." in blue. Have the final printout tell, in reverse brown, how many 7's were thrown.

16. Generate 50 random integers from 413 through 719. If the integer is divisible by 3 print it in reverse orange; otherwise, print a blank space ("   "). Tell, in reverse cyan, how many were divisible by 3 and, in reverse light blue, how many integers were not divisible by 3.

17. Throw a pair of dice 100 times. Print the 7's in blue and the 11's in red. The rest of the printout should be a ".". Count the number of 7's and 11's. The summary printout is to tell how many 7's and 11's in reverse blue and red, respectively. (Use CTRL Y to get a white screen and CTRL 1 for a black cursor.)

CHAPTER THIRTEEN

logic operators
****

objectives

At the completion of the chapter the student will be able to:

1. Explain the logic operator AND;
2. Explain the logic operator OR;
3. Explain the logic operator NOT;
4. Write a program using AND; and,
5. Write a program using OR.

```
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
c                                                               c
o  SCAN                                                         o
m  ----                                                         m
a                                                               a
l  As programs become longer, the number of program-structural  l
c    errors tend to increase. RUNning a program  is one method  c
o    of discovering them.  But this has many built-in frustra-  o
m    tions.                                                     m
a                                                               a
l  After typing in a program --and before RUNning it, type the  l
c    command SCAN  and  then tap the RETURN key.  The computer  c
o    will do a prepass of the program and print an  error mes-  o
m    sage for the first structure error found. Once corrected,  m
a    SCAN can be used again for the balance of the program.     a
l                                                               l
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
```

13.1 COMAL SUPPORTS THREE logic operators

AND

OR

NOT

(There is an extension of the first, AND, to AND THEN; and the second, OR, to OR ELSE. Their application is not within the scope of this introductory text.)

13.2 THE SYNTAX FOR AND is

<condition> AND <condition>

Where <condition> is a numerical expression.

The root here is Boolean --FALSE being 0; TRUE being 1. In order for the syntax to be TRUE (1) **both** <condition>s must be met. If EITHER one is not true, then the statement is FALSE. An illustration with a pair of dice might help.

=====problem

Throw a pair of dice 100 times. If an even number is tossed and the number is greater than 5, print it; otherwise, print a ".". Count the number of times this happens.

```
1000 page
1010
1020 // program to get # of dice >5 and even
1030
1040 count:=0
1050
1060 randomize
1070
1080 for dyce:=1 to 100 do
1090
1100  die1:=rnd(1,6)
1110  die2:=rnd(1,6)
1120  dice:=die1+die2
1130
1140  if dice>5 AND dice mod 2=0 then
1150   print dice;
1160   count:+1
1170  else
1180   print ".";
1190  endif
1200
1210 endfor dyce
1220 print
1230 print "# of dice >5 and even =";count
```

LINE 1140. If a 4, as an example, is tossed then

dice>5

is false AND THE **ELSE** CONDITION OF THE IF/THEN IS EXECUTED.

If a 7 is tossed, the first part of the logic statement is TRUE and the computer now evaluates the second portion of the logic operator

and dice mod 2=0

Because 7 mod 2 will give a remainder of 1, it is not true and, again, the **ELSE** is executed. Only 6, 8, 10 and 12 will satisfy both conditions as set forth in 1140.

13.3 THE SYNTAX FOR OR is

<condition> OR <condition>

Where <condition> is a numerical expression.

In this instance only one of the two <condition>s has to be true in order for the statement to be a "1". **Both** must be wrong to be FALSE (0).

====problem

Throw a pair of dice 100 times. If an even number is tossed **or** the number >5, print it; otherwise, print a ".". Count the number of times this happens.

```
1000 page
1010
1020 // program to # of dice >5 or even
1030
1040 count:=0
1050
1060 randomize
1070
1080 for dyce:=1 to 100 do
1090
1100  die1:=rnd(1,6)
1110  die2:=rnd(1,6)
1120  dice:=die1+die2
1130
1140  if dice>5 OR dice mod 2=0 then
1150   print dice;
1160   count:+1
1170  else
1180   print ".";
1190  endif
1200
1210 endfor dyce
1220 print
1230 print "# of dice >5 or even =";count
```

LINE 1140. If a 4, as an example, is tossed then

```
dice>5
```

is false and the second part of the expression is evaluated by the computer.

```
or dice mod 2=0
```

Well, 4 mod 2 does = 0, so the **THEN** part of the IF/THEN is executed and the ELSE is ignored.

If a 5 is tossed then

```
dice>5
```

is false and the second part of the expression is now evaluated.

```
or dice mod 2=0
```

Here, 5 mod 2 returns a remainder of 1 and because this part is **also** FALSE, the **ELSE** is executed. In fact, EVERY toss of the dice, except 3 and 5, satisfies LINE 1140. Whereas, in the first program the number of tosses meeting the requirements of Line 1140, was generally in the 40's, this program has a count often running in the high 80's.

13.4 THE SYNTAX FOR NOT is

```
NOT <condition>
```

We will deal with this logic operator in the chapter on WHILE - DO loops. However, we should note here that **NOT** is the opposite of the <condition>.

```
NOT true is FALSE.
NOT false is TRUE.
```

```
..................................................
.                                                .
. RECAP                                          .
. *****                                          .
.                                                .
. LOGIC OPERATORS                                .
.                                                .
. AND  Both <condition>s must be met before the .
.      computer will evaluate them as TRUE.      .
.                                                .
. OR   If either <condition> is met the  compu- .
.      ter will accept the statement as TRUE.    .
.                                                .
. NOT  NOT false is TRUE; NOT true is false.     .
.                                                .
..................................................
```

problems

****

1. Throw a pair of dice 100 times. Print the 2's and 12's and count each. For the others, print a ".". The final printout should state how many 2's and 12's.

2. Throw a pair of dice 100 times. Print those tosses >4 and <11. Print a "." for the others. Count how many tosses fall into the specified range. Sum those tosses and take an average.

3. Throw a pair of dice 100 times. Print those tosses >2 and <6 or = 11. Print a reverse purple space for the others. Sum the tosses that fit the requirements. Take an average to the NEAREST WHOLE NUMBER.

4. Generate 100 random numbers in the sequence 319 - 947. Print only those divisible 7 or 17. The final printout should state how many numbers there were.

5. Generate 100 random numbers from the sequence 247 - 851. Print those numbers > 436 and divisible by 3. Total them and take an average to the nearest thousandths (.001).

6. Generate 100 random numbers from 1011 - 6319. Print, IN FOUR COLUMNS only those > 2090 and < 3106 or > 4791 and < 6147. Count how many and take an average to the nearest whole number.

7. Generate 50 random numbers from 209 - 817. After each random number is generated, throw a pair of dice. If random numbers > 351 and < 743 can be divided evenly by the total of the dice just thrown, then print the random number and the dice. Have the printout read

number	dice	dividend
300	6	50
516	4	129
660	11	66

if AND is NOT OR OR OR is NOT AND

then OR AND AND are NOT AND OR OR

chapter fourteen

read - data
****

objectives

At the completion of the chapter the student will be able to:

1. Name and demonstrate each of three methods of specifying data within any given program;

2. Name the three "married commands" and their "children";

3. State the factors that determine when READ .. DATA is embedded in a FOR-DO loop;

4. Write programs that have one, two or more numeric variables after a READ statement;

5. Write programs that have mixed variables in the READ statement --i. e., numeric and $tring; and,

6. Write a program that embeds a READ statement in a FOR-DO loop, along with an embedded IF-THEN statement.

```
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
c                                                               c
o  ENDING MESSAGES                                              o
m  ---------------                                              m
a                                                               a
l  After RUNning a program, the computer prints a message like  l
c                                                               c
o            end at 1340                                        o
m                                                               m
a  Customized messages can be  incorporated by a final program  a
l   line similar to                                             l
c                                                               c
o            1340 end "random problem finished"                 o
m                                                               m
a  To eliminate an ending message altogether, type              a
l                                                               l
c            1340 end ""                                        c
o                                                               o
m  No space(s) are needed within the quotation marks.           m
a                                                               a
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
```

14.1 THERE ARE THREE main methods for entering data into a program:

The first is to specify the data within the program itself. We did this when we wrote programs to calculate the areas of several polygons.

```
width:=10
height:=8
```

This is very limited, because we have the frustration of having to EDIT the program each time we change the values for another polygon.

The second is by use of the INPUT statement. We will study this in the next chapter.

A third method is by using READ .. DATA statements --which is the thrust of this chapter.

14.2 COMAL HAS MANY "married commands" --"married" in the sense that if we use one, the other must be there also. A FOR requires a DO; an IF necessitates the THEN. These "married commands" are inseparable. Where one goes, so does the other. They never travel alone!!!

If we write a program with "Mr." FOR, then "Ms." DO must be there also --no exceptions!! If we write a program with "Mr." IF, "Ms." THEN must be there also --no exceptions!!

On occasion, the FOR-DO's will bring along their "child" ENDFOR. The IF-THEN's have several children --"ELSE", "ELIF" and "ENDIF." (The ELIF's are identical siblings --it is not certain how many there are, but as

many as eleven of them have been seen!)

The third "married command" we will meet is

"Mr." and "Ms." READ-DATA

They too are inseparable. If one is in the program, **the other must be also!** The READ-DATA's have no children. They are, however, "foster parents" to RESTORE. Once in awhile they will bring this child along to an algorithm. We will meet their "foster child" in a later chapter.

14.3 **PERHAPS THE BEST** way to introduce the READ - DATA concept is through a simple program. In this instance, we will embed the READ statement in a FOR-DO loop, which is a common practice **when it is known exactly how many items are in the DATA line!**

The DATA line for this program will contain THREE SETS of two numbers each. The first number in the set will refer to the WIDTH of a rectangle; the second, to the HEIGHT. The program will calculate the area of the three rectangles by calling out, one at a time, the two measurements. Right away, we can see that this will save a great deal of time just in EDITing procedures alone.

```
1000 page
1010
1020 // getting area of rectangles
1030 // using read-data statements
1040
1050 for rectangle:=1 to 3 do
1060
1070  read width, height
1080
1090  area:=width*height
1100  print "the area of rectangle";rectangle;"is";area
1110  print
1120
1130 endfor rectangle
1140
1150 data 8,12,6,9,7,11
1160
1170 end "end rectangle program"
```

LINE	COMMENT
1050	The beginning of the FOR-DO loop where the variable, "rectangle", takes on the value of 1, 2 or 3 on each successive trip through the loop.
1070	"Mr." READ of the READ-DATA's.
	In this instance, we have asked READ to come up with two

values --one for the variable, WIDTH; the other, for the variable, HEIGHT.

As soon as the computer sees READ and gets the instruction of what is wanted, it **immediately** seeks out the **DATA** line!! It skips over every other line in the program to find that DATA line. When the computer finds it, it assigns WIDTH the value of 8 and HEIGHT the value of 12.

The next time through the loop, the computer will skip over the 8 and the 12 --it's as though they were erased-- and assigns 6 to the WIDTH and 9 to the HEIGHT.

On the last trip, the third, WIDTH will be 7 and HEIGHT will be 11.

This logic is built into the computer. There is nothing we have to do --it's all part of the ROM design!

1090 Inside the computer, on the first trip, it looks like

area:=8*12

The second trip

area:=6*9

The third trip

area:=7*11

1100 This line assigns the values and appears like the final printout shown below.

1110 "Kicks" the computer down one space for each printout.

1130 Closes the FOR-DO loop.

1150 "Ms." DATA of the READ-DATAs.

Note that the DATA line is outside of the loop. We could have put it before line 1050 if we had wanted.

The final printout is--

```
the area of rectangle 1 is 96

the area of rectangle 2 is 54

the area of rectangle 3 is 77
```

We can vary the printout so that it looks better cosmetically. With the previous program in the computer's memory, type the following lines. (Double check the typing --particularly, the line numbers!)

```
1015 zone 9

1041
1042 print "width","height","area"
1043 print "-----","------","----"
1044

1100 print width,height,area
```

Renum 1000. RUN.

As important as the program design is, it is just as important to have an intelligent printout. Brilliant, innovative programming techniques are nice but the user, the person who is to benefit from the program, must be able to make sense of the final results. Anything that the programmer can do to help make the results more easily understood and helpful is more than just "nice", it is essential.

14.4 IT SHOULD BE understood that READ statements are not restricted to two variables --nor just numerics, for that matter. We could as easily have written a program to call for one piece of data, three pieces of data, or more --and we can mix the data types ($trings and numerics)!

A salesman is selling Garbanzo Buttons. His sales for five consecutive days are 10, 21, 18, 31 and 11 buttons. Here is a program that will print the daily sales and the cumulative sales for the week.

```
1000 page
1010
1020 // record of garbanzo button salesman
1030
1040 zone 7
1050
1060 // print table headings
1070
1080 print "day","sales","cum sales"
1090 print "---","-----","---------"
1100
1110 // initialize variable  cumsales
1120
1130 cumsales:=0
1140
1150 for day:=1 to 5 do
1160  read sales
1170
1180  cumsales:+sales
1190
1200  print day,sales,cumsales
1210
1220 endfor day
1230
1240 data 10,21,18,31,11
1250
1260 end ""
```

To make the point once more: READ statements are embedded in FOR-DO loops when **the exact number of DATA items are known.**

Run the above program. Remember, the DATA line could have been anywhere --though the accepted convention is to put them last. Because the READ line is INSIDE the loop, we can pick up the five values in the data line.

14.5 DATA LINES ALSO do not have to be exclusively numeric. They may include $trings. Remember that $tring variables have to have the "$" tagged on the end of the variable name.

Make the following changes in the Garbanzo Button salesman algorithm (program). First, list it; then retype the lines as given below (starting at the flashing cursor).

```
1160 read dae$,sales
```

We do not want a conflict in variables. If we type daY$ instead of daE$, then the computer becomes confused and there will be the error message, "WRONG TYPE".

```
1200 print dae$,sales,cumsales

1240 data "mon",10,"tue",21,"wed",18,"thu",31,"fri",11
```

The $trings in DATA lines must have quotation marks. (Leave a set off and note the error message printed on the screen.) Now, line 1140 asks the computer to READ a $tring first (dae$) and a numeric value second (sales). Hence, the **DATA** line must contain variables **in that order!**

On the first trip, the computer assigns "mon" to DAE$ and 10 to SALES. LINE 1200 prints

```
mon    10    10
```

The final printout will be

```
day    sales  cum sales
---    -----  ---------

mon    10     10
tue    21     31
wed    18     49
thu    31     80
fri    11     91
```

====problem

Among its work force, a company employs Joe, Ted and Lou. Joe is paid $28 per hour; Ted, $18 per hour; and, Lou, $8 per hour. Time over 40

hours returns time-and-a-half pay. One week, respectively, they worked 21, 60 and 80 hours. Write a program showing their pay for the week.

The final printout should be

```
wrkr  hrs   rate  reg   1 1/2 total
name  wrkd  p/hr  pay   pay    pay
----  ----  ----  ---   ----  -----


joe   21    28    588   0     588
ted   60    10    400   300   700
lou   80    8     320   480   780
```

One solution is given below, but work a personal solution before studying the given answer.

After working out the first solution, change the order of the names so that they are listed as: TED; JOE; and, LOU. With these changes, the TOTAL/PAY for Joe remains correct but he will have an amount under the "1 1/2 pay." column. (We can fix this in the program by adding line 1125 to reinitialize some variable to 0.)

(Another reminder: The following solution does not have the heading, PAGE, nor some of the "cosmetic" programming lines. These, of course, are things to be added to the program.)

```
1000 ZONE 6
1010
1020 PRINT "wrkr","hrs","rate","reg","1 1/2","total"
1030 PRINT "name","wrkd","p/hr","pay","pay"," pay "
1040 PRINT "----","----","----","---","----","----"
1050 PRINT
1060 FOR people:=1 TO 3 DO
1070
1080  READ name$,hours,pay
1090
1100  IF hours=<40 THEN
1110   earnings:=hours*pay
1120   wagesreg:=earnings
1125   wageover:=0
1130  ELSE
1140   wageover:=(hours-40)*(pay+pay/2)
1150   wagesreg:=40*pay
1160   earnings:=wageover+wagesreg
1170  ENDIF
1180
1190  PRINT name$,hours,pay,wagesreg,wageover,earnings
1200
1210 ENDFOR people
1220
1230 DATA "joe",21,28,"ted",60,10,"lou",80,8
```

LINE	COMMENT
1020	The choice of column headings is arbitrary; the completeness of the printout is the choice of the programmer.
1030	The second line so the column headings will be complete.
1050	"Kicks" the printer down one space for "cosmetic" reasons.
1060	The beginning of the FOR-DO loop
1080	The **READ** statement. On the first trip of the loop "joe" is assigned to the $tring variable, name$ 21 is assigned to HOURS 28 is assigned to PAY
1100	We need an IF-THEN statement because hours worked and pay are critical. One rate of pay for 40 hours or less and another for more than 40 hours. Hence, this is a classical BINARY decision --which is easily handled by IF-THEN statements.
1110-20	Pay for up to 40 hours. We are going to call for WAGESREG as a printout on line 1190; therefore, we take the earnings and assign it WAGESREG. As we note, JOE has less than 41 hours, so he is the only one taken care of in this TRUE condition of the IF-THEN.
1140	Only hours over 40 get time and a-half, hence we subtract 40 from the total hours. Pay is increased by 1/2 again (+pay/2).
1150	We must figure out regular wages for those who have more than 40 hours, because they are dealt with only after **ELSE**.
1160	Total earnings for these people includes overtime also.
1170	Ends the IF-THEN statements.
1190	PRINT statement for the various variables all separated by commas, so the ZONE 6 in line 1000 will dictate the format.
1210	Ends the FOR-DO loop.

RECAP
*****

READ statements to take information out of a DATA line must be embedded in a loop.

READ variables may be numeric and/or $tring

READ statements may contain one or more variables.

DATA lines may contain numeric and/or $tring data

problems
****

For the following problems write programs to reproduce and complete the following tables.

1. Print the volume of five cubes, complete with their dimensions.

volumes of cubes

cube	width	depth	height	volume
1	6	4	8	
2	7	9	11	
3	16.01	8.02	9.65	
4	2.03	3.01	7.59	
5	86.54	91.73	72.04	

2. Print the area and circumference of the 5 circles.

circles

	radius	area	circum
circle 1	2.63		
circle 2	13.21		
circle 3	4.005		
circle 4	128		
circle 5	17.96		

3. Print the area, perimeter and altitude of an equilateral triangle. If the side is SIDE, then the following formulas hold.

```
perimeter = 3*side

altitude  = (side*sqr(3))/2

area      = ((side↑2)*sqr(3))/4
```

equilateral triangles

side	perim	altde	area
4			
8			
16.41			
30.09			
45.87			

4. A Garbanzo Button salesman sells the following buttons.

day	sold	cum
mon	18	18
tue	23	
wed	19	
thu	37	
fri	45	

5. He sells the buttons for $3 each. Add two more columns, one to show $ income and the other for cumulative $ income.

day	sold	cum	$inc	$cum
mon	18			
tue	23			
wed	19			
thu	37			
fri	45			

6. His commission is $2 per button. Add a sixth column to the chart in problem 5 which will give the commission for each day.

7. People who work for the Jose Maldez Coffee Pickin' Corporation get $6 per sack of beans picked. If they pick more than 2 sacks a day, they get 50% more per sack for those above the minimum of 2. The following is a partial list of workers and the sacks picked for one day.

wrkr name	sacks pickd	base wage	xtra wage	total wages
jose	2			
jack	4			
jill	5			
john	3			
joan	7			

8. To increase production an additional bonus of $2/sack is given only for sacks turned in above 5. The next day the tally sheet showed

wrkr name	sacks pickd	base wage	xtra wage	bons wage	total wages
jose	4				
jack	6				
jill	8				
john	5				
joan	9				

9. The cost of phone calls (per minute) vary according to distance. If the call is up to 200 miles, the cost is .13; 201 - 500 miles, .18; and, from 501, .22.

The following phone calls were placed. Compute the costs.

name	dist	time	cost
mark	186	12	
mary	205	18	
merv	750	25	
matt	400	41	
myra	850	36	

10. A furniture store is having a SELLATHON. There is a 20% discount if the item purchased costs $500 or less and a 40% discount if it costs more. The state sales tax is 5% and is figured on the cost of the furniture after the discount. Items for sale and their cost are given in the table. (The first line is completed as an example.)

furn item	orig cost	perc disc	new cost	amt tax	final cost
sofa	500	.20	400	20	420
bed	800				
table	100				
stool	200				
clock	900				

11. Curtain rods are made 1 meter in length. Any variance in length is to be dealt with by the workers, as follows

a. If they are within + or -1 centimeter of the specified length they are to--

"certify o.k."

b. If they are from 2 through 10 centimeters too short they are to--

"weld on piece"
"polish joint"

c. If they are too long they are to--

"cut off piece"
"buff the end"

d. If the rods are > 10 centimeters too short they are:

"reject rod"

Design an algorithm that will take the lengths of the rods --from the following DATA line-- and place them in a chart:

length	disposition
90	weld on piece; polish joint
85	reject rod
99	certify o.k.

et cetera.

```
DATA 90,85,99,110,105,100,101,89,107,102
```

12. Redo problem number 1 so the results are integer answers.

13. Redo problem number 2 so the results are integer answers.

14. Redo problem number 3 so the results are integer answers.

answers
****

1. Volumes =

192
693
1239.06193
46.377177
571876.158

2. Area and circumference =

21.7300822	16.5247774
548.220799	83.0008779
50.3912247	25.1641571
51471.854	804.24772
1013.35715	112.84008

3. perimeter, altitude and area =

12	3.46410162	6.92820323
24	6.92820323	27.7128129
49.23	14.2114769	116.605168
90.27	26.0587044	392.053208
137.61	39.7245853	911.083364

4. cum totals are

18
41
60
97
142

5.

cum	$inc	$cum
18	54	54
41	69	123
60	57	180
97	111	291
142	135	426

6. The 3rd, 4th and 5th columns are as shown in 5; the 6th column is

com

36
46
38
74
90

7.

base wage	xtra wage	total wages
12	0	12
12	18	30
12	27	39
12	9	21
12	45	57

8.

base wage	xtra wage	bons wage	total wages
12	18	0	30
12	36	2	50
12	54	6	72
12	27	0	39
12	63	8	83

9.

cost
1.56
3.24
5.50
7.38
7.92

10.

perc disc	new cost	amt tax	final cost
.20	400	20	420
.40	480	24	504
.20	80	4	84
.20	160	8	168
.40	540	27	567

11.

length	disposition
90	weld on piece: polish joint
85	reject rod
99	certify o.k.
110	cut off piece; buff the end
105	cut off piece; buff the end
100	certify o.k.
101	certify o.k.
89	reject rod
107	cut off piece; buff the rod
102	cut off piece; buff the rod

CHAPTER FIFTEEN

input - numerics and $trings

objectives

At the completion of the chapter the student will be able to:

1. Write programs calling for various types of INPUT statements:

   1.1 Numeric;

   1.2 Multiple numerics in the same statement;

   1.3 $tring;

   1.4 Multiple $trings in the same statement; and,

   1.5 Mixed numeric and $trings in the same statement.

2. Write a program which uses a numeric INPUT variable to specify the length of a FOR-DO loop;

3. Embed an INPUT statement in a FOR-DO loop;

4. Write a program to control where, on the screen, a Users response to an INPUT request is to be placed;

5. Write a program to control the length on an INPUT request; and,

6. Define what is meant by PROTECTED FIELD for an INPUT statement.

```
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
c                                                               c
o  VERIFY                                                       o
m  ------                                                       m
a                                                               a
l  VERIFY is a command that can be used after a SAVE  command.  l
c                                                               c
o          save "trapezoidbase"                                 o
m                                                               m
a    followed by                                                a
l                                                               l
c          verify "trapezoidbase"                               c
o                                                               o
m    will yield an error message if the program saved  to  the  m
a    disc ("trapezoidbase", here) differs in ANY  respect from  a
l    the same program in the computer.                          l
c                                                               c
o  As programs become longer and more complex this one command  o
m    assumes more and more importance.                          m
a                                                               a
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
```

15.1

INPUT IS ONE of the three main means of entering data into a program. Of the three, **INPUT** is the one the user has the most control over.

It is, however, subject to more errors than the other two. In fact, if it is at all possible, READ - DATA should be used in place of INPUT. Not only is the information readily available for review, but misinformation is not as likely to occur.

Nevertheless, INPUT is a powerful command and can be used by the programmer to control the progress or termination of a program.

15.2

WE WILL BEGIN with INPUTting numerics and demonstrate this in a very elementary program.

```
1000 // simple example of input statement
1010
1020 // finding area of rectangle
1030
1040 print "give me dimensions of a rectangle"
1045 print
1050
1060 input "length? ":length
1070
1080 input "height? ":height
1090
1100 // formula for area
1110
1120 area:=length*height
1130
1135 print
1140 print "area of the rectangle is";area
```

line	commentary
1040	This is a PRINT statement that helps the user to know what the outcome will be. If is often called a prompt. ---AT ALL TIMES, THE PROGRAMMER SHOULD GIVE AS MANY PROMPTS AS POSSIBLE---
1045	"Kicks" the printer down one space so that there will be a separation between the print prompt and the print coming up in line 1060.
1060	The first INPUT statement. Please note that the word LENGTH is enclosed in quotation marks, so it will be printed on the screen. The question mark is included as another helpful prompt for the user. Also observe the space before the last quotation mark. Again, this is helpful to the user. When he types in a value it will be printed with a space between his response and the $tring "length? " Note the use of the colon (:) to separate the variable from the prompt.
1080	The second INPUT statement.
1100	The area can be calculated by taking the value of the varibles we entered (INPUT).

Enter the program into the computer and RUN it several times with different INPUT values.

15.3 **WE MAY CALL** for more than one variable after the INPUT statement.

A line like

```
input "give measures of cube " :length, depth, height
```

is valid.

In this instance the variables are separated by commas. (Don't forget the space after the last letter of the prompt.) When the program is RUN, we type in one value **and either tap the space bar before entering the second or, if we wish, we can tap RETURN** In either case the cursor will stay on the same line as the prompt. RETURN will give us question marks after the first two values entered --tapping the space bar does not.

15.4 **WE ALSO MAY** INPUT $trings. But remember,

the variable must have the $ tagged on the end of it and they must be DIMensioned. Enter the following four-line program.

```
1000 dim name$ of 20
1010 input "what is your name? ":name$
1020 print
1030 print "you said your name was";name$
```

Like numeric INPUTs, we can have more than one $tring variable for the INPUT statement. Remember to DIMension each one.

15.5 **NUMERIC AND $TRING** variables may be mixed in an INPUT statement.

```
1000 dim name$ of 20
1010 input "please give age and name: ":age,name$
1020 print
1030 print "your age is";age
1040 print
1050 print "your name is";name$
```

Though we had two variables on the same line --numeric and $tring (in that order)-- we chose to put the results on two separate lines with a space (line 1040 print) between them.

15.6 **NUMERIC INPUTS CAN** be used as a control variable in a FOR-DO loop. Consider this program.

```
1000 total:=0
1010
1020 input "how many numbers do you want to add? ":num
1030
1040 for add:=1 to num do
1050
1060 input "enter a number":figure
1070 total:+figure
1080
1090 print "the total now is ";total
1100
1110 endfor add
```

Enter the program and RUN it a few times. Change the value of **NUM**.

15.7 **'(WE) CAN PUT** the cursor anywhere on the screen with the CURSOR command. The lines are numbered 1-25 with the top line as line 1. The columns are from 1 thru 40. To place the cursor at position 9 of line 5 (we) would use the following:

```
CURSOR 5,9
```

"So far so good. But there is more. What if (we) want to move the cursor to line 8 but stay in the same column (position)? And (we) don't

know what position that is? COMAL can handle it. If (we) specify a row or position of 0 that means keep the current value. So the way to move to line 8 without changing position would be:

```
CURSOR 8,0
```

"The cursor positioning is also expanded to the INPUT and PRINT statements via the AT keyword. This is very useful. INPUT AT is nice when used with a screen layout for data input. For example:

```
INPUT AT 10,1:"name: ": name$
```

"This prints the prompt "name: " starting at row 10 column 1 and then waits for a reply to be assigned to the variable NAME$. Also, the COMAL Cartridge provides a protected INPUT FIELD for every INPUT statement. . . . Keys that don't make any sense as a reply to an INPUT request are simply ignored -- or even redefined. For example, CURSOR DOWN has no meaning to an INPUT request, so it is ignored. So is CURSOR UP, REVERSE ON and REVERSE OFF. In BASIC and most other programming languages it takes lines and lines of code to set up a protected INPUT field. In COMAL (we) get it automatically.

"While waiting for a reply to an INPUT request, COMAL ignores irrelevant keys (like CURSOR UP) and redefines HOME to mean go back to the first position in the INPUT FIELD. CLEAR SCREEN is redefined to mean clear only the current INPUT field, and then go back to the first position in that field. This is very nice. (We) can now have impressive programs with no extra work whatsoever.

"Also, keep in mind that COMAL will set up a protected INPUT field that goes from the current cursor position to the end of that line --- unless (we) specify otherwise. This is important. If (we) use this default INPUT field length, (we) cannot continue a reply from the end of one line to the next line. To do that (we) MUST specify the field length! And how do (we) do that . . .?

"Easy! (We) do it with an INPUT AT statement -- and (we) just saw one just above. But, in addition to specifying the line and position after the AT, (we) can also specify the field length. Now this can be used when (we) know how many characters to expect, or when (we) want to automatically limit the reply. For example:

```
INPUT AT 5,1,7:"Phone: ":phone

INPUT AT 10,1,10:"Name: ":name$
```

"The first example can be used to prompt for a local phone number (which is always 7 digits) with the prompt "Phone: " starting at line 5 position 1. The second example requests the users name, and limits it automatically to 10 characters. If the user tries to type past the limit, all extra keys are overprinted on the final position of the field.

"One more special note: if (we) specify an INPUT field length of 0, COMAL will only accept the RETURN key as a reply (perfect for that HIT RETURN TO CONTINUE place in (our) program) (1)."

RECAP
*****

INPUTs can be numeric or $tring.

INPUT requests can be single or multiple.

INPUT requests can be both numeric and $tring on the same line.

BOTH the placement on the screen and the length of the reply can be specified in the program on the INPUT line.

problems
****

commentary on problems

As noted in the chapter, READ .. DATA is preferred to INPUT. Go back to the previous chapter and rewrite the programs for problems 2, 6, 9, 10 and 11. Remove the READ .. DATA lines and in their place use INPUT --this will include both $tring as well as numeric INPUTs.

For problem 2, we will be asking for an INPUT value for the radius; the INPUT line may look like this

```
INPUT "what is radius? ":radius
```

If we use this INPUT statement, then the two formulas will be

```
area:=pi*radius↑2

circum:=2*pi*radius
```

in our program.

Because we have not covered ARRAYS as yet, we cannot reproduce the completed tables EXACTLY AS SHOWN in the previous chapter. When using INPUT we would have to STORE the values --this we can do in ARRAYS-- but having no method for doing that at present, the best we can do is to get a printout one line at a time.

Here is one possible program for problem 5 to use as a guide.

```
1000 dim day$ of 3
1010
1020 page
1030
1040 cumsold:=0;cumincome:=0
1050
1060 zone 7
1070
1080 for button:=1 to 5 do
1090
1100 input at 10,1,3:"what day? ":day$
1110 input at 11,1,3:"how many sold? ":sold
1120 cumsold:+sold
1130 income:=sold*3
1140 cumincome:+income
1150 page
1160 print "day","sold","cum","$inc","$cum"
1170 print "---","----","---","----","----"
1180 print
1190 print day$,sold,cumsold,income,cumincome
1200
1210 endfor button
1220 end ""
```

line	commentary
1040	Note that **two** variables are initialized on the same line! They are separated by a semi-colon.
1100	The use of input at 10,1,4 will ask for the INPUT 10 lines DOWN the screen and starting at the LEFT EDGE of the screen. The "3" restricts the input to no more than 3 characters; this way, if we forget and try to type "monday", only the "mon" will be printed. This will keep the printout uniform.
1110	The "11" assures us that the next request for an INPUT will be **one line below** the previous request. "3" restricts the INPUT for the number sold.
1120	Summation statement for the third column.
1130	Formula to get the amount requested for the fourth column.
1140	Summation statement for the fifth column.

1150 Clears screen to avoid a possible overprint problem should an input have less digits than a current printout.

1160-80 Because we are not able to store the INPUT values, we must check our figures with the answers, line by line, as they are printed on the screen.

Therefore, it will help to print the headings after the INPUT values are processed by the program. This is why they are put within the FOR-DO loop and, as a consesquence, are reproduced five different times.

1190 Prints the variables.

For problems 1 - 5 put the column headings inside the FOR-DO loop (as shown in the sample program above); i.e., do not request them as an INPUT statement.

1. Do problem 2, Chapter 14 as an INPUT program for everything but the three column headings.

2. Do problem 6, Chapter 14 as an INPUT program for everything but the six column headings.

3. Do problem 9, Chapter 14 as an INPUT program for everything but the four column headings.

4. Do problem 10, Chapter 14 as an INPUT program for everything but the six column headings.

5. Do problem 11, Chapter 14 as an INPUT program for everything but the two column headings.

6. Write a program employing two INPUT statements. The first is to give the upper limit of a FOR-DO loop; the second is to be embedded in the loop and request the INPUT of the following real numbers:

   34.1, 45.6, 110.003, 17.0, .0106, 1.0005, 78.96

   The running printout on the screen is to read

   # you entered ----

   the sum is now ----

7. Write a program that will convert an INPUT of feet into miles.

8. Write a program that will convert an INPUT of miles into feet and inches.

9. Write a program that will convert an INPUT of money less than $1,

into how many quarters, dimes, nickels and pennies are needed. (Hint: use INT function.)

(Example:)

amount = 93 cents

quarters 3
dimes 1
nickles 1
pennies 3

10. Write a program that will accept the INPUT of an area and will give, centered to the screen

    10.1 The length of the side of a square needed to equal that area; and,

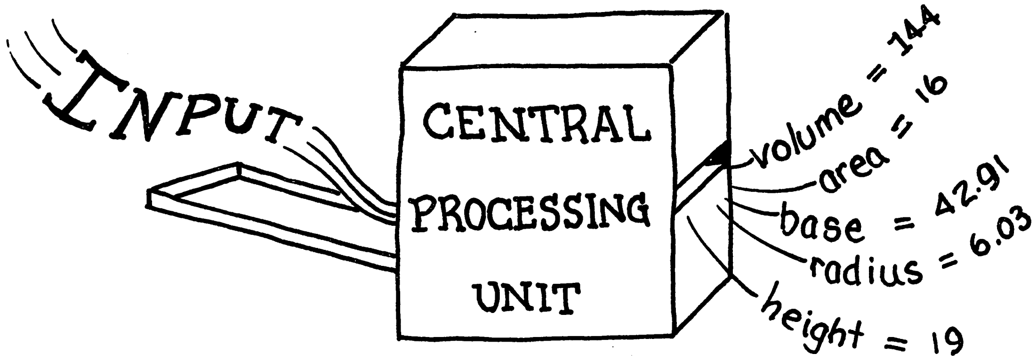
    10.1 The length of the radius of a circle needed to equal that area.

The printout should take the form

area	side	radius
25	5	2.82094792

****

1. Lindsay, Len, COMAL TODAY, #6, COMAL Users Group, Madison, WI.



chapter sixteen

repeat - until
****

objectives

At the completion of the chapter the student will be able to:

1. Name the four "married commands" in COMAL;

2. Explain why the body of a REPEAT-UNTIL statement is executed at least once;

3. State the difference between "+" between two $trings as opposed to the "+" between two numbers; and,

4. Write a program containing a REPEAT-UNTIL statement where the UNTIL evaluation (for true) is a numeric or a $tring.

----

comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal

SPECIFYING PLACEMENT OF MERGE

A SEQuential file can be MERGEd into an existing program in the computer by specifying the beginning line number and subsequent line numbers. The command is similar to

```
merge 1000,5 "title.1"
```

where the "1000" and the "5" can be any numbers the programmer wishes, and "title.1" is any sequential file.

Be careful not to pick numbers that will overwrite existing program lines.

comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal

16.1 WE WILL NOW introduce a fourth "married" command:

'Mr.' and 'Ms.' REPEAT-UNTIL

As the other couples, these too are inseparable! Where one goes, the other must also!!

The general format is

```
REPEAT
 statement 1
 statement 2
 .......
 et cetera
UNTIL (some true condition is met)
```

The classic program illustrating a REPEAT..UNTIL loop is having the computer generate a number and then we try to guess what it is.

```
1000 PAGE
1010
1020 number:=RND(1,100)
1030
1040 REPEAT
1050
1060  INPUT "what number do I have? ":guess
1070  IF guess>number THEN PRINT "too high"
1080  IF guess<number THEN PRINT "too low"
1090
1100 UNTIL guess=number
1110
1120 PRINT "you got it!"
1130 END "end of number game"
```

The statements inside the REPEAT-UNTIL loop are **continually** executed until the condition that is set up after UNTIL becomes true.

In this program, the loop executes the INPUT each time and also one of the IF-THEN statements. When GUESS does = NUMBER, the condition established in 1100 is met **AND** the program falls through to 1120 and then 1130.

As can be seen, a REPEAT-UNTIL loop **must be executed at least once** because the condition is checked **AFTER** the execution of the body.

====problem

Write a program to print, at random, the first three letters of the alphabet until the order is "abc". Keep a count of how many times it takes.

```
1000 DIM let1$ OF 1, let2$ OF 1, let3$ OF 1, alph$ OF 3
1010
1020 PAGE
1030
1040 count:=0
1050
1060 REPEAT
1070  count:+1
1080  let1$:=CHR$(RND(65,67))
1090  let2$:=CHR$(RND(65,67))
1100  let3$:=CHR$(RND(65,67))
1110  alph$:=let1$+let2$+let3$
1120  PRINT count;alph$
1130 UNTIL alph$="abc"
1140
1150 END "finished with letter program"
```

line	commentary
1000	DIMensioning all $trings
1040	Inititalize the counting loop
1070	Counting loop
1080	The first three letters of the alphabet have chr$ codes of 65, 66 and 67. This line generates one of those three values.
1090	Same as 1060.
1100	Same as 1060.
1110	alph$ is replaced by the letters generated at random in lines 1080-1100. The "+" sign means "join to" here --not add, in the sense of 6 + 5.
1120	As count increases on each trip through, its value is printed first and is followed by alph$.
1130	Until "abc" is generated, the body of the loop is FALSE. When it is generated, UNTIL will evaluate it TRUE and the program will fall through to line 1140 and line 1150.
1150	Our message, instead of END AT 1150

Enter the program into the computer. Change the condition on 1130 to "ccc" or "cac" --or some other statement.

We can nest (embed) this program in a FOR-DO loop and run it --say, 10 times. We could then take an average of how many times the random

letters have to be generated in order to get the desired output.

====Let's restate the problem.

Write a program to print, at random, the first three letters of the alphabet until the order is "abc". Run the program 10 times, and strike an average for the ten runs.

```
1000 DIM let1$ OF 1,let2$ OF 1,let3$ OF 1,alph$ OF 3
1010
1020 total:=0
1030
1040 FOR letters:=1 TO 10 DO
1050
1060  PAGE
1070
1080  count:=0
1090
1100   REPEAT
1110    count:+1
1120    let1$:=CHR$(RND(65,67))
1130    let2$:=CHR$(RND(65,67))
1140    let3$:=CHR$(RND(65,67))
1150    word$:=let1$+let2$+let3$
1160    PRINT count;word$
1170   UNTIL word$="abc"
1180
1190  total:+count
1200
1210 ENDFOR LETTERS
1220
1230 PRINT "ave # of throws= ";total/10
1240
1250 END "finished with letter program"
```

line	commentary
1000	All input $trings should be DIMensioned
1020	TOTAL is initialized OUTSIDE(!) the loop, so that it will accumulate the value of the COUNT statement found in line 1080 --which, as we note, is **inside** the REPEAT - UNTIL loop.
1040	The beginning of the FOR-DO loop, which will make 10 trips.
1060	Every time a new "trip" is made within the FOR-DO loop --and there are 10 of them-- this will clear the screen for the new printout.
1080	We must set the COUNT back to 0, after it has been accumulating inside of REPEAT-UNTIL. If we do not, then COUNT will increase in value just as TOTAL does. Note that

	COUNT is set back to 0 INSIDE the FOR-DO loop, but OUTSIDE the REPEAT-UNTIL loop.
1120-1150	See the discussion on the previous program for comments on these lines.
1190	TOTAL summation loop. After "abc" is generated, then the value of COUNT in line 1110 is added to TOTAL **before** it --count-- is set back to 0 in line 1080.
1210	The termination of the FOR-DO loop.
1230	Gives the average number of throws to get "abc" over a ten-run trial.
1250	See earlier comments.

Enter the program and RUN it. Try different letters on line 1170. Will the average increase or decrease if we RUN the program more times (line 1040)? To that end--

EDIT line 1000. Insert

```
let4$ of 1
```

and change word$ to read

```
word$ of 4
```

Add line 1145

```
1145 let4$:=chr$(rnd(65,67))
```

change 1170 to read

```
1170 until word$="abcc"
```

or any other combination of four letters. How much does a fourth letter add to the average?

====problem

Throw a pair of dice until die2 is 2 more than die1. Note how many throws.

```
1000 PAGE
1010
1020 RANDOMIZE
1030
1040 count:=0
1050
1060 REPEAT
1070
1080  die1= RND(1,6)
```

```
1090   die2:=RND(1,6)
1110
1120   count:+1
1130
1140   PRINT "throw";count;"die1 =";die1;"die2 =";die2
1150
1160 UNTIL die2=die1+2
1170 PRINT
1180 PRINT "number of throws =";count
```

Run the program a few times and then take time out to study the algorithm in detail. It is direct and there should be no difficulty in following it.

```
...................................................
.                                                 .
.  RECAP                                          .
.  *****                                          .
.                                                 .
.  A REPEAT-UNTIL loop is executed at least once  .
.                                                 .
.  "+" in $trings means "join to"                 .
.                                                 .
.  A REPEAT-UNTIL loop terminates when the UNTIL  .
.      evaluates a <condition> as TRUE            .
.                                                 .
...................................................
```

problems
****

All problems should have a printout to the screen.

1. Throw a pair of dice until 11 is generated.
2. Throw a pair of dice until die1 and die2 are the same.
3. Throw a pair of dice until die1 = 5 and die2 = 3.
4. Throw a pair of dice until their cumulative sum is equal to or greater than 100. Tell how many throws it took.
5. Generate letters of the alphabet until a "q" is printed. Tell how many letters were generated BEFORE it appeared.
6. Run the previous problem 10 times and strike an average.

7. Generate the letters ghi until "hig"appears. Note how many times before it happens.

8. Do the previous problem (7), but run it in a FOR-DO loop 25 times. Strike an average. Be certain to clear the screen before each new run of letters.

9. Generate the first five letters of the alphabet. Generate the last five letters of the alphabet. Print them until "az" appears. How many times?

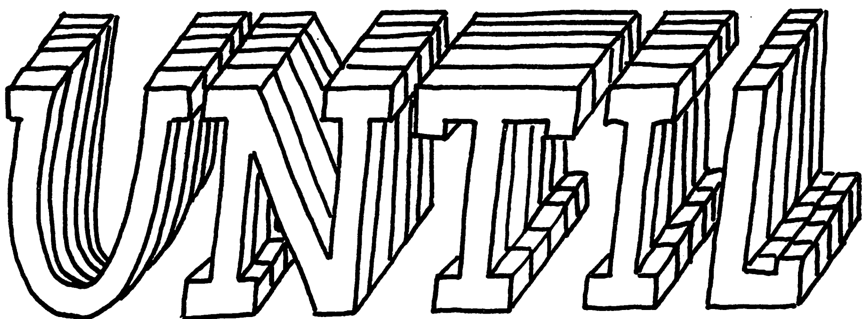
10. Throw a pair of dice until a 7 or an 11 is thrown. How many times?

11. Do problem ten 25 times and strike an average.

12. (Print the following in two columns, one headed "JACK" and the other "JILL." Be certain to round off to the nearest penny. See earlier chapter for discussion and help.)

    If Jack earns 5% (compounded annually) on $5000 and Jill earns 7% (compounded annually) on $4000, in how many years will Jill have more money than Jack?

    (The answer is 12 years.)



chapter seventeen

nested loops
****

objectives

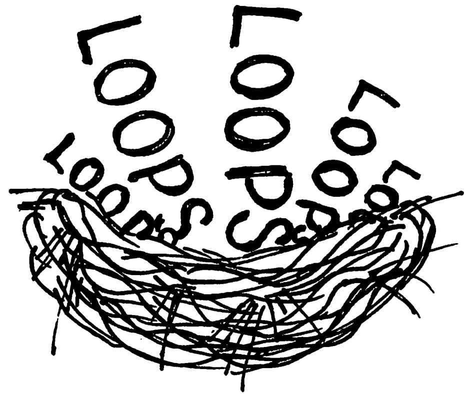
At the completion of the chapter the student will be able to:

1. Name the original developer of COMAL;
2. Write a random expression to simulate the "flipping" of a coin;
3. Write a program which involves the "flipping" of a coin a given number of times, and for a given number of "games";
4. Write a program which involves NESTED FOR-DO loops;
5. Write a random expression that will generate numbers greater than zero (0), but less than one (1); and,
6. Use the CURSOR command to place a printout anywhere on the screen.

```
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
c                                                               c
o  LISTing WITH F6                                              o
m  ---------------                                              m
a                                                               a
l  The current  program in memory can be LISTed to the screen   l
c    by pressing the SHIFT key, tapping f5 and following that   c
o    with a RETURN.                                             o
m                                                               m
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
```

17.1 AS WE HAVE progressed through these chapters we have introduced here and there, some of the powerful built-in features of the COMAL cartridge. Some of the commands we might have wished we had had earlier. On the other hand, too much too soon would have led to much confusion; further, our appreciation of these fea-

tures is enhanced because we worked with the more traditional approaches to programming. In other languages (Pascal, COBOL, FORTRAN, et cetera) we will be "stuck" with the "old ways" of having to do everything for ourselves. In the development of COMAL, the needs of the programmer were first! Indeed, it was Borge Christensen's frustration with BASIC in the late 60's that led him to develop COMAL. Introduced to the Danish educational program in the early 70's, it has since been refined and enhanced to the point where it has been adopted by several European school systems and, from all indications, might soon take its place in the American schools.



17.2 **ACTUALLY, WE ALREADY** have been using nested (embedded) loops. In the last chapter we nested a REPEAT-UNTIL loop within a FOR-DO loop (problem 6). In this chapter we will nest FOR-DOs within FOR-DOs.

Consider this request. We would like to make three copies of a table of squares from 1 through 10.

One method is to write a simple FOR-DO loop and print the table; having done that once, we could RUN it again a second time and then a third. Or better still, we can write a program similar to the following.

(Beginning now, we will follow the accepted practice of leaving off program lines in our discusions. If a particular line is important, we will identify it so that it can be referred to in the commentary.)

```
PAGE

FOR table:=1 TO 3 DO

 ZONE 5

 FOR nums:=1 TO 10 DO
  PRINT nums,nums*nums
 ENDFOR nums

 PRINT
 PRINT
ENDFOR table
```

The inside (nested) loop --**FOR nums**-- will print the numbers 1 through 10 and their squares. The two PRINT statements after the ENDFOR nums, assures there will be two spaces between each table. After the first table is completed the computer leaves the loop, falls through the

two PRINT lines, and then starts a second trip through the **FOR table:= 1 to 3 DO.** Because the OUTSIDE loop calls for three trips, the INSIDE loop will do its task three times.

17.3 COIN FLIPPING IS another task computers can simulate.

Since coins have but two sides, we can use the random statement

```
coin:=rnd(1,2)
```

If a "1" is generated, we will call it "heads"; if a "2", "tails."

To "flip" a coin 100 times and count the number of heads and tails, we could use a program similar to

```
countheads:=0
counttails:=0

page

randomize

for flip:=1 to 100 do

 coin:=rnd(1,2)

 if coin=1 then
  print "h";
  countheads:+1
 else
  print "t";
  counttails:+1
 endif

endfor flip
print

print "# of heads: ";countheads
print "# of tails: ";counttails
```

We have nested the IF-THEN within the FOR-DO. To run this program --say, 10 times, all we have to do is nest THE ABOVE program in another FOR-DO.

Type

```
for tymes:=1 to 10 do
```

and give it a line number smaller than

```
countheads:= 0
```

This is important because (remember an earlier program) we must set the counters back to 0 after they have finished their task and, in this

instance, counting the number of heads and tails in the FIRST 100 FLIPS.

After the last line number, type

```
endfor tymes
```

RUN the program and note that it will do the task 10 separate times. With the addition of an appropriate summation statement --say

```
totalheads:+countheads
```

(don't forget to inititalize it also!) we can strike an average of the number of heads thrown, per 100 flips, for the ten times. We would expect it to approximate 50, of course.

If we make the outside loop REPEAT-UNTIL, we could continue the 100 flips per trial until, on one of the trials the heads were equal to 50 --or any other number we might wish. With another counting loop like

```
trial:+1
```

we could determine how many trials (of 100 flips/trial) it took for that to happen. The algorithm would be similar to the following one.

```
trial:=0

REPEAT

 countheads:=0
 counttails:=0

 PAGE

 FOR flip:=1 TO 100 DO

  coin:=RND(1,2)

  IF coin=1 THEN
   PRINT "h";
   countheads:+1
  ELSE
   PRINT "t";
   counttails+1
  ENDIF
 ENDFOR flip
 PRINT
 PRINT "# of heads: ";countheads
 PRINT "# of tails: ";counttails

 trial:+1
 for pause:=1 to 750 do null

UNTIL countheads=50
PRINT
PRINT "# of runs before exactly 50 heads: ";trial
```

17.4 **TED WILLIAMS WAS** the last major league baseball player to hit for a .400 batting average (.407 in 1941). He played for the Boston Red Sox. (In fact, at the very height of his playing ability, he served 5 years in the armed forces --first in WWII and then in Korea. There are many who believe if he had remained an active player, he would have set hitting records that might never have been broken.)

We are to write a program to simulate "The Splender Splinter" --as they called him-- coming to bat 4 official times per game for 10 games, and to determine how many hits he would get with this batting average.

First, recall that the statement

```
hits:=rnd
```

will generate numbers between .000000000 through .999999999

If he had a batting average of .407, then any random number generated up through .407 would be one that can be counted as a "hit". In other words, random numbers generated from

```
.000000000 through .407000000
```

are "hits".

Hence, a program sequence like

```
IF hits=<.407 THEN
 counthits:+1
 PRINT "hit";
ELSE
 PRINT "...";
ENDIF
```

would be valid.

He comes to bat four times, so we add to the above

```
counthits:=0

FOR atbats:=1 TO 4 DO

 hits:=RND

 IF hits=<.407 THEN
  counthits:+1
  PRINT "hit";
 ELSE
  PRINT "...";
 ENDIF

ENDFOR atbats
```

Now, we want "Thumpin' Theodore" (another of his names) to play in 10

games; and while we are at it, we might just as well figure out his batting average for those games.

```
              PAGE

              totalhits:=0

              FOR games:=1 TO 10 DO

               counthits:=0

               FOR atbats:=1 TO 4 DO

                hits:=RND

                 IF hits=<.407 THEN
                  counthits:+1
                  PRINT "hit";
                 ELSE
                  PRINT " 0 ";
                 ENDIF

               ENDFOR atbats

               PRINT "number of hits=";counthits
     -£-       PRINT
               totalhits:+counthits

     -££-      FOR pause:=1 TO 500 DO null

              ENDFOR games

              batave:=totalhits/40

              PRINT
              PRINT "total number of hits ted had:";totalhits
              PRINT
              PRINT "his batting average was:";batave
              PRINT
              END "end ted williams at bat"
```

-£- Provides a space between each "game" played.

-££- A time delay loop so that we can "look over" each game before the next one is printed.

17.5 THERE ARE OTHER interesting things we can do with nested FOR-DO loops. Enter the following program. After RUNning it a time or two, go back and change some of the control variables. Try some different PRINT statements.

```
              // triangle

              page
```

```
        for star:=1 to 10 do

-£-     for triangle:= 1 to star do

-££-    print "*",
        endfor triangle

        print
        endfor star
        end "triangle terminated"
```

-£- Note the use of the variable "star" FROM the first loop!

-££- The comma is used to eliminate spacing between the *'s.

If the program has been typed correctly, the printout will be

```
*
**
***
****
*****
******
*******
********
*********
**********
```

As the variable star is assigned to the nested FOR-DO loop, it becomes

```
FOR triangle:=1 TO 1 DO
FOR triangle:=1 TO 2 DO

. . . .

FOR triangle:=1 TO 9 DO
FOR triangle:=1 TO 10 DO
```

To move the triangle so that it is away from the left edge of the screen add, immediately after

```
FOR star:= 1 TO 10 DO
```

this line

```
CURSOR star,10
```

The CURSOR command has two numbers after it separated by a comma. The first number always refers to the ROW POSITION, the second, always to the COLUMN POSITION.

By adding

```
CURSOR star,10
```

to this particular program, the triangle starts at the first row (STAR = 1 the first time around) and the 10th column; the second time around, STAR is 2, but the column remains at 10; and so forth.

To move the triangle DOWN as well as over, then type something like this

```
CURSOR star+8,10
```

An interesting printout is

```
CURSOR star,star
```

or

```
CURSOR star+5,star+5
```

RUN the program and "play around" with different combinations of variable and CURSOR calls. That's how to learn new things --and the computer doesn't "chew" us out if we make an error.

```
.................................................
.
.  RECAP                                        .
.  *****                                        .
.                                               .
.  BORG Christensen was the developer of COMAL. .
.                                               .
.  RND(1,2) simulates the flipping of a coin.   .
.                                               .
.  RND generates numbers => 0 and <1            .
.                                               .
.  CURSOR X,Y will place a printout in row X and .
.      column Y                                 .
.                                               .
.................................................
```

problems
****

Write programs that will produce the following print outs.

```
1. **********       2. *          3. *****
   *********           **            *****
   ********            ***           *****
   *******             ****          *****
   ******              *****         *****
   *****               *****         *****
   ****                ****          *****
   ***                 ***           *****
   **                  **            *****
   *                   *             *****
```

4.

```
**************
**************
**************
**************
**************
```

5.

```
*************

*************
*************

*************
*************
*************

*************
*************
*************
*************

*************
*************
*************
*************
*************
```

(Hint: consider three nested loops.)

6.

```
*

 ***

   *****

     *******

       *********
```

7. Place problem 2 centered to the screen. (Hint: Use the cursor command.)

8. Place problem 1 so that it ends in the lower left corner of the screen.

9. Rewrite problem 5 so that it starts with five rows of stars and ends with only one row of stars; put it in the lower right hand corner of the screen.

10. Center the following to the bottom of the screen.

```
         *
        **
       ***
      ****
     *****
    ******
   *******
  ********
 *********
**********
```

11. Redo the Ted Williams problem and have him play 102 games. Give the total number of hits and the final batting average for the 102 games.

12. Modify problem 11 so that it tells in how many games he had four hits, three hits, two hits, one hit and no hits.

The printout should be

4 hits in --- games

3 hits in --- games

2 hits in --- games

1 hit in --- games

0 hits in --- games

13. Flip a coin. When 50 heads are flipped, stop the game and tell how many flips it took.

14. Do problem number 13 for 10 games. Strike an average of the number of flips per game it took to get the 50 heads

15. Flip a coin 100 times for 25 games. Strike an average of the number of heads that were flipped per game, for the 25 games.

16. Throw a pair of dice 10 times. Do this for 20 games. Count the total number of 7's that were thrown and the average number of 7's per game.

17. Throw a pair of dice 20 times. Do this for 25 games. Count the total number of SNAKE EYES that were thrown and the average per game.

18. Throw a pair of dice until 7 appears three times in a row. Do this for 5 games and strike an average of how many throws it takes per game.

19. Flip a coin until heads appears five times in a row. Do this ten times and strike an average per game.

20. Throw a pair dice. Continue throwing them until the number that occurred on the first throw comes up again. (If for example, a 9 is tossed the first time, then keep throwing until a 9 appears again.) Count the number of throws. Do this 15 times. (The first number on each game will probably be different --but for that game keep throwing until that particular number comes up.) Strike an average of how many times it takes to match the first number.

chapter eighteen

while - do
****

objectives

At the completion of the chapter the student will be able to:

1. Embed a **WHILE-DO** in a program to provide a terminus value for the variable in a FOR-DO loop;

2. Define **EOD** and explain how it works;

3. Use a WHILE-DO loop to do the work of a FOR-DO loop; and,

4. Name the five "married commands" and their "children."

```
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
c                                                               c
o  RENAMING A FILE ON THE DISC                                  o
m  ---------------------------                                  m
a                                                               a
l  The command for giving a new name to a file  already on the  l
c    disc, takes the general format of                          c
o                                                               o
m        rename <old file name>,<new file name>                 m
a                                                               a
l        rename "chap 8 pr 6","chap 9 pr 6"                     l
c                                                               c
o    Both files must be on the same disc.                       o
m                                                               m
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
```

18.1 **A FOR-DO LOOP** is terminated after the last number in the variable sequence has been reached.

A REPEAT-UNTIL loop terminates when the condition evaluated by the UNTIL line is satisfied. As observed earlier, this loop is ALWAYS EXECUTED at least ONCE.

In Chapter 11 we made the observation that a FOR-DO loop can be used in READ-DATA statements IF the EXACT NUMBER of data items are known. This is seldom the case. And even if it were, when there is a large number of DATA items, there is an increasing chance of human error creeping in.

18.2

WHILE-DO is the answer. (They have a "child" ENDWHILE, who appears at the end of multi-line statements.) So, "MR." AND "MS." WHILE-DO become our fifth "married" command. Where one goes, the other must be also!

Let's illustrate one usage of the loop in a simple program.

```
              // illustrate while-do

              PAGE

              ZONE 6

              countnums:=0
              sumnums:=0

              PRINT "nums","sqre","cube"

              PRINT

    -£-       WHILE NOT EOD DO
               READ integers
               countnums:+1
              ENDWHILE

    -££-      RESTORE

    -£££-     FOR table:=1 TO countnums DO
               READ integers
               PRINT integers,integers↑2,integers↑3
              ENDFOR table

              DATA 12,13,45,73,54,27
**********
```

-£- EOD --end of data-- is a SYSTEMS FUNCTION word. In this particular context, the line

WHILE NOT EOD DO

says, in the vernacular

"I WILL KEEP READING DATA (NUMBERS IN THIS PROGRAM) UNTIL I FIND THERE ARE NO MORE, AT WHICH TIME I WILL QUIT MY TASK! O.K.?"

For those with a BASIC background, this does away with the need of **FLAG** data items!!

-££- RESTORE is also a systems function word. ONCE the DATA has been READ in a program, it must be RESTOREd so that it can be READ again! And the command RESTORE must precede the READ statement; in this instance, just before the FOR-DO loop that contains another READ statement.

-£££- The WHILE-DO has a counter called COUNTNUMS. We took the final value of the COUNTNUMS and attached it as the ENDING VALUE of the FOR-DO loop.

After all, the computer can do things with an accuracy we cannot match. Even though this program has a DATA line easily counted, the point of the illustration is what counts. (No pun.)

The final printout is

```
nums sqre cube
12   144  1728
13   169  2197
45   2025 91125
73   5329 389017
54   2916 157464
27   729  19683
```

Now, the previous 27-line program, could have been written with only 15 lines by using a WHILE - DO loop!!

```
// illustrating a while-do loop

PAGE

ZONE 6

PRINT"nums","sqre","cube"
PRINT

WHILE NOT EOD DO
 READ nums
 PRINT nums,nums↑2,nums↑3
ENDWHILE

DATA 12,13,45,73,54,27
```

The printout would be the same.

```
nums   sqre   cube

12     144    1728
13     169    2197
45     2025   91125
73     5329   389017
54     2916   157464
```

It is apparent by now that WHILE - DO takes the form of

```
WHILE <conditions> DO
 <statement>
 <statements>
  ......
ENDWHILE
```

There is also a simple one-line version of the WHILE - DO loop that is of use in calling out PROCEDURES. We will be studying those in a later chapter. It takes the form

```
WHILE <condition> DO <statement>
```

It must not have an ENDWHILE attached to it.

18.3 THERE ARE TWO distinct advantages found in the WHILE-DO loop:

1. In problems involving READ/DATA we do not have to count how many items are in the DATA line; and,

2. If the condition stated in the WHILE portion of the loop is NOT MET, then the loop is ignored -- in other words, the loop does NOT have to be executed even once.

====problem

Jack, Jill, John and Joan are salespersons for Garbanzo Buttons, Inc. During one day they had sales of 42, 38, 78 and 59 buttons, respectively. Write a program to summarize this and to give the grand total of their sales.

```
// garbanzo button sales

page

totalbuttons:=0

zone 9

print "person","# sold"
print "------","------"
print

while not eod do

read name$,sales
totalbuttons:+sales

print name$,sales
print
```

```
endwhile

print
print "total sales for day";totalbuttons

data "jack",42,"jill",38,"john",78,"joan",59
```

RUN the program and study the results. The printout is

```
person    # sold
------    ------

jack      42

jill      38

john      78

joan      59

total sales for day 217
```

```
.................................................
.                                               .
.  RECAP                                        .
.  *****                                        .
.                                               .
.  WHILE - DO loops often may  be  used in place .
.      of FOR - DO loops  when  READ/DATA state- .
.      ments are used to solve a problem.        .
.                                               .
.  IF the <condition> associated  with the WHILE .
.      portion of the loop is not met, the  loop .
.      will NOT BE executed; instead, the compu- .
.      ter will fall through to the next line.   .
.                                               .
.................................................
```

problems
****

In the following, put what is GIVEN in a DATA line. Then, using a WHILE-DO as the primary structure, complete them.

1. Do problem 2, Chapter 14. (Answer to nearest hundredths.

2. Do problem 3, Chapter 14. (Answer to nearest thousandths.)

3. Do problem 7, Chapter 14.

4. Do problem 10, Chapter 14. (Determine the discounts and enter

them as the third item after the price of the furniture.)

```
DATA sofa,500,.20, ....
```

5. Do problem 11, Chapter 14.

6. For the following DATA lines (for "cosmetic" reasons, put each DATA statement on a separate DATA line in the program) count the numbers, give their sum and finally the average --both as an integer and to the nearest tenth.

```
DATA 6,4,5,18,32
DATA 42,15,16,49,-10
DATA 37,-14,-79,-46,82
DATA 73,85,47,62,91
```

(ANS: NUMBER = 20; SUM = 515; INTEGER AVE. = 25; AVE. TO NEAREST TENTH = 25.8)

7. HEATHER has had a good week in selling garbanzo buttons. Her sales are

```
mon  $ 75
tue  $108
wed  $ 92
thu  $112
fri  $309
```

Prepare the following table

```
            heather's sales
            ---------------

day   mon   tue   wed   thu   fri

amt   75    108   92    112   309

cum   75    183   275   387   696
```

(Hint: One solution to this problem is to use RESTORE and two count statements --say, count and kount within two IF/THEN/ELSE/ENDIFs, respectively (these, to get "amt" and "cum" printed). This is a challanging problem without a knowledge of ARRAYS; however, it can be done.

Remember, RESTORE allows the data to be reREAD by the computer.)

chapter nineteen

case - of
****

objectives

At the completion of the chapter the student will be able to:

1. Write into a program the case - of structure with the control expression being either numeric or $tring;

2. State the provision in CASE - OF for trapping errors; and,

3. Embed CASE - OF in a FOR-DO loop to get multiple printouts.

```
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
c                                                               c
o  DISPLAY                                                      o
m  -------                                                      m
a                                                               a
l  DISPLAY will put a program on the screen or a printer WITH-  l
c    OUT any line numbers.                                      c
o                                                               o
m       DISPLAY             displays all lines                  m
a       DISPLAY "LP:"       displays all lines on a printer     a
l       DISPLAY -500        displays lines 0-500                l
c       DISPLAY 8000-       displays lines 8000-9999            c
o       DISPLAY 1000-1500   displays lines 1000-1500            o
m                                                               m
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
```

19.1 **VERY OFTEN CASE** is a better choice than a series of ELIFs, in an IF - THEN structure.

Its general format is

```
CASE  <control expression>  OF
WHEN  <expression list>
   <statements>
OTHERWISE
   <statements>
ENDCASE
```

19.2 **ENTER THE FOLLOWING** program.

```
randomize
die1:=rnd(1,6)

case die1 of

when 1
 print "no fun"
when 2
 print "lost shoe"
when 3
 print "burned tree"
when 4
 print "close door"
when 5
 print "oh, my!"
when 6
 print "glue stixs"
otherwise
 print "something wrong"

endcase
```

LIST the program and carefully study the indentation and what system reserved words are capitalized. RUN it a few times.

If we had to write this same program using IF - THEN, we would see how awkward and time consuming it would be. CASE is a more direct way of handling problems of this type. There are no one-liners for CASE - OF.

Note also, that **'Mr' and 'Ms' Case-Of** always have their "children" (one or more **WHENs** and --always-- **ENDCASE**) with them. (That's why they don't get invited to many parties ... they are forever dragging their children along.)

19.3 **ADD THESE TWO** lines to the above program.

```
for trials:=1 to 10 do   (put this in as the first line)

endfor trials            (put this in as the last line)
```

RUN the program again and note the printouts.

After the random number is generated for die1, the computer checks each of the **WHEN** cases in the structure. If a 5 were generated, it would ignore the first four WHENs and print what it found in

```
when 5
print "oh, my!"
```

Having done that, it would exit from the program.

19.4 THE CONTROL EXPRESSION also can be a $tring.

```
// demonstrates CASE with $tring control

randomize
letter$:=chr$(rnd(65,90)

case letter$ of

when "a","e","i","o","u","y"
 print letter$," is a vowel"
when "b","c","d","f"
 print letter$," is a consonant"
when "g","h","j","k"
 print letter$," is a consonant"
when "l","m","n","p"
 print letter$," is a consonant"
when "q","r","s","t"
 print letter$," is a consonant"
when "v","w","x","z"
 print letter$," is a consonant"
otherwise
 print "check program, error somewhere"

endcase
```

We broke up the rest of the alphabet for cosmetic reasons. If we had wished, we could have listed the balance of the letters with the second "when". And don't overlook the strength of the "otherwise" statement. It is a nice trap to catch errors.

RUN the program a few times. Add a FOR-DO loop, as we did in the previous algorithm.

```
.............................................
.                                           .
.  RECAP                                    .
.  *****                                    .
.                                           .
.  THE general structure of CASE - OF is    .
.                                           .
.        CASE <control expression> OF       .
.        WHEN <expression values>           .
.             <statements>                  .
.        OTHERWISE                          .
.             <statements>                  .
.        ENDCASE                            .
.                                           .
.  THE <control expression> is numeric or $tring .
.                                           .
.............................................
```

problems
****

1. Write a program where the user enters a number and the program prints out what day of the week it represents.

   For an input use,

   ```
   repeat
   input "what day number do you want? ":number
   until number=>1
   ```

   The OTHERWISE should catch a "foolish" input like 6.3 or 456.

2. Do the same for the number of the months of the year (1 = January, 8 = August).

3. The birthstones for the months are

   ```
     january     garnet
    february     amethyst
       march     aquamarine
       april     diamond
         may     emerald
        june     pearl
        july     ruby
      august     peridot
   september     sapphire
     october     opal
    november     topaz
    december     turquoise
   ```

   Using a $tring input as the control expression, write CASE - OF that will give the birthstone for an input of the month.

4. Throw a pair of dice. If it comes up

    2 or 12, print: "snake eyes in the box cars --you lose"

    3, 4 or 5 print: "low mish mash --it's a push"

    6, 7 or 8 print: "you win!"

    9, 10 or 11 print: "high mish mash --you lose."

5. A still popular game is BRICK, PAPER and SCISSORS. It has led to more than one sore wrist (the "winner" getting to slap the "loser" on the wrist with two fingers!!).

   BRICK (a clenched fist) breaks the SCISSORS (two fingers) which, in turn, cuts the PAPER (an open hand). The PAPER wraps the BRICK.

   Write a program to handle the six possibilities: SS; BB; PP; SB; SP; and, PB. The user INPUTS one of his choices (P, B or S) and the computer generates its choice from a random notation.

   Print the winner and tell exactly why. Some of the printouts might be

    paper covers brick; you are winner

    brick breaks scissors; computer wins

    no winner; both the same

   (Hint: Make paper = 1; scissors = 2; brick = 3. Have the input from the player a $tring, even though he will type a 1 or 2 or 3. Have the computer input generated by a CHR$ random expression. Chr$(49) = 1; Chr$(50) = 2; Chr$(51) = 3.)

chapter twenty

procedures

****

objectives

At the completion of the chapter the student will be able to:

1. Define what MCS means;
2. Write a program employing MCS PROCedure names;
3. MERGE seq files into a program;
4. LIST to disc, with the appropriate tag, "l", to distinguish prg from seq files;
5. Write a time-delay PROCedure that can be MERGED into any program;
6. Write a CLEARSCREEN PROCedure in one of two ways

   Either embedding the PAGE command, or

   PRINT CHR$(147);
7. Write a PROCedure that will make a variable LOCAL to that particular PROCedure;
8. Know how to rewrite within a PROCedure, the GLOBAL variable ZONE so that it is applicable only to that PROCedure;
9. Put at any place on the screen, through the use of the CURSOR command, any given character; and,
10. Use the CURSOR command to print a variety of designs and sequences on the screen --including the tracing of outlines, with and without step erasures.

```
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
c                                                               c
o  LIST, EDIT, DISPLAY FOR PROCEDURES                           o
m  ---------------------------------                            m
a                                                               a
l  PROCEDURES may be LISTed,  EDITed or  DISPLAYed  by calling  l
c    them by the name assigned.  For PROC sample, it would be:  c
o                                                               o
m      list sample     will list all lines in the PROCedure     m
a                      named SAMPLE                             a
l                                                               l
c      edit sample     edits lines, one at a time, for the      c
o                      PROCedure named SAMPLE                   o
m                                                               m
a      display sample  displays all lines in the PROCedure      a
l                      named SAMPLE, but without line numbers   l
c                                                               c
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
```

20.1 **THE DISTINCT ADVANTAGE** of structured programming languages is the facility of being able to break a complex program into small parts. Whenever a routine that is embedded in a PROCedure is needed in the algorithm (no matter how many times), it can be "called" by the MAIN CONTROL SEQUENCE (MCS) or even from **within** a PROCedure itself! We will demonstrate this by a ridiculously simple program --a program that would never use PROCedure calls; however, because it will be so direct, we can concentrate on the principle of what is being discussed.

20.2 **FIRST, LET'S REWRITE** our logo to be

```
// select output "lp:"

// ********************
// *   <your  logo>   *
// ********************
// * <identification> *
// ********************

// ..........................
// .. main control sequence ..
// ..........................
```

Re-LIST the logo with the overwrite command

```
list "@0:title.l"
```

Now, with the logo still in the computer, add these lines

```
num1:=8
num2:=3
page
add
```

```
subtract
multiply
divide
powers
page
review
end "end introduction to procedures"
```

Do not RUN!

20.3 CERTAINLY, THE FIRST two lines need no explanation. We have assigned the value of 8 to the variable, num1 and 3 to the variable, num2.

As the computer falls into the third line --page-- the screen will be cleared. When this task is completed, it will return to the MCS and fall through to the next line --add. Because "add" (unlike PAGE) is not a system-reserved word, the computer assumes we are issuing a PROCedure call! Therefore, it will seek throughout the algorithm for a PROCedure named "add" (PROC add). (If there is no PROC add, an error message will be generated to the screen.) When found, it will execute what ever commands are there. The computer will continue to do this for each succeeding line, until it falls through the last PROCedure call --or until it arrives at the system-reserved word END.

OBSERVE!!!! There are no special commands or programming techniques we have to write to make the computer do this! It's all part of the COMAL 80 CARTRIDGE!! --it's built in! FURTHER, it makes little difference *where* the PROCedures are located in the program. And even though we are "calling" them in a specific order, they *do not* have to be entered into the algorithm in that order!

20.4 NEXT, CONTINUE WITH the program in the computer, by adding the following PROCedures.

```
proc add
 print "the sum of ",num1," and ",num2," = ",num1 + num2
 print
 pause
endproc add


proc subtract
 print num 1," - ",num2," = ",num1-num2
 print
 pause
endproc subtract

proc multiply
 print num1," X ",num2," = ",num1*num2
 print
 pause
endproc multiply
```

```
proc divide
 print num1," divided by ",num2," = ",num1/num2
 print
 pause
endproc divide

proc powers
 print num1," raised to ",num2," = ",num1↑num2
 print
 pause
endproc powers

proc review
 add
 subtract
 multiply
 divide
 powers
endproc review

proc pause
 for wait:=1 to 1000 do null
endproc pause
```

RENUM. RUN the program. If the lines have been entered correctly, this is what happens.

```
[screen is cleared]
                        the sum of 8 and 3 = 11
[1 second pause]
                        8 - 3 = 5
[1 second pause]
                        8 X 3 = 24
[1 second pause]
                        8 divided by 3 = 2.66666667
[1 second pause]
                        8 raised to 3 = 512
[1 second pause]

[screen is cleared]
```

And then the sequence is repeated again; only this time, the screen is not cleared after the last printout.

20.5 OBSERVE TWO THINGS in the program.

1. PAUSE is a PROCedure call made from WITHIN ANOTHER PROCedure!

2. The last PROCedure --PROC review-- calls five other PROCedures; ones, which we know, already have been executed before!

20.6 **VERY OFTEN, VARIABLES** within a PROCedure are used elsewhere in the program. DICE, is a good example. We can make variables within a PROCedure LOCAL to that PROCedure, so the computer will not take the value of the variable from there and use it elsewhere. To do this, the programming technique employs the system word **CLOSED** and is written as follows.

```
PROC tally CLOSED

 die1:=RND(1,6)
 die2:=RND(1,6)
 dice:=die1+die2

 IF dice>6 THEN
  PRINT dice;
  PRINT "you win"
 ELSE
  PRINT dice;
  PRINT "you lose"
 ENDIF

ENDPROC tally
```

The variable, DICE, can be used again in another part of the program without causing any difficulties. The jargon is: "It is unknown to other parts of the program."

There are, however, global variables that will "push" right through the "walls" of a PROCedure. **ZONE** is one of them. If we have a ZONE command outside of a PROCedure, and it would force an awkward output for the commands within that PROCedure, then ZONE can be redefined within that sequence. Here is an illustration.

```
          proc stars closed

£          oldzone:=zone

££         zone 3
           for x:=1 to 40 do print "*",

£££        zone oldzone

          endproc stars
```

=====

£ "oldzone" has been given the current value of ZONE

££ a new value (3) has been given to zone

£££ after printing the "*"s, ZONE has been returned to its original value.

20.7 **AT THIS POINT** we should note the

following:

1. Most of what is to be presented is for output to the screen;

2. A few printers might get "hung up" if we try to print graphics; and,

3. Some printers will become "hung up" or will eject a whole sheet if PAGE is "called" for.

(By the way, we might choose to make a PROCedure call for PAGE. It can take either one of these two forms:

```
proc clearscreen
 page
endproc clearscreen
```

or

```
proc clearscreen
 print chr$(147)
endproc clearscreen
```

This means we can use the direct command PAGE and/or, if we prefer, a PROCedure call throughout the algorithm.)

The best way to handle those hard copy RUNs is to // PAGE or, if we are using a PROCedure call, // that. Now, if we are using the latter approach, it does not mean we also have to DELete or even // PROC clearscreen. Even though there is not a "call" for a PROCedure in the MCS (or the "call" has been //), it will cause no program breakdown.

We could have a hundred PROCedure sequences in a program --and we do not have to call for one of them. As long as they are not "called", the computer will ignore them and the program will RUN just as though they were not there.

20.8 **FOR THE BALANCE** of the chapter --and by using screen graphics-- we will be strengthening our grasp on PROCedures and learning, at the same time, how to control the **CURSOR** command. Every exercise is important (sequentially) and none should be skipped or treated casually.

When each is completed, the algorithm itself should be printed --don't, unless specifically stated, try to RUN the program on the printer --all we will want is a hard copy of the program listing. The best way to do this, is to make certain the program is in the computer's memory and then **WITHOUT A PROGRAM LINE** type

```
list "lp:"
```

Further, each progression is going to be presented as an assignment. Because of our development to this point, detailed discussion will be limited. Instructions will be given, but a great deal of its execution will be left to the programmer.

And remember, there is no one way --no ONE "correct" alogorithm-- for the programs. Just be consistent and use PROCedures liberally.

(It should be understood here, the following introduction to PROCedure structures and techniques is basic. An exhaustive discussion is beyond the scope of this introductory text.)

20.9 1. MESSAGE FLASHO.

MERGE the logo.

Establish a

proc message1

as follows

```
proc message1

 print "**********************"
 print "*                    *"
 print "*  put in a message  *"
 print "*                    *"
 print "**********************"

endproc messsage1
```

Establish a

proc message2

(Don't forget the advantage of using the AUTO command.)

It is to have a different message but the box dimensions are to be identical.

Establish a

proc signoff

The same size box.

Put a closing message in it.

Now, write a MCS so that message1 and message2 "blink" off and on. If the design of the PROCedures is correct, the box should appear "fixed," and only the message will change. Be certain to PAUSE between each message. Finally, use PROC signoff. The MCS could be like

```
page
message1
pause
page
message2
pause
page
signoff
page
end ""
```

Vary the MCS sequence to produce a variety of effects. (This is a good way to further understanding.)

20.10 2. ALPHABET.

MERGE the logo.

Enter the following MCS

```
onealpha
onealpha
onealpha
end ""
```

Establish a PROCedure onealpha

```
proc onealpha
 print "i"
endproc onealpha
```

RUN the program

We should get a letter that is similiar to I.

Establish a PROCedure fivealpha

```
proc fivealpha
 print "1111"
endproc fivealpha
```

In the MCS add FIVEALPHA just before END.

Go back to PROC onealpha and change the "i" to "l"

RUN. We should get a letter that is similar to "L."

Do the following letters, adding any ALPHA PROCedures to the existing program as may be necessary. (Remember, even though there may be some alpha PROCedures we do not use, as long as they are not called for in the MCS, our algorithm will be unaffected.) Use capital format for the letter design.

C, E, F, G, H, M, N, Q, U, V, Y, Z.

For each letter, change "alpha" to the letter being designed. RUN and LIST the programs on the printer for the letters G, M, V, Y, Z. What follows is the alogorithm for "Q."

```
// ** main control sequence **

PAGE
q
end ""

PROC q
 fiveqs
 twoqs
 twoqs
 twoqs
 fiveqs
 tailq
 tailqq
ENDPROC q

PROC fiveqs
 print "qqqqq"
ENDPROC fiveqa

PROC twoqs
 print "q   q"
ENDPROC twoqs

PROC tailq
 print "      q"
ENDPROC tailq

RPOC tailqq
 print "       q"
ENDPROC tailqq
```

20.11 3. RUNNING o's.

Merge the logo.

Define a PROCedure OROW which will print a ROW of 30 o's a-cross the screen. Base the PROCedure on the following pseudocode.

```
procedure orow
 initialize count
 repeat
  print an "o"
  increment the count
 until count is at the limit
end procedure orow
```

Translate the above pseudocode into COMAL.

Call for PAUSE from inside the PROCedure. Change the interval to 1 to 150; this slows the printing of the o's.

RUN on the screen. Print a hard copy of the program listing.

20.12 4. SLASH o's.

Design a PROCedure SLASHO which will print a row of 18 o's on a diagonal. The diagonal is to start at the upper left corner and continue towards the lower right. Call for PAUSE to slow down the printout.

Here is one solution. PAGE will be called from a PROCedure to illustrate its use. (PROGRAMS FOR THE BALANCE OF THIS CHAPTER ARE IN THE ANSWERS.)

```
           // ** main control sequence **

           clearscreen
           slasho
           END "slasho finished"

           PROC slasho
  -£-       count:=1
            REPEAT
             pause
  -££-       CURSOR count,count+1
             PRINT "o",
             COUNT:+1
  -£££-     UNTIL count=19
           ENDPROC slasho

           PROC clearscreen
            PAGE
           ENDPROC clearscreen

           PROC pause
            FOR wait:=1 TO 150 DO NULL
           ENDPROC pause
```

*****

-£- If we initialize COUNT at 0, then we are going to have difficulty at the line

```
CURSOR count,count+1
```

The first parameter after CURSOR (count) is the ROW indicator for placing the cursor. There is no zero row, the first is 1.

If we initialized count at 0, the first two times through the REPEAT loop would give ROW as being 0 and 1. The beginning printout for this would be

```
o o
   o
    o
     o
      o
```

Therefore, we initialize COUNT at 1, to avoid the problem.

-££- As we move the CURSOR positioning, the COUNT+1 becomes one greater than COUNT. Translated, it means

row 1 column 2
row 2 column 3
row 3 column 4

and so forth

(CURSOR count,count

would have been just as valid.)

- ££- If we want 18 o's and COUNT is initialized at 1, then we need to terminate the loop at 19 to get those 18 o's.

*****

ALTER the program so that the diagonal of o's runs up the screen from lower left to upper right. To do this, make four changes in the present program and add one line.

CHANGES

1. Initialize COUNT to 19.

2. Change the CURSOR line to

   CURSOR count,0

3. Change count:+1 to count:-1

4. Make the UNTIL read

   UNTIL count=1

ADD

1. Just after SLASHO in the **MCS** add

   CURSOR 21,1

   The reason for this is to place the END message at the bottom of the screen. If we don't put that in, this message will overprint the tail end of the o's. We will be at the top of the screen after printing the

last o, --that's where the CURSOR is-- and that's where the computer will place the END message from the MCS.

To strengthen our understanding of what is happening, take the comma away from the line

```
PRINT "o",
```

and see what the printout is. It is important to go no further until this is clearly understood!

You see, in the cursor command

```
CURSOR count,0
```

as count is decremented (or incremented, in other programs) the ROW where the cursor is, changes --the column will stay the same, because of the 0. HOWEVER, in the line

```
PRINT "o",
```

there is a comma. A comma is a TRAILING punctuation command and says

"In the next PRINT command, place what is to be printed immediately after this."

This means THE CURSOR HAS ALREADY MOVED OVER ONE PLACE to make room for the next character. Therefore, as the ROW changes position (in this instance up the screen) the CURSOR continues to move one more place to the right to make room for the next character(s) to be printed.

Take away the comma and the column will stay the same. That's why the o's went up the left edge of the screen when we removed it.

By the way, to keep the ROWS the same, use the command

```
CURSOR 0,count
```

in this program, that is --because we may have a different command instead of COUNT in another program.

Run both programs on the screen; make hard copies of the algorithms.

20.13 5. BLANK o.

Design a program which moves an o across the screen 35 times and blanks out each previous o as it does. The effect, then, is of a sprite "walking" across the screen.

Merge the logo.

(This will not be mentioned again.)

Initialize count at 2 and continue it through 37.

Add

```
cursor 0,count
```

to the program just before

```
print "o",
```

We will also need to add (somewhere) these two lines

```
cursor 0,count-1
print " ",
```

Be certain to increment COUNT by 1. Why is count initialized at 2? Erase the last o after a two-second pause? (Try it; then see 20.14 for help.)

```
// ** main control sequence **

PAGE
orow
END ""

PROC orow
 count:=2
 REPEAT
  pause
  CURSOR 0,count
  PRINT "o",
  CURSOR 0,count-1
  PRINT " ",
  count:+1
 UNTIL count=37
ENDPROC orow

PROC pause
 FOR wait:=1 TO 100 DO NULL
ENDPROC pause
```

20.14 6. BACK BLANKo.

Design an algorithm similar to BLANKo which, when RUN, will move the o from the right of the screen (position 37) to the left of the screen (position 2). Again, it must erase the last o before printing another. Also, erase the ending o after a two-second pause.

Having done that, combine this algorithm with the previous (20.13). In this way, the o will move across the screen left to right and then move right to left --and then vanish!

One PROCedure to effect the final erasure is

```
PROC erase
 FOR delay:=1 TO 2000 DO NULL
 CURSOR 0,3
 PRINT " "
ENDPROC erase
```

20.15 7. LETTER TRACING

Design another algorithm which, when RUN, will trace an o on the screen moving in the form of the letter Z. To help, here is an algorithm for the letter J --and being erased as it is traced.

```
// ** main control sequence **

PAGE
downo
backo
erase
END "letter j complete"

PROC downo
 count:=2
 REPEAT
  pause
  CURSOR count,15
  PRINT "o"
  CURSOR count-1,15
  PRINT " "
  count:+1
 UNTIL count=12
ENDPROC downo

PROC backo
 kount:=15
 REPEAT
  pause
  CURSOR 11,kount
  PRINT " ",
  CURSOR 11,kount-1
  PRINT "o",
  kount:-1
 UNTIL kount=9
ENDPROC backo

PROC erase
 FOR wait:=1 TO 2000 DO NULL
 PRINT "  "
ENDPROC erase
```

```
PROC pause
 FOR wait:=1 TO 100 DO NULL
ENDPROC pause
```

Design algorithms for the following upper-case letters --first tracing the letter and then tracing the letter but erasing as it goes.

C, L, O, P, S, V

(FOR P, start at the bottom of the stem.)

20.16 8. CHOP TREES - CLEAR STUMPS

For this particular program make certain the keyboard is in upper-case letters.

Define a PROCedure trees which will display 600 "trees" next to each other on the screen.

(The "tree" will be found as an on-board graphic. SHIFT X will print it. It's actually the "club" graphic found in a deck of cards.)

RUN this on the screen to make certain it works. The PAUSE PROCedure inside of the REPEAT-UNTIL loop should be cut to --1 to 25. Once this is working, define another PROCedure CHOPDOWN, which will overprint each "tree" with the letter O.

```
CURSOR 1,1
```

will place the cursor in the upper left corner of the screen. Put this cursor command inside PROCedure chopdown, but just before the REPEAT - UNTIL loop.

Once there

```
PRINT "o"
```

will overprint the "trees" to effect the CHOPDOWN simulation and suggest a "stump."

"Stumps," of course, have to be cleared. Home the cursor again (CURSOR 1,1) within another PROCedure. Call it

```
PROCedure clearstumps
```

In it, use the print command

```
print " ",
```

*****

Design another program that will "plant" the trees starting at 1,1 but will cut them down in REVERSE order.

Instead of using a REPEAT-UNTIL loop inside of the PROCedure chopdown, use nested loops

The first could read

```
FOR kount:=15 TO 1 step -1 DO
```

The second is to use the variable COUNT

The CURSOR command will be

```
cursor kount,count
```

FINALLY, instead of removing all the stumps, design the PROCedure CLEARSTUMPS so that only 75 trees are removed at random.

Here is one line to help.

```
column:=RND(1,40)
```

Remember, there are 40 columns to a screen.

RUN each of the tree programs on the screen and save to disc.

(It might be interesting to fill the "hole" where a stump is removed --say, with a blue square.)

RECAP
*****

THE order of the PROCedure calls has no bearing on the order in which the PROCedures are placed in the algorithm.

ZONE is a global variable.

VARIABLES may be kept local to a PROCedure by attaching the word "CLOSED" to the PROCedure name.

IN the CURSOR command,

CURSOR X,Y

X specifies the ROW and Y the COLUMN

---

problems

****

1. Write a program that will produce the following

```
ooooooooooo
 ooooooooo
  ooooooo
   ooooo
    ooo
     o
```

2. Write a program that will trace a box about in the middle of the screen --9 x 9 is a good size, and will have a series of numbers being printed out, as though it were a one-arm-bandit type of machine. The numbers should run after the box is traced. (Hint: use rnd(abc,xyz) and embed it in a REPEAT-UNTIL LOOP.)

3. The same as problem 2, but be certain the concluding number, after the "spin", is 100. (The CURSOR command will place the spinning numbers in the middle of the box.)

4. Trace the following numbers on the screen. Put all three in the same program. When an integer appears on the screen, have it stay about 3 seconds, disappear, then trace the next --and so forth.

   4, 5, 9

5. Trace a good size plus sign (+) on the screen. Have it

   5.1) remain whole; and then next,

   5.2) erase itself as it goes.

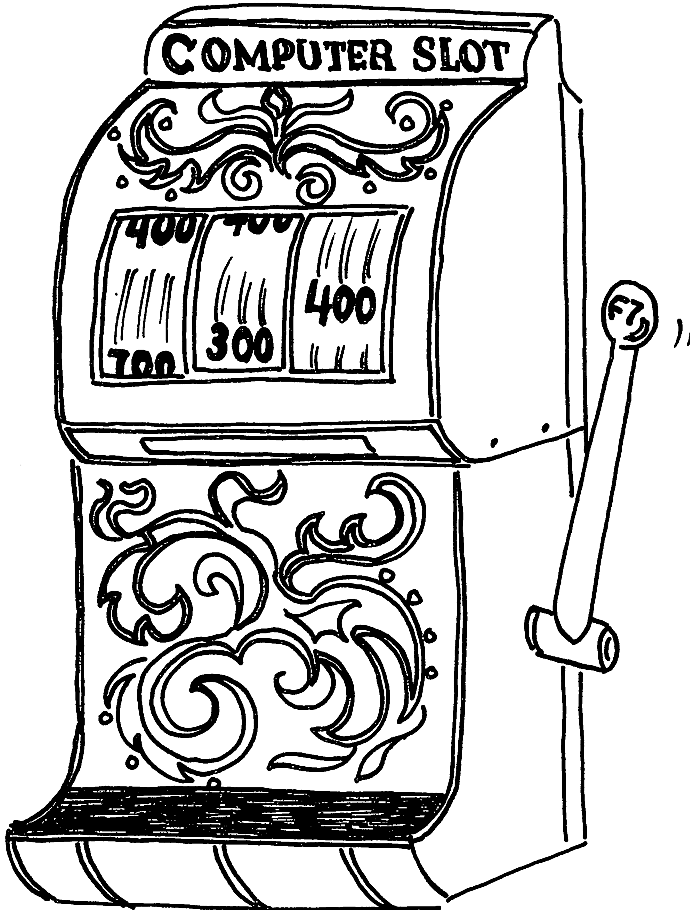
6. Trace a good-sized box-shaped moving van on the screen. Put in some wheels after it is drawn.

7. **ONE-ARM BANDIT.** Draw three boxes across the screen in one line. 9 x 9 is a good size.

   Generate, by use of five distinct variables, five random numbers --each that start and end in three digits. However, at the end of each of these spins, the first is to have a final value of 100, the second, 200 ... the fifth, 500.

   The random numbers that are to "spin" in the middle of the three boxes, are to be placed in any given box by random choice also. At the conclusion of the pulling of the **ARM** (represented by tapping f7) there will be one number in each box. They can be any combination of 100, 200, 300, 400 or 500. Any of these five values can appear 0 to 3 times. **ONE** --and only one-- possibility could be

```
*********   *********   *********
*       *   *       *   *       *
*       *   *       *   *       *
*       *   *       *   *       *
*  100  *   *  300  *   *  300  *
*       *   *       *   *       *
*       *   *       *   *       *
*       *   *       *   *       *
*********   *********   *********
```



chapter twenty-one

arrays
******

objectives

At the completion of the chapter the student will be able to:

1. DIMension a numeric array;
2. Explain the role of subscripts in an array;
3. Explain the difference between the NAME of the array and its subscripted name;
4. Write a program which uses a subscript in a variable or numeric function;
5. Use the proper verbs to store the values of an array;
6. Write a program that can call out certain of the stored values in an array;
7. Write a program that uses a summation statement employing an array notation; and,
8. Write a program using more than one array and be able to manipulate the data in the arrays in accordance with specified outputs.

```
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
c                                                               c
o  SCAN BY f8                                                   o
m  ----------                                                   m
a                                                               a
l  The very  useful command, SCAN, can be activated by tapping  l
c    the f8 key + RETURN.                                       c
o                                                               o
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
```

21.1 CONSIDER A TYPICAL classroom occurrence.

The teacher has recorded the following examination grades (understand, the list is kept brief for ease of illustration.)

65, 43, 82, 98, 86, 73, 91, 63, 90, 79

What he would like is the following information:

The class average;

The number of grades above the average;

The number of grades below the average;

How much each grade is above the average;

How much each grade is below the average; and,

How many grades are in each 10-point interval.

In terms of actual printout

Class average = 77

Number of grades above average = 6

Number of grades below average = 4

EXAM GRADE	POINTS ABOVE	POINTS BELOW
65		12
43		34
82	5	
98	21	
86	9	
73		4
91	14	
63		14
90	13	
79	2	

GRADE INTERVAL	NUMBER OF GRADES
40 - 49	1
50 - 59	0
60 - 69	2
70 - 79	2
80 - 89	2
90 - 99	3

Based on what we now know, we could write such a program --but it would

take 45 to 50 program lines; further, once having written it, the data would be difficult to use (manipulate) again without additional "fiddling" around with RESTORE lines and other rather tedious programming routines.

21.2 **THERE IS A** better way to achieve the same results without having to resort to such a lengthy program. The simpler method is through the use of **ARRAYS**.

ARRAYS are often called subscripted variables.

We might say (and the following discussion will center on the first five test scores only)

GRADES 1 = 65

GRADES 2 = 43

GRADES 3 = 82

GRADES 4 = 98

GRADES 5 = 86

The 1, 2, 3, 4 and 5 are subscripts.

Valid as they are in mathematics, the computer is not capable of recognizing variables written in that manner --even if the system had the physical features to allow typing subscripts. Instead, the computer requires us to write them as

```
grades(1) = 65
grades(2) = 43
grades(3) = 82
grades(4) = 98
grades(5) = 86
```

We call these an array GRADES. The 1, 2, 3, 4 and 5 are subscripts. Together, they are SUBSCRIPTED VARIABLES -- in COMAL, simply an **ARRAY**.

So we say

```
grades(1) is grades sub 1
grades(2) is grades sub 2
grades(3) is grades sub 3
grades(4) is grades sub 4
grades(5) is grades sub 5
```

Then

```
grades sub 1 = 65
grades sub 2 = 43
grades sub 3 = 82
```

grades sub 4 = 98
grades sub 5 = 86

We can, for conceptual purposes, visualize the array GRADE as looking like this inside the computer:

ARRAY GRADES				
1	2	3	4	5
65	43	82	98	86

--with each entry of the array GRADES being put into separate **CELLS**. Once the data is "placed" into the CELLS it can be retrieved, used and "put back," again and again and again.

The importance and inherent power of ARRAYS cannot be overemphasized. Look at it this way, if we have a journey of some distance to make we can

--walk	use elementary computer skills
--ride a bike	use computer skills, such as looping
--drive a car	use above skills combined with ARRAYS

21.3 **OUR FIRST TASK** is to place these five grades in the cells of an array GRADES. But before we can do that, we must take care of some "housekeeping tasks" first.

Unless we "tell" the computer we are going to use an ARRAY, all our programming will come to naught. And after saying, we must also state how many STORAGE CELLS we wish to set aside for the DATA.

Here is a statement that does both of these tasks.

```
DIM grades(5)
```

DIM is short for DIMensioning; in effect

```
DIM grades
```

can be translated as

"Computer, make provisions for an array called GRADES."

The "(5)"--

"...and in these provisions, allow for 5 STORAGE CELLS."

The name "GRADES" is, of course, quite arbitrary. We could just as easily have named the array TESTSCORES or TESTRESULTS. What is important (!!) is, once having named the array, the only way to get the computer to

go back and retrieve the data in those storage cells, is to use the SAME NAME of the array!!

Placing these five test scores into the CELLS is easily accomplished with the following algorithm.

```
              DIM grades(5)

     -£-      FOR scores:=1 TO 5 DO
     -££-      READ grades(scores)
              ENDFOR scores

              DATA 65,43,82,98,86
```

*****

-£- "scores" is just the name of the variable for the FOR-DO loop. Like "grades," it does have some relevance to what we are doing.

-££- READ is the key word here. It's THE verb that does the "work" of placing into the CELLS each of the test results as the variable SCORES assumes values of 1, 2, 3, 4 and 5.

Note, also, the way the line is written.

```
READ grades(scores)
```

The "grades(scores)" continues the ARRAY format found in the DIM grades(5) line.

READ, of course, finds the DATA line and --when SCORES has the value of "1", places 65 into that STORAGE CELL in the array GRADES.

21.4 ADD THESE LINES to the program.

```
print grades(2)
print grades(3)
print grades(4)
print grades(1+4)
```

When we RUN the program, our printout is

```
43
82
98
86
```

The first three are straight forward.

```
print grades(2)
```

tells the computer to find an array named GRADES, go into cell 2 and whatever is there, print it on the screen. We could put that in a loop and have the "43" printed as many times as we wished --and WITHOUT HAVING TO USE THE RESTORE statement!!! It's the last line that is significant.

```
print grades(1+4)
```

Unlike specifying a simple variable, the subscript may, in itself, be a variable expession or even a numerical expression! Hence

```
print grades(1+4)  becomes
print grades(5)
```

And in CELL 5, the value is 86.

One further example.

```
              print grades(6*4 - 100/5)

  becomes,    print grades(24 - 100/5)

  becomes,    print grades(24 - 20)

  becomes,    print grades(4)

  which is,   98
```

At first, it may seem somewhat esoteric to specify subscripts as a function of a variable. But do not be too quick to dismiss the idea. There are times when it is very handy.

21.5 **WE WILL NOW** write a program to sum the grades. Because of the small number of scores, we will us a FOR-DO loop instead of the WHILE-DO. In actual practice the latter is preferred and we will illustrate its use in the next section.

```
-£-       DIM grades(6)

-££-      sumgrades:=0

          FOR scores:=1 TO 5 DO
           READ grades(scores)
-£££-      sumgrades:+grades(scores)
          ENDFOR scores

-££££-    grades(6):=sumgrades

-£££££-   PRINT grades(6)

          DATA 65,43,82,98,86
```

*****

-£- We have allowed for an extra CELL in the array

-££- Initializing the variable SUMGRADES

-£££- The summation statement --note the use of the array notation: GRADES(SCORES)

-££££- After getting the sum of the grades (see, -£££-), it was placed in STORAGE CELL 6 of the array GRADES

-£££££- The PRINT statement to retrieve the sum of the scores placed there.

The sum is 374.

Once we place the sum into CELL 6 we can use it as many times as we wish, and for any purpose.

A further note: It is good programming practice to allow for a few more STORAGE CELLS than absolutely required. In longer programs, there may be an unforseen need to use this facility which ARRAYS provide. But don't be too foolish about it. A good number for this problem would have been 7 --or perhaps, 8.

Remember this, when we ask the computer to set aside the CELLS, it is using up space to do so. And it is space we cannot use later even though several STORAGE CELLS may remain empty. The computer is "loyal." If we want 400 BOXES, 400 we will get and nothing, BUT NOTHING, will cause the computer to give those up. So, add a few in the DIM statement --but not too many!

21.6 **SUPPOSE (AND FOR** large numbers of test scores it is not unusual to do the following for statistical purposes) we wish to sum the odd-position scores in the DATA statement. (When this is done, the even-positioned scores are also added and the two distinct totals are then subjected to statistical analysis.)

Enter the following program and spend what time is necessary studying it, so that it is fully understood.

```
        // ** main control sequence **

        clearscreen
        setvariables
        storedata
        oddscores
        printgradesum
        END " "

        // ** end main control sequence **

        PROC setvariables
 -*-      oddgradesum:=0
 -*-      scores:=0
        ENDPROC setvariables
```

```
        PROC storedata
-£-       DIM grades(8)
          WHILE NOT EOD DO
-££-        scores:+1
-£££-       READ grades(scores)
          ENDWHILE
        ENDPROC storedata

        PROC oddscores
-££££-    FOR sumodd:=1 TO scores STEP 2 DO
-£££££-     oddgradesum:+grades(sumodd)
          ENDFOR sumodd
        ENDPROC oddscores

        PROC printgradesum
          PRINT "sum of the odd grades =";oddgradesum
        ENDPROC printgradesum

        DATA 65,43,82,98,86
```

(The answer is 233.)

0 + 65 (65)

65 + 82 (147)

147 + 86 (233)

*****

-*- Some programmers prefer to initialize variables within the PROCEDURES using them. This is understandable, particularly if some of the variables are going to be used in a CLOSED procedure. In this particular program there are no such procedures.

-£- We set aside two extra cells.

-££- We must increment SCORES to provide for separate storage cells for the ARRAY "grades."

-£££- The array GRADES where values from the DATA line are placed in cells 1 through 5, successively.

-££££- By specifying STEP 2, then the variable SUMODD takes the values 1, 3 and 5 on the first, second and third trips.

-£££££- The summation loop for the odd grade scores.

Now, the fact we used

```
grades(SUMODD)
```

instead of

```
grades(SCORES)
```

to manipulate the same DATA scores, should cause no difficulty PROVIDING we remember that THE NAME OF THE ARRAY IS THE IMPORTANT THING. The name of the variable to facilitate "getting around" the loop, is of no consequence.

If SCORES equals 3 and SUMODD equals 3, then the following three expressions will produce the same results in this program --and that is, 82.

```
print grades(SCORES)

print grades(SUMODD)

print grades(3)
```

Consider the subscripts as "names" to help us keep the different routines clear in our thinking and in the algorithm.

We said, "The name of the variable to facilitate 'getting around' the loop, is of no consequence." This is not a precise statement. We know we can use variables for subscripts. A more correct observation would be:

> As long as we placed (READ) the values GRADES(SCORES) into the CELLS, calling them out by GRADES(SUMODD) or GRADES(MARK) or GRADES(NUMBERS) is not going to change what was originally placed into the STORAGE CELLS.

Here is a simple program to illustrate this point. In this instance, the grade results are INPUT by the user, and then stored in an array called GRADES. The assumption, of course, is that the user knows how many grades are to be entered.

```
        sumgrades:=0

        count:=0

        input "how many grades? ":number

-£-     dim grades(number+4)

        // ** main control sequence **

        clearscreen
        enter
        recall
        total
```

```
results

end "sample grade program"

// ** end main control sequence **

         PROC enter
-££-       FOR entry:=1 TO number DO
             PRINT "enter grades";entry
-£££-        INPUT grades(entry)
           ENDFOR entry
         ENDPROC enter

         PROC recall
           PRINT "the grades you entered were";
           FOR show:=1 TO number DO
             PRINT grades(show);
           ENDFOR show
         ENDPROC recall

         PROC total
           FOR add:=1 TO number DO
             sumgrades:+grades(add)
             count:+1
           ENDFOR add
         ENDPROC total

         PROC results
           averagegrades:=sumgrades/count
           PRINT
           PRINT "number of grades entered";count
           PRINT
           PRINT "sum of the grades";sumgrades
           PRINT
           PRINT "the class average";averagegrades
         ENDPROC results
```

*****

-£- DIM can't be done until after the input of variable NUMBER

Subscripts can be a function. We allowed 4 extra cells.

-££- FOR-DO with a variable as a terminator! We've done this before!

-£££- INPUT grades(entry) has the same effect as READ grades(entry)

The balance of the algorithm (program) is direct. Enter it into the computer, RUN it a few times and then take time to study it in depth.

21.7 A CLASSIC PROBLEM is to throw a pair of dice a given number of times and count how often each value appears.

We could write this as a series of eleven (11) IF-THEN statements (there is no 1 value for a pair of dice); or, we could use CASE-OF; or, for that matter, any two or three other approaches. The simplest, however, is to employ an ARRAY structure.

Enter the following. (PROCEDURE structure is set aside for this illustration.)

```
            page

   -£-      dim dice(14)

            randomize

            zone 4

            for throw:=1 to 100 do
            die1:=rnd(1,6)
            die2:=rnd(1,6)
            cube:=die1+die2
            print cube;

   -££-     dice(cube):+1

            endfor throw

            zone 7

            print
            print
   -£££-    for results:=2 to 12 do

            print results, "was rolled",dice(results),"times"

            endfor results
```

*****

-£- A couple of extra STORAGE CELLS are set aside.

-££- An array summation statement!

When --say, CUBE is a 6, then the array statement

```
dice(cube):+1   --becomes
dice(6):+1
```

and the value stored in BOX 6 of the array DICE is incremented by 1. If there is no value there, then it

becomes 1 (0+1). If, on the other hand, dice(6) had had the value of 7, then this amount in the CELL would would been increased to 8 (7+1).

This holds for all the CELLS in the array called DICE

-£££- The FOR-DO loop to retrieve the results stored in the array DICE. The lowest number we can roll is a 2.

One thing we can study with a program like this is to compare the results we get with those expected from mathematical probability. For example, how many times out of 100 should we expect a "7"?.

Mathematically, it should be 16+. Any number thrown on the first die can be combined with one number on the second, to get a 7. That being so, the probability on the first die is

1 out of 1 --or 1/1

The second die is

1 out of 6 --or 1/6

$$\frac{1}{1} \times \frac{1}{6} = \frac{1}{6} \times 100 = 16.6666667$$

Run the program and note how close it is to the mathematical probability.

21.8 GIVEN THE FOLLOWING two sets of integers.

8 13 51 19 22

44 79 36 1 85

Store these integers in two separate arrays and then print them out in the sequence

85 22 1 19 36 51 79 13 44 8

```
page

dim first(7)

dim second(7)

for toprow:=1 to 5 do
read first(toprow)
endfor toprow

for bottomrow:=1 to 5
read second(bottomrow)
endfor bottomrow
```

```
        for results:=5 to 1 step -1 do
-£-     print second(results);
-£-     print first(results);
        endfor results

        data 8,13,51,19,22
        data 44,79,36,1,85
```

*****

-£- As the value of RESULTS is introduced into the array notation, the computer goes to that CELL, picks out the value placed there and then prints it to the screen.

On the first loop both arrays read

    second(5) and first(5)

The two values --printed in that order-- are 85 and 22.

The trailing punctuation (;) assures the printouts will be on one line.

The second trip through the loop gives

    second(4) and first(4)

and so forth.

RECAP
*****

ALL arrays must be DIMensioned.

READ or INPUT are the verbs used to place the data into the array cells.

SUMNUMS:+ARRAYNAME(SUBSCRIPT) is the form of the summation statement for an array.

ARRAYNAME(SUBSCRIPT):+1 is the form of the count statement for an array.

problems

****

1. Take the following Computer test grades and calculate their average. Tell home many grades were above and below the average. (78, 84, 96, 92, 91, 88, 99, 89, 95, 98)

2. In the previous problem, print the point difference of the scores from the average.

3. Put the following integers into two different arrays.

   -- 15, 12, 14, 28, 17, 21, 31, 19

   -- 21, 18, 32, 11, 25, 24, 20, 15

   Have the printout in 3 rows --the third row to be the product of the first two.

row 1	row 2	product
15	21	315
12	18	216

   et cetera

4. Store these values.

   18, 21, 13, 25, 27, 11, 19, 28, 22, 10

   Sum the odd-positioned values; sum the even-positioned values. Have the following printout.

odd	even	
18	21	
13	25	
27	11	
19	28	
22	10	
99	95	difference = 4

5. Roll a pair of dice 100 times. Print a table showing the number of times each even value was thrown.

6. Do the same as problem 5, but print the results for the odd values.

7. Throw the dice 100 times. Instead of counting the number of times each value appeared, store the FACE VALUE of the dice. Thus, if 6 three's were thrown, the printout would be 18.

   total face value of the 2s = ----

total face value of the 3s = ----

. . . .

total face value of the 12s = ----

8. Sum the face values for all the throws in problem 7 and take an average/throw. How much does the average differ from 7?

chapter twenty-two

$tring arrays
****

objectives

At the completion of the chapter the student will be able to:

1. DIMension a $tring array properly, by specifying the correct number of CELLS to be set aside and making allowance for the number of characters in the $tring;

2. State what will happen to the output $tring if the DIMension statement is incorrect;

3. Write a program involving two or more $tring arrays;

4. DIMension two or more $tring arrays on the same program line;

5. Center the printout to the screen by using the PRINT AT command;

6. Combine a $tring array and a numeric array in one program; and,

7. READ, in one program line, any combination of $tring and/or numeric array.

```
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
c                                                               c
o  TOGGLING REVERSE PRINTING                                    o
m  -------------------------                                    m
a                                                               a
l  Pressing the <CTRL> key and then 9 or 0 toggles  reverse on  l
c    and off for the characters printed to the screen.          c
o                                                               o
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
```

22.1 **$TRING ARRAYS ARE** a logical extension of numeric arrays. Actually, the essential difference is the use of the $ tagged to the end of the variable name and THE SPECIFICATION OF HOW MANY CHARACTERS TO ALLOW FOR EACH $TRING. We will illustrate with a simple program.

PROBLEM: Store the following five names in an array and then print them out in reverse order: zedekiah, selah, rahab, hezekiah, balaam. Print the results in the middle of the screen.

```
          // illustrating a simple $tring array

          page

-£-       row:= 8

-££-      dim name$(7) of 12

          for storenames:=1 to 5 do
          read name$(storenames)
          endfor storenames

-£££-     for recall:=5 to 1 step -1 do
          row:+1
-££££-    print at row,15:name$(recall)
          endfor recall

          data "zedekiah","selah","rahab","hezekiah","balaam"

-£££££-   cursor 21,1

          end "end $tring illustration"
```

*****

-£- Because we want to print the results in the middle of the screen, we will use the command PRINT AT. Doing that we will need to initialize a variable (ROW in this instance) to something other than 0 --8, here.

-££- Two extra cells are provided.

The 12 limits the length of each name to 12 characters; if more letters are entered the name(s) will be truncated.

-£££- Reverse order, because we want the names to be printed so.

-££££- Here is the PRINT AT. The first time around BALAAM will be printed on the screen at row 9, column 15. The next time through, HEZEKIAH, will be printed at row 10, column 15 .... and so on.

(Just a review of a command introduced earlier.)

-£££££- This line is optional; it may be left out of program.

22.2 WE CAN COMBINE $tring arrays as we did numeric ones. Consider this problem.

Store the following nouns, verbs and objects in three arrays. Print ten sentences by using random selections from each array. (If a sentence appears more than once, could it be true?)

NOUNS. loretta, beth, doug, shawn, dallas

VERBS. salutes, kisses, pets, kicks, phones

OBJECTS. frogs, dolls, toys, worms, computers

```
        // illustrating three arrays in one program

-£-     DIM noun$(5) OF 10,verb$(5) OF 10,object$(5) OF 10

        // ** main control sequence **

        clearscreen
        propernouns
        sillyverbs
        foolishobjects
        sillysentences
        END " "

        // ** end main control sequence **

        PROC propernouns
          FOR name:=1 TO 5 DO
-££-        READ noun$(name)
          ENDFOR name
        ENDPROC propernouns

        PROC sillyverbs
          FOR action:=1 TO 5 DO
-££-        READ verb$(action)
          ENDFOR action
        ENDPROC sillyverbs

        PROC foolishobjects
          FOR goal:=1 TO 5 DO
-££-        READ object$(goal)
          ENDFOR goal
        ENDPROC foolishobjects

        PROC sillysentences
          FOR sentences:=1 TO 10 DO
            RANDOMIZE
-£££-       word:=RND(1,5)
```

```
              PRINT noun$(word);
     -£££-    word2:=RND(1,5)
              PRINT verb$(word2);
     -£££-    word3:=RND(1,5)
              PRINT object$(word3)
              PRINT
            ENDFOR sentences
          ENDPROC sillysentences

          DATA "loretta","beth","doug","shawn","dallas"
          DATA "salutes","kisses","pets","kicks","phones"
          DATA "frogs","dolls","toys","worms","computers"
```

*****

-£- We can DIMension more than one array in a line --each, however, must be separated by a comma.

-££- Storing the nouns, verbs and objects in three different arrays. Note how the DATA lines are written to make certain there is no conflict in storage.

-£££- We use three different random notations. If we had used but one, then whatever was generated would be in the same relative position for each word. This way the nouns, the verbs and the objects will be more randomly chosen. It will minimize sentence redundancy.

22.3 **AND, OF COURSE,** we can combine numeric and $tring arrays. Because this chapter is an extension of the previous, let's introduce the idea through another problem.

PROBLEM. Personnel of the Garbanzo Company, Limited turn in the following sales for one week.

```
john    296
pete    312
mary    408
alma    471
bilj    388
```

Store the information in two arrays. Retrieve it and create the following table.

sales person	amount sold	cum total
john	296	296
pete	312	608
mary	408	1016
alma	471	1487
bilj	388	1875

Center it to the screen.

```
// combining $tring and numeric arrays

cumtotal:=0
row:=8
dim name$(6) of 5,button(6)
zone 9

// ** main control sequence **

clearscreen
salesperson
sales
table
end ""

// ** end main control sequence **

-£-     print at 6,9:"sales","amount"," cum"
-££-    print at 7,9:"person"," sold","total"
        print

        proc salespersons
         for person:=1 to 5 do
          read name$(person)
         endfor person
        endproc salespersons

        proc sales
         for garbanzo:=1 to 5 do
          read button(garbanzo)
         endfor garbanzo
        endproc sales

        proc table
         for results:=1 to 5 do
          cumtotal:=+button(results)
          row:+1
-£££-     print at row,9:name$(results),button(results),cumtotal
         endfor results
        endproc table

        data "john","pete","mary","alma","bilj"
        data 296,312,408,471,388
```

*****

-£- Row 6 will be about the upper limit to center the table.

-££- Specifying row 7 is important, otherwise this line will print over the previous one.

-£££- Because of the statement in the previous line--

row:+1

--printing will start at row 10, thus leaving a blank between the heading and the column printouts.

As RESULTS changes in value from 1 through 5, the various names and button sales totals will be called out of their respective CELLS and printed.

The CUMTOTAL summation statement was written two lines ago, and is called out here to PRINT its running tally.

As straight forward as this algorithm is, it really is not a "good" one. If, in the real world, the names of the sales people had to be written in one DATA line and their sales on another (in respective order) all sorts of errors and inconveniences would occur. Imagine having to make double entries for 50 or 100 sales people!

Let's make some simple changes that will not be difficult to understand.

1. Delete both of the present DATA lines.

2. Enter this DATA line:

```
"john",296,"pete",312,"mary",408,"alma",471,"bilj",388
```

Now, this makes more sense. It is far more logical to enter the name of the sales person and put immediately after, the number of garbanzo buttons sold.

With this type of entry the operator's errors will be reduced to a minimum.

Of course, this means we will have to change the line(s) calling for the READ statement.

3. Delete, in the MAIN CONTROL SEQUENCE, the line calling for the procedure SALES.

The MCS will now read:

```
clearscreen
salesperson
table
end " "
```

4. Delete the five program lines in the SALES procedure sequence. (Use the command-- DEL SALES.)

5. In PROC SALESPERSON, rewrite the line

```
read name$(person)
```

to

```
read name$(person),button(person)
```

RUN the program and study the algorithm to determine what has happened.

```
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
.                                                                 .
.  RECAP                                                          .
.  *****                                                          .
.                                                                 .
.  $TRING arrays  must  be  DIMensioned  for the                  .
.      number of cells  to  be set aside and for                  .
.      the length of each $tring to be entered.                   .
.                                                                 .
.  NUMERIC  and/or  $tring arrays may be  DIMen-                  .
.      sioned on the same program line.                           .
.                                                                 .
.  PRINT AT  allows printing at any place on the                  .
.      screen by specifying row and column.                       .
.                                                                 .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
```

problems
****

1. Store the following towns in an array:

   dunmore, beard, slatyfork, green bank, frost, hillsboro, woodrow, frank, durbin, cass, arbovale, stillington, minnehaha, watoga, buckeye, marlinton, campbelltown, showshoe, seebert, droop, denmar, lobelia, huntersville

   Print the list in reverse order AND only the first four letters of each town.

2. Six sales persons for the Garbanzo Buttons, Limited factory are:

   ```
   jack   361 buttons sold
   john   234 buttons sold
   bilj   512 buttons sold
   mary   635 buttons sold
   alma   489 buttons sold
   pete   609 buttons sold
   ```

   Their target sales for the week was 3000 buttons.

   Complete the following table. Put a sentence below the table showing by how much they exceeded or came short of the target. Have the computer do all the work! Center to the screen.

```
sales target:   3000 buttons

person       sold       left

jack         361        2639
john         234        2405

    and so forth
```

3. Create 15 single-spaced sentences at random from these words:

   --pete, mary, jack, john, beth
   --runs, walks, speaks, yells, whispers
   --without, on, to, into, with
   --shoes, toe nails, trees, elbows, teeth

4. Modify problem 2 so that there is no DATA line, but everything is stored in the arrays by INPUT statements. (The last sentence is still to be printed.)

5. Put the alphabet into an array. Print the alphabet in reverse, starting with the letter z and skipping every other letter. (z x v t .....). Print it in 2 columns and centered to the screen.

ARRAY NOUN$(NUM)

1	2	3	4	5
pete	mary	jack	john	beth

ARRAY VERB$(BOX)

1	2	3	4	5
runs	walks	speaks	yells	whispers

ARRAY PREP$(CELL)

1	2	3	4	5
without	on	to	into	with

ARRAY OBJECT$(COUNT)

1	2	3	4	5
shoes	toe nails	trees	elbows	teeth

chapter twenty-three

two-dimensional arrays
****

objectives

At the completion of the chapter the student will be able to:

1. State how two-dimensional arrays place data in CELLs;
2. Properly DIMension multi-dimensional arrays;
3. State how the positioning of a CELL is determined;
4. Write an expression that will perform an arithmetic computation on any series of pre-determined CELLs;
5. Draw a chart to show how CELLs are identified; and,
6. Write a program that will total rows and columns of data through the use of one- and two-dimensional arrays.

----

comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal

LONG VARIABLE NAMES

COMAL allows up to 78 characters for naming variables. This is a tremendous help, but it can be confusing because the names run together. To assist the programmer, COMAL allows the use of an apostrophe (') in the variable name to separate the words. Thus, the variable

totalofcolumns

can be written as

total'of'columns

comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal

23.1 **IN THE PREVIOUS** two chapters we assigned single variables --numeric and/or $trings-- into individual CELLS. And we did this in a linear way. For example, if we had a DATA line that read

```
DATA 65,36,19,18,27,44
```

we put 65 into CELL 1; 36 into CELL 2; 19 into CELL 3; and so forth.

Thus, if the array had been assigned the name GRADES then

```
print grades(5)
```

would put 27 on the screen.

This is fine until there is a need to store and process data that cannot be handled conveniently in a linear manner. Suppose the response to a questionnaire concerning the Junior/Senior prom was as follows.

QUESTION: Do you believe the Junior/Senior prom should include the Sophomore class?

	YES	NO	UND
JUNIORS	9	93	24
SENIORS	35	81	27

How do we store this information --how do we place it into CELLS, so it can be processed?

As far as the numbers (the vote results) are concerned, we have two **ROWS**.

ROW 1: 9 93 24

ROW 2: 35 81 27

Or, considered from another perspective, we have three **COLUMNS**.

column 1	column 2	column 3
9	93	24
35	81	27

In multi-dimensional arrays, the computer stores DATA by **ROWS** and **COLUMNS**.

For the vote results on the Junior/Senior prom

```
 9 is in ROW 1  COLUMN 1
93 is in ROW 1  COLUMN 2
24 is in ROW 1  COLUMN 3
```

```
35 is in ROW 2  COLUMN 1
81 is in ROW 2  COLUMN 2
27 is in ROW 2  COLUMN 3
```

After the vote results have been stored properly (we'll discuss how to do that shortly) we can get a printout of --say, these four numbers-- 9, 93, 81 and 27 as follows.

```
print votes(1,1) would give  9
print votes(1,2) would give 93
print votes(2,2) would give 81
print votes(2,3) would give 27
```

The first number inside of the parentheses refers to the ROW where the vote is stored; the second number to the COLUMN. Together, the ROW and COLUMN pinpoints the CELL where the vote can be located.

Using the same array, consider this important property.

```
print votes(1,1) + votes(2,3).

             9   +       27 = 36
```

In processing numeric data, we make use of this constantly.

Do the following problems for this matrix.

```
16 14 21 23 31
61 12 32 13 41
11 24 21 33 41
```

(This matrix has 3 rows and 5 columns.)

```
print tally(2,2) + tally(3,4)
print tally(1,5) - tally(3,3)
print tally(3,1)*tally(1,5)
print tally(3,4)/tally(3,1)
print tally(2,1)-tally(2,5)+tally(5,5,)
```

(The answers, in mixed order, are 10, 3, 1, 343, 45.)

23.2 AGAIN, AS WITH linear arrays, before we can place the data into CELLS, the array has to be DIMensioned. And, quite logically, the DIM statement will have to specify the number of ROWS and the number of COLUMNS --and in that order.

For the results of the Prom Survey, the DIM statement would be

```
dim votes(2,3)
```

The "2" because thre are two ROWs --juniors (1) and seniors (2). The "3" because there are three COLUMNS --Yes (1), no (2) and und (3).

Having done this, we can take the results and place them into individual

CELLs. We can visualize the "storage" area inside the computer as

	col 1	col 2	col 3
row 1	cell 1,1	cell 1,2	cell 1,3
row 2	cell 2,1	cell 2,2	cell 2,3

Here is the algorithm to place the data in the proper storage CELLS.

```
1000 dim votes(2,3)
1010
1020 for row:=1 to 2 do
1030
1040 for column:=1 to 3 do
1050 read votes(row,column)
1060 endfor column
1070
1080 endfor row
1090
1100 data 9,93,24,35,81,27
```

When we enter it and then LIST, it will look like

```
1000 DIM votes(2,3)
1010
1020 FOR row:=1 TO 2 DO
1030
1040   FOR column:=1 TO 3 DO
1050     READ votes(row,column)
1060   ENDFOR column
1070
1080 ENDFOR row
1090
1100 DATA 9,93,24,35,81,27
```

(Remember, this is a partial algorithm of a larger program. The usual logo, main control sequence and other "housekeeping" parts are not included here --though they would be in our final program.)

Now, let's follow in some detail the process.

1000 Sets aside six storage CELLS (2*3 = 6)

1020 On ROW first trip, ROW = 1.

(falls through to)

1040 On COLUMN first trip, COLUMN = 1

(falls through to)

1050 The READ searches out the DATA line and places the "9" in CELL 1,1.

(goes back to 1040 for trip 2)

1050 The READ searches out the DATA line and places the "93" in CELL 1,2.

(goes back to 1040 for trip 3)

1050 The READ searches out the DATA line and places the "24" in CELL 1,3.

(GOES BACK TO 1020 FOR ROW TRIP 2 ! !)

(falls through to)

1040 On COLUMN first trip COLUMN = 1

(falls through to)

1050 The READ searches out the DATA line and places the "35" in CELL 2,1.

(goes back to 1040 for trip 2)

1050 The READ searches out the DATA line and places the "81" in CELL 2,2.

(goes back to 1040 for trip 3)

1050 The READ searches out the DATA line and places the "27" in CELL 2,3.

After completing these three rounds in the FOR-DO loop, it will have finished its task for ROW:=1 to 2 and COLUMN:=1 to 3.

Add the following lines.

```
1200 print votes(2,3)
1210 print votes(1,3)
1220 print votes(2,1)
1230 print votes(2,4)
```

Your printouts will be

```
27
24
35
index out of range
```

There is no CELL 2,4 --hence, the error message.

Now, delete 1200 - 1230.

23.3 THE QUESTIONNAIRE RESULTS

which we have stored, were

	yes	no	und
juniors	9	93	24
seniors	35	81	27

We would like to total the results.

First, we want the totals for "yes", "no" and the "und" votes. The following lines will give us this. (Do the necessary "housekeeping" tasks to provide room for these lines. Line numbers will not be given here so we can arrange the algorithm to our liking.)

```
sumyes1:=0
sumno2:=0
sumund3:=0

// *****

for row:=1 to 2 do
 sumyes1:+votes(row,1)
endfor row
print sumyes1

for row:=1 to 2 do
 sumno2:+votes(row,2)
endfor row
print sumno2

for row:=1 to 2 do
 sumund3:+votes(row,3)
endfor row
print sumund3
```

By fixing the COLUMN to 1 or 2 or 3 in each of the FOR-DO loops, we are assured that as ROW changes from 1 to 2, the COLUMN will remain the same. So, in the first loop, only the two totals --9 and 35-- will be added, because they are in COLUMN 1. The same is true for the "no" COLUMN (which is 2) and for the "und" COLUMN (which is 3).

At the end of each of the loops, we asked for the respective sums to be printed. This was for illustration purposes only. More work would have to be done to place them under the correct columns --either through the use of trailing punctuation or ZONE. It's not important because, we will agree, this is an awkward and cumbersome way to get the totals. (Imagine if we had 14 columns to add!) But it will help us to understand what can

be done to get the results we need.

We must get the computer to do more of the work --and we can!

Before making changes in the algorithm, SAVE it as is. Should we get into difficulty, we can reLOAD and start again.

Delete the following lines and loops.

```
sumyes1:=0
sumno2:=0
sumund3:=0

*****

FOR-DO row loop where COLUMN was 1
FOR-DO row loop where COLUMN was 2
FOR-DO row loop where COLUMN was 3
```

The sum of the columns is linear, i. e., there is only one line of sums. This being so, we can store the sums in another array that is linear in nature. Go back to the DIM line and change it by adding --total'of'col(3), so that it now reads

```
dim votes(2,3),tot'of'col(3)
```

Add the following to the end of your present algorithm.

```
for row:=1 to 2 do

for column:=1 to 3 do
tot'of'col(column):+votes(row,column)
endfor column

endfor row
```

This nested loop routine goes back into the CELLs where the votes have been stored, and gets them "back out again." As they are brought out --each column in turn ("yes", "no" and "und")-- are added, and the final totals are stored in the CELLs of the one dimensional (linear) array, TOT'OF'COL.

When we have need of them we can get the totals at any time. We may verify they are there by adding, temporarily, these lines and RUNning the program.

```
for column:=1 to 3 do
print totcol(column);
endfor column
```

23.4 ANOTHER TOTAL WE would like is the number of Juniors and Seniors who voted. The is a ROW sum.

Having followed the logic of this discussion, it is not difficult to see that we can get this quite easily by adding one more line to the TOT'OF'COL subroutine.

```
for row:=1 to 2 do

for column:=1 to 3 do
tot'of'col(column):+votes(row,column)
TOT'ROW(ROW):+VOTES(ROW,COLUMN)
endfor column

endfor row
```

We also need to change the DIM line to read

```
dim votes(2,3),tot'of'col(3),tot'row(2)
```

The total for the ROWs are now stored in the linear array TOT'ROW.

To verify this, add the following to the program

```
for row:=1 to 2 do
print tot'row(row)
endfor row
```

and RUN the program.

The figures are not printed in the proper place --as yet.

Delete the last two temporary sub-routines.

When the algorithm is completed, the table will look like this.

	yes	no	und	tot
juniors	9	93	24	126
seniors	35	81	27	143
totals	44	174	51	269

One version of a final algorithm (minus the logo) is

```
// junior-senior prom vote

DIM votes(2,3),tot'of'col(3),tot'row

total'votes'cast:=0

count:=0

// we will use count:=0 to
// print jrs. & srs. later

// ** main control sequence **
```

```
PAGE
storevotes
sumrowcolumns
totalvotes
printtable

CURSOR 22,1

END "end prom survey"

// ** end main control sequence **

PROC storevotes

 FOR row:=1 to 2 DO
  FOR column:=1 to 3 DO
   READ votes(row,column)
  ENDFOR column
 ENDFOR row

ENDPROC storevotes

PROC sumrowcolumns

 FOR row:=1 to 2 DO
  FOR column:=1 to 3 DO
   tot'of'col(column):+votes(row,column)
   tot'row(row):+votes(row,column)
  ENDFOR column
 ENDFOR row

ENDPROC sumrowcolumns

PROC totalvotes

 FOR row:=1 to 2 DO
  FOR column:=1 to 3 DO
   total'votes'cast:+votes(row,column)
  ENDFOR column
 ENDFOR row

ENDPROC totalvotes

PROC printtable

 ZONE 8

 PRINT " ","yes","no","und","total"

 // the space   " ",  was needed
 // otherwise, "yes" ends up
 // over the word "juniors"

 PRINT
```

```
FOR row:=1 to 2 DO

 count:+1

 IF count=1 THEN PRINT "juniors",
 IF count=2 THEN PRINT "seniors",

 FOR column:=1 to 3 DO
  PRINT votes(row,column),
 ENDFOR column

 PRINT tot'row(row)

 // no trailing punctuation in
 // the above print command so
 // the ZONE command is "kicked"

 PRINT

ENDFOR row

PRINT " ",

// the   " ",   so the first
// column doesn't end up under
// the word "seniors"

PRINT

PRINT "totals",

FOR column:=1 to 3 DO
 PRINT tot'of'col(column),
ENDFOR column

PRINT total'votes'cast

ENDPROC printtable

DATA 9,93,24,35,81,27
```

Though the program looks long and involved, in reality it is direct. Through the use of trailing punctuation the various $trings can be put in their proper place -- and so can the row and column totals. Some will observe that more could have been accomplished in PROC STOREVOTES.

Quite frankly, that is true. But the programmer must never lose sight of the user --or, for that matter, debugging. Considering the power of the program, it is a small price to pay by adding two or even three more PROCEDURES.

The skills introduced through the detailed discussion of this problem will be further strengthened in the exercises to follow.

23.5

BEFORE LEAVING THIS chapter, we should modify the previous algorithm to show how INPUT can work in place of READ - DATA.

Actually, in "the real world" we would prefer to enter the results of a questionnaire by INPUT anyway. Typing a program and then having to enter a DATA line is not convenient to the user.

First, type

```
list totalvotes
```

The computer will go into the program and list to the screen all the lines in the

```
procedure totalvotes
```

This procedure, you will remember and will note on the screen, contained the line

```
read votes(rows,columns)
```

We want to replace that line with an INPUT statement. However, it would be well to add some other lines so that the user will know exactly what numbers he should type in. Therefore, we will put in some prompts.

The next thing to do is to DEL the line which now reads

```
read votes(rows,columns)
```

After that, restructure the procedure totalvotes to be as follows.

```
PROC storevotes

  FOR rows:=1 TO 2 DO
    FOR columns:1 TO 3 DO

      IF rows=1 THEN PRINT "enter senior";
      IF rows=2 THEN PRINT "enter junior";
      IF columns=1 THEN PRINT "yes votes",
      IF columns=2 THEN PRINT "no votes",
      IF columns=3 THEN PRINT"und votes",

      INPUT votes(rows,columns)

    ENDFOR columns
  ENDFOR ROWS

ENDPROC storevotes
```

The value of the prompts will become evident when the program is run. Type in the numbers as they are requested.

RUN the program.

RECAP
*****

MULTI-DEMENSIONAL arrays are DIMensioned by ROWs and COLUMNs.

READ and INPUT are the two verbs used to store data in a the cells of a multi-dimensional array.

problems
****

1. The results of a survey on the question:

   Do you feel we should have a yearbook
   that primarily features Seniors?

   was--

	yes	no	und
fr	89	26	31
sp	52	73	12
jr	17	90	23
sr	96	11	12

   Write a program to sum the rows and columns and tallies the total number of votes cast.

2. The table below represents the totals of sales made by four sales people during one week. Write a program to reproduce the table, complete with all totals.

	mon	tue	wed	thu	fri	sat
alma	48	40	73	120	100	90
bilj	75	130	90	40	110	85
mary	50	72	140	125	106	92
joan	108	75	92	152	91	87

3. Given 12 "?" marks in a DATA line. Write a program to have the following three printouts.

```
????            ??????          ??
????            ??????          ??
????                            ??
                                ??
                                ??
                                ??
```

4. Given the following two arrays:

```
14 12 26 35        41 21 62 53

45 51 81 17        54 15 18 71
```

   Write a program that will store these two 2 by 4 arrays and then will add them together --corresponding element plus corresponding element-- and will print out all three arrays.

5. Go back to problem 1 and make the necessary changes in the algorithm so that data can be entered as INPUT. Put in the program prompts.

chapter twenty-four

sorting
****

objectives

At the completion of the chapter the student will be able to:

1. Write a program using the method of "seeding" a minimum number and a maximum number, to print the largest and smallest numbers in a list;

2. Write a program to print the largest and smallest numbers in a list, without resorting to "seeding"; and,

3. Use a simple sort to order a list of items.

```
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
c                                                               c
o  OOPS! I MADE A LINE ENTRY ERROR!!                            o
m  ---------------------------------                            m
a                                                               a
l  If a program line is correct but typed with odd gaps in it,  l
c    put the CURSOR at any position  on the program  line  and  c
o    tap the letter "A" while depressing the CTRL  key.   This  o
m    is particularly helpful for program lines  that are  more  m
a    than one line in length.                                   a
l                                                               l
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
```

24.1 QUITE OFTEN IT is necessary (or of interest) to find the smallest and largest number in a given list. It may be that a company would like to know the name of the salesman who had the most sales for a given period of time; a teacher may wish to know what the top and bottom scores were on an examination; or, a person may wish to know the highest and lowest temperatures for a month at a particular vacatation spot.

24.2 WE SHALL INTRODUCE two methods for arriving at an answer. The first is the simpler of the two, but it does call for an input on our part. The input will be a result of a visual scanning of the data to see what is the general range of the numbers.

The second method will need no such visual scanning --it is a tick more difficult to understand, but should be well within our grasp by this time.

24.3 METHOD ONE. GIVEN the following series of integers

6, 8, 19, 2, 4, 12

It is obvious that 2 is the smallest and 19 is the largest. But how can we write a program to that end?

Because we are dealing with a DATA line and must read each item in it, we will use the WHILE-DO loop. The next thing we will incorporate is a predefined minimum and a predefined maximum. Both of these will be done OUTSIDE of the loop. After having done that, we will be in a position to make comparisons within the loop.

```
          // finding smallest and largest numbers

          page

-£-       minnum:=999

-££-      maxnum:=-999

          while not eod do

          read num
-£££-     if num<minnum then minnum:=num
-££££-    if num>maxnum then maxnum:=num

          endwhile

          print "smallest number is";minnum
          print
          print "largest number is";maxnum

          data 6,8,19,2,4,12

          end "search ended"
```

*****

-£- By setting MINNUM (i.e, minimum number) to 999, which is very large, then the first number read WILL BE smaller! And, of course, we are looking for the smallest number. A visual scan of the data indicated we only needed to set

MINNUM to 99, seeing there were no three-digit numbers.

If there had been a four-digit number we would have set MINNUM:=99999.

-££- The same argument holds for MAXNUM --only in reverse.

-£££- Let's follow this through by inserting, for each item in the DATA line, the actual numbers:

```
if num<minum then minnum:=num
```

```
for the
-------
 6      if  6 < 999  then minnum (replaced by) 6

 8      if  8 <  6  (it isn't, so go no further)

19      if 19 <  6  (it isn't, so go no further)

 2      if  2 <  6   then minnum (replaced by) 2

 4      if  4 <  2  (it isn't, so go no further)

12      if 12 <  2  (it isn't, so go no further)
```

MINNUM, then, is 2.

The changes MINNUM went through were

999, 6, 6, 6, 2, 2, 2

-££££- The same general discussion holds for MAXNUM.

The changes MAXNUM will go through are

-999, 6, 8, 19, 19, 19, 19

Observe that in the first READ of the DATA (6), this number is both MINNUM and MAXNUM --which makes sense, for 6 is both larger than -999 and smaller than 999.

24.4 METHOD TWO. GIVEN the same series of integers

6, 8, 19, 2, 4, 12

How can we write a program without "seeding" MINNUM to 999 and MAXNUM to -999?

Consider

```
// finding smallest and largest numbers
```

```
        page

-£-     read num

        minnum:=num
        maxnum:=num

        while not eod do

        read num
-££-    if num<minnum then minnum:=num
-££-    if num>maxnum then maxnum:=num

        endwhile

        print "smallest number is";minnum
        print
        print "largest number is";maxnum

        data 6,8,19,2,4,12

        end "search ended"
```

*****

-£- This is only ONE READ statement. It will READ only the first number in the DATA line --which is 6. This means that MINNUM = 6 and MAXNUM = 6 in the following two lines.

Having read the first number in the DATA line, there will be only five left: 8, 19, 2, 4 and 12.

-££- The basic principle is the same for both of these lines, so this time let's do the second one step by step

```
if num>maxnum then maxnum:=num
```

for the	
8	if 8 > 6 then maxnum (replaced by) 8
19	if 19 > 8 then maxnum (replaced by) 19
2	if 2 > 19 (it isn't, so go no further)
4	if 4 > 19 (it isn't, so go no further)
12	if 12 > 19 (it isn't, so fo no further)

MAXNUM then is 19.

The changes MAXNUM went through were

6, 8, 19, 19, 19, 19

The same general discussion holds for MINNUM.

The changes MINNUM will go through are

6, 6, 6, 2, 2, 2

As we can see, this second method is not that much more difficult than the first. It does have the additional advantage of not having to scan the data line in order to "seed" artificial minimum and maximum numbers. In fact, if there were several hundred pieces of data, the scan could be fruitless --not to mention time consuming. Let the computer do the work. One doesn't buy a car and then push it around by hand --or walk to town instead of driving.

24.5 FINDING THE FIRST and last name

(alphabetically) in a list, is a natural extension of determining the largest and smallest numbers in an array of numbers.

From Chapter Twelve we know the computer "sees" $trings as numerics. "A" is 65, "C" is 67 and "Z" is 90 --to name three. It is this property that will allow us to write a program to find the first and last name in an array of names.

For example, in the list

bill, heather, stefanie, john, alfred, george

"alfred" is the first name and "stefanie" is the last. In terms of CHR$ numbers, the computer "sees" the first letter in the above as

```
 bill, heather, stefanie, john, alfred, george
 ↑     ↑        ↑         ↑     ↑       ↑
66    72       83        74    65      71
```

Therefore, when we ask for the first name, the computer has little difficulty in locating "alfred" because the value assigned to "a", 65, is the smallest number in the list; the argument for "stefanie" being the last name since 83 is the largest number, is the same.

If there were two or more names in a list starting with the same letter, the computer would then look at the next letter in determining which has the lesser value (or the greater, if that is what is wanted). Consider

joel, joan, jody, josh

The computer would "see" these names as

```
 j  o  e  l,   j  o  a  n,   j  o  d  y,   j  o  s  h

74 79 69 76,  74 79 65 78,  74 79 68 89,  74 79 83 72
```

Because each has the first two letters as 74 and 79, the computer will have to go to the third letter before it can determine which is first name and which is last. In this list, respectively, it is "joan"

(because the third letter, "a", has the lowest value of any third letter) and "josh" (because the third letter, "s", has the highest value of any third letter).

The algorithm for such a program is

```
// finding first and last name in a list

page

read name$

firstname$:=name$
lastname$:=name$

while not eod do

read name$

if name$<firstname$ then firstname$:=name$
if name$>lastname$ then lastname$:=names$

endwhile

print "first name on list is";firstname$
print
print "last name on list is";lastname$

data "jack","george","zelda","henry","albert","bill","mary"

end " "
```

A modification of this algorithm will allow us to use it as an INPUT program. This is more nearly what we would like in the "real world" anyway.

```
        // input names
        // finding the first and last names on a list

        page

        input "how many names? ":howmany

        dim name$ of howmany

        input "enter name 1? ":name$

        firstname$:=name$
        lastname$:=name$

-£-     for names:=2 to howmany

-££-    print "enter name ";names,

        input name$
```

```
if name$<firstname$ then firstname$:=name$
if name$>lastname$ then lastname$:=name$

endfor names

print "first name on list is";firstname$
print
print "last name on list is";lastname$

end " "
```

*****

-£- Because we already have had one name entered, then the loop must start at one greater --hence

2 to howmany

with "howmany" as the ending control variable in the FOR-DO loop.

-££- Because, in a previous line, we stated

"enter name 1? "

we add the variable "name" from the FOR/DO loop in order to get the following output

enter name 2?

enter name 3?

et cetera

Enter the program and RUN it. For the first two or three times, use inputs of 3 or 5 names --then try one of 15 names.

24.6 **THIS IS FINE** for finding (alphabetically) the first and last names; but, in reality, what we really want is a program to order an entire list of names.

This is called sorting.

There are all sorts of sorts: bubblesort; insertion sort; quicksort; merge sort; block sort; collating sort; tag sort; shakersort; and, selection sort --to name a few

The primary goal of a sort is to order data as quickly as possible. (In fact, the esoteric, unspoken, "shangri-la", goal of programmers is to write an algorithm that will be able to take an infinite list of items and order it instantly! None of them will admit to it, but it is there, nevertheless.) To that (secret, but unconfessed) end, a great deal of time and effort has been expended in refining the many sorting algorithms

available --in hopes of developing that ONE program which will solve it all!

The truth is, each has their strengths and their weaknesses. Highly refined sort algorithms are so long they do not work well for limited data; simpler programs are not efficient for long lists of data.

Studying the many different sorts is best left to an advanced text. This chapter will introduce the basic concepts behind a sort through the use of an elementary, but quite efficient one --the SELECTION SORT.

A SELECTION SORT does just that --selects. It will search an array for the first item, then it will look for the next and so forth, until the array is ordered in the manner the programmer originally defined. (This may mean, for example, the programmer wishes the items to be listed in ascending or reverse alphabetical order --or, perhaps, in some other manner.)

The first thing we will need to do is store the items in an array; having done that, we then can exchange them from cell to cell, until they are ordered according to the specified criteria.

For purposes of illustration, we will order the following five names in ascending order.

ted, lou, bob, zed, jon

With but five names, the final order can easily be seen to be

bob, jon, lou, ted, zed

As we have done before, let's begin with the algorithm. This time, we will list program line numbers to facilitate the explanation.

```
1000 PAGE
1010
1020 DIM name$(5) OF 3,temp$ OF 3
1030
1040 FOR lyst:=1 TO 5 DO READ name$(lyst)
1050
1060 FOR begin:=1 TO 4 DO
1070
1080   smaller:=begin
1090
1100   FOR swap:=begin TO 5 DO
1110
1120     IF name$(swap)<name$(smaller) THEN smaller:=swap
1130
1140   ENDFOR swap
1150
1160   exchange
1170
1180 ENDFOR begin
1190
1200 PROC exchange
```

```
1210
1220    temp$:=name$(begin)
1230    name$(begin):=name$(smaller)
1240    name$(smaller):=temp$
1250
1260 ENDPROC exchange
1270
1280 FOR sorted:=1 TO 5 DO PRINT name$(sorted)
1290
1300 DATA "ted","lou","bob","zed","jon"
1310
1320 end " "
```

*****

LINE(S)	COMMENTARY
1020	We must DIMension the array, NAME$. Because we are going to work with 5 names, we will set aside 5 cells. In PROCEDURE EXCHANGE, we will be using TEMP$, it too must be DIMensioned for the maximum length of any name. In this illustration we have purposely kept all names to 3 letters. Note the array NAME$ is DIMensioned for 3 characters.
1040	This is one-liner FOR-DO loop to place the 5 names in 5 cells (1 through 5) of the array.
1060	This loop begins the selection and exchange process. If we are to order five names, then placing four of them in proper sequence automatically takes care of the fifth one!
1080	The role this statement plays will be seen soon.
1100-40	We initialize the variable swap in the FOR-DO loop at begin because, once we exchange "bob" from its present position in cell 3 to cell 1, we are NO LONGER interested in cell 1 but, instead, want to put the next name (jon) in cell 2. The variable begin will change in value from 1 to 2 to 3 to 4 on successive trips of the outside loop --therefore the nested loop will have the appropriate beginning value on each of its trips.

1120 This line is the first of two very essential pieces of logic. We shall follow it in some detail

But before we do, note the use of the subscript SMALLER in the second part of the IF portion of the IF-THEN statement. We set SMALLER equal to BEGIN on line 1050. This means that when "bob" (as the first example) is placed in cell 1, the succeeding values of SMALLER will continue to be one larger than each name whose position has been moved. This is so, because BEGIN in the outer loop takes on values one greater than on the previous trip. With SMALLER being replaced with the new values of BEGIN each time around the outer FOR-DO loop, there will be no mixup in where to place each succeeding name.

Here is a detailed run of line 1120 for the first trip of the outer loop.

```
*****************************************************
*                                                   *
* IF name$(swap)<name$(smaller) THEN smaller:=swap *
*                                                   *
*****************************************************

1.   IF name$(1) < name$(1)        THEN  smaller:=swap

     IF  ted     <  ted            THEN  (not so --no action)

-----

2.   IF name$(2) < name$(1)        THEN  smaller:=swap

     IF  lou     <  ted            THEN  smaller:=2

-----

3.   IF name$(3) < name$(2)        THEN  smaller:=swap

     IF  bob     <  lou            THEN  smaller:=3

-----

4.   IF name$(4) < name$(3)        THEN  smaller:=swap

     IF  zed     <  bob            THEN  (not so --no action)

-----

5.   IF name$(5) < name$(3)        THEN  smaller:=swap

     IF  jon     <  bob            THEN  (not so --no action)
```

```
****************************************************
*                                                  *
* At the conclusion of the first run the variables *
* SMALLER and BEGIN have these values:             *
*                                                  *
*            smaller=3                             *
*                                                  *
*            begin=1                               *
*                                                  *
****************************************************
```

At this point, the computer falls into line 1160

exchange

Of course, we recognize this as a PROCEDURE call.

```
PROC exchange

  temp$:=name$(begin)             temp$:=name$(1)         temp$ = ted
  name$(begin):=name$(smaller)    name$(1):=name$(3)      name$(1) = bob
  name$(smaller):=temp$           name$(3):=temp$         name$(3) = ted

ENDPROC exchange
```

At the conclusion of the EXCHANGE, "ted" has been placed in CELL 3 and "bob" in CELL 1. The computer, after having completed PROCEDURE EXCHANGE, returns to line 1080, falls through to 1090 and begins a second trip through the FOR begin-DO loop. Now BEGIN equals 2 and because of line 1080, so does SMALLER. The selection process begins again, only this time the names in the array NAME$ have been re-ordered to read

bob, lou, ted, zed, jon.

Here is the run for the second trip of the outer loop.

begin = 2

smaller = 2

```
****************************************************
*                                                  *
* IF name$(swap)<name$(smaller) THEN smaller:=swap *
*                                                  *
****************************************************
```

```
1.      IF name$(2) < name$(2)       THEN smaller:=swap

        IF  lou     <  lou           THEN (not so --no action)

-----

2.      IF name$(3) < name$(2)       THEN smaller:=swap

        IF  ted     <  lou           THEN (not so --no action)
```

```
3.        IF name$(4) < name$(2)        THEN smaller:=swap

          IF  zed     <  lou            THEN (not so --no action)
```

```
4.        IF name$(5) < name$(2)        THEN smaller:=swap

          IF  jon     <  lou            THEN smaller:=5
```

```
****************************************************
*                                                  *
* At the conclusion of the first run the variables *
* SMALLER and BEGIN have these values:             *
*                                                  *
*             smaller=5                            *
*                                                  *
*             begin = 2                            *
*                                                  *
****************************************************
```

At this point, the computer falls into line 1160

exchange

again.

```
PROC exchange

  temp$:=name$(begin)              temp$:=name$(2)       temp$ = lou
  name$(begin):=name$(smaller)     name$(2):=name$(5)    name$(2) = jon
  name$(smaller):=temp$            name$(5):=temp$       name$(5) = ted

ENDPROC exchange
```

At the conclusion of the EXCHANGE, "jon" has been placed in CELL 2 and "lou" in CELL 5. Once more the computer returns to line 1080, falls through to 1100 and begins a third trip through the FOR begin-DO loop. BEGIN is now 3 as is SMALLER. The selection process begins again, only this time the items in the array NAME$, have been re-ordered to read

bob, jon, ted, zed, lou

Because of the speed with which a computer works, lists of moderate length can be ordered fairly quickly. But, because a SELECTION SORT must evaluate each item (less one) EVERY TIME, it doesn't take much to realize that this sort is not efficient for long lists. Still, it is a sort and if time is not a critical factor, its simplicity makes it quite useful.

After the fourth trip, the computer falls into line

1280   A one-liner FOR-DO loop to print the names that have been rearranged in the CELLs in alphabetical order.

```
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
.                                                           .
.  RECAP                                                    .
.  *****                                                    .
.                                                           .
.  THE better method for determining the largest            .
.      and smallest -or, first and last items is            .
.      to make comparisons after having READ the            .
.      first item outside of the program loop.              .
.                                                           .
.  THE role of sorts is to order an entire list.            .
.                                                           .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
```

problems
****

1. Find the smallest and largest numbers in the following list

    84, 375, 83, 569, 106, 215, 314, 806, 800, 303

2. Find the smallest and largest numbers in the following list

    4.77, 3.03, 8.765, 7.653, 1.18, 0.314, 0.194, 7.77, 9.936

3. Take the same numbers in problem 2 and print the following

    their sum;

    their average;

    the largest number; and,

    the smallest number.

    (ANSWERS: SUM = 43.612; AVERAGE = 4.84577778; SMALLEST = 0.194; LARGEST = 9.936)

4. Generate 10 random numbers between 164 and 219 and print them to the screen. Determine which is the largest and the smallest. (A FOR-DO may be the best bet here.)

5. Generate 10 random numbers between 23 and 98 and print them to the screen. Determine their average. Tell how much the largest and then the smallest number differs from that average.

6. Order the following names in:

6.1 Ascending order; and,

6.2 Reverse alphabetical order.

jack, george, alfred, heather, beth, may, albert, zelda, zelga, henry, alf, john, josh, mary, vada, winnie-goo

7. Place the months of the year (in their present order) in a data line and then arrange them alphabetically.

8. Go back to Chapter 22, problem 1 and enter the names of the towns as they appear. Print their complete names in alphabetical order.

9. Generate, at random, 20 three-letter nonsense words and store them in an array. Print the "words" to the screen as they are generated. Then print them in alphabetical order.

(Hint: Use a nested loop. This loop will be from 1:=3. Inside that loop include the line

```
print chr$(rnd(65,90)),
```

Make the outside loop 1:=20.)

	SUN	MON	TUE	WED	THU	FRI	SAT
pass 1	MON	SUN	TUE	WED	THU	FRI	SAT
pass 2	MON	FRI	TUE	WED	THU	SUN	SAT
pass 3	MON	FRI	SAT	WED	THU	SUN	TUE
pass 4	MON	FRI	SAT	SUN	THU	WED	TUE
pass 5	MON	FRI	SAT	SUN	THU	TUE	WED

chapter twenty-five

introduction to files

****

objectives

At the completion of the chapter the student will be able to:

1. Define the relationship between a record and a file;
2. Tell the difference between a sequential and a random file;
3. State, in proper order, the six steps to be followed in writing a file;
4. Write a sequential file to enter data;
5. Write a program to retrieve data placed in a sequential file;
6. Use either a FOR-DO or REPEAT-UNTIL loop in a file program;
7. Write a random file to enter data;
8. Write a program to retrieve data placed in a random file:

    8.1 All the data;

    8.2 Any specific item in the file; and,

    8.3 Specific items that contain a common sub-$tring of the larger $tring.

    and,

9. Use either a FOR-DO or REPEAT-UNTIL loop in a random file program.

```
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
c                                                               c
o  CHANGING COMPUTER FROM COMAL TO BASIC                        o
m  -------------------------------------                        m
a                                                               a
l  To run programs in BASIC,  the COMAL cartridge  has  to  be  l
c    "turned off" first.  To do this, type                      c
o                                                               o
m            basic + <RETURN>                                   m
a                                                               a
l    Once the BASIC heading appears on the screen, programs in  l
c    that language can be used.                                 c
o                                                               o
m    To return to COMAL, type                                   m
a                                                               a
l            sys 50000 + <RETURN>                               l
c                                                               c
comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal
```

25.1 **WE SHALL CONSIDER**, in a most cursory manner, two types of files.

1. sequential

2. random.

Files, to merit usefulness, need to be of a substantive nature. Creating a file --say, for 10 names or 8 pieces of information, though of some limited value, is not really worthy of the effort --much like buying a computer system (complete with a printer) just for the purpose of balancing a home checking book, doesn't justify the investment of a thousand dollars.

Though file development is not "horrendously difficult", the longer the file and the more substantial it is --the more need there is for memory space and the more appreciable the problems become in handling data input and retrieval. Writing and developing files is rather a unique discipline in itself; still, with a little work, the technique can be mastered and we can end up designing files that fits our particular demands.

We will introduce and explain the principles of files through the creation of some very elementary ones.

25.2 **A RECORD IS** an element of a **FILE** --much like the JACK OF CLUBS is an element of a deck of cards, or 12th grade is an element of HIGH SCHOOL.

RECORDS may contain a single item of information (a letter of the alphabet, a question mark, et cetera) or a "whole lot of things."

25.3 **FILES ARE A** compilation of RECORDS. The HIGH SCHOOL **FILE** is a compilation of 9th grade,

10th grade, 11th grade and 12th grade.

If, to talk with a student in the 11th grade, we had to start with the first student (alphabetically) in the 9th grade and "walk past" each student in the 9th grade, the 10th grade and partially through the 11th grade until we "retrieved" the 11th grade student we wanted to talk with, the FILE, HIGH SCHOOL, would be called

SEQUENTIAL

If, on the other hand, we can go directly to that student (by-passing the 9th grade, the 10th grade and the rest of the students in the 11th grade) then FILE, HIGH SCHOOL, would be called

RANDOM

25.4 THERE ARE SIX steps we must attend to in the developing of a file.

STEP 1 -- CREATE A FILE.

In some languages there must be a separate statement for creating a file. COMAL does not require this.

STEP 2 -- OPEN THE FILE.

When, in COMAL, we OPEN a file, it is also "created."

What we are doing is "reserving" a place within the computer; that is, we are informing the computer we are going to create a file and we want space reserved for this activity.

The mechanical statement is

OPEN FILE 7,

"OPEN FILE" is a system command and will initiate the necessary action to set aside space for a file.

The "7" is the channel number. It can be any number from 1 through 255.

STEP 3 -- NAME THE FILE.

The file must have a name --and, it seems obvious, the name should be relevant to what the file is about. If the file concerns birthdays, then the above statement should be added to, to read

OPEN FILE 7, "BIRTHDAYS",

The commas are essential.

STEP 4 -- WRITE TO A FILE.

Having established the protocol to begin a file, the next logical thing to do is to put data into the file. This is called **WRITING TO A FILE.**

The statement can now be completed to read

```
OPEN FILE 7, "BIRTHDAYS", WRITE
```

(If the file is to contain $trings then, of course, it must be DIMensioned.)

```
DIM name$ of 20
OPEN FILE 7, "BIRTHDAYS", WRITE
```

STEP 5 -- READ FROM A FILE.

Having placed data into a file, we certainly want to be able to take it out --printing either to the screen or to a printer.

To do this, the following statement is typed

```
OPEN FILE 7, "BIRTHDAYS", READ
```

If the file contains $trings, then it must be DImensioned.

```
DIM name$ of 20
OPEN FILE 7, "BIRTHDAYS", READ
```

STEP 6 -- CLOSING A FILE.

Once the file has been OPENed, it is important to advise the computer when we are finished with either entering data or reading the data.

CLOSE is a housekeeping task, that keeps the file in order, protecting it for future use.

25.5 **SEQUENTIAL FILES ARE** just that --sequential. In order to read --say, the fifth record, the computer must read the first four. Once it has located the fifth, it will print it to the screen or to a printer --whatever we have specified.

To clarify the file concept, we shall create (WRITE) a file and then take the data and (READ) the file --printing it to screen. Enter the following program BUT BEFORE RUNNING OR SAVING IT, BE CERTAIN TO WAIT UNTIL THE INSTRUCTIONS ARE GIVEN!!

```
// sequential file

// sample $tring file
```

```
dim letter$ of 1
open file 7, "alpha", write

for recnum:=1 to 10 do
letter$:=chr$(rnd(65,90))
write file 7: letter$
endfor recnum

close
```

If the program was typed correctly it will look like this when LISTed.

```
// sequential file

// sample $tring file

DIM letter$ OF 1
OPEN FILE 7, "alpha", WRITE

FOR recnum:=1 to 10 DO
  letter$:=CHR$(RND(65,90))
  WRITE FILE 7: letter$
ENDFOR recnum

CLOSE
```

READ CAREFULLY!

1. SAVE the file first. Use

   SAVE "alpha file"

   By SAVEing it, it will be placed on the disk as a PRG, instead of a SEQ file. This means we can recall it and make changes to the program if we wish --or just have it for study and/or reference.

2. RUN the file.

   In this particular program, ten letters will be generated at random from the alphabet and stored in RECORD NUMBERS 1 through 10 (recnum). Further, the FILE will be created and placed on the disk.

Now that the file has been created, the next thing we want is to be able to get the information (READ) out. This will call for the following program.

```
// sequential file

// READing information in ALPHA file

dim letter$ of 1
open file 7, "alpha", read
for recnum:=1 to 10 do
```

```
read file 7: letter$

print letter$;
endfor recnum

close
```

WAIT!!

Before RUNning, SAVE the program.

```
save "getalphafile"
```

And for the same reasons as given previously.

RUN the program.

If we are not certain how many entries there are in the file, we can alter our program to be

```
// sequential file

// READing information in ALPHA file

dim letter$ of 1
open file 7, "alpha", read

REPEAT
read file 7: letter$
print letter$;
UNTIL EOF(7)

close
```

EOF is a system command in the same sense as **EOD**. It means END OF FILE.

In fact, any COMAL logic is just as applicable here to FILE programs as to other aspects of programming.

Type NEW, and then LOAD "getalphafile".

Make the necessary changes to the two lines in the program (replace them with REPEAT and UNTIL EOF(7)). RUN this program again.

Note the printout is the same as for the original READ file program. In fact if we RUN this several times, we will continue to get the same printout. The author had

d u a d g m q x c g

Logically, this is so because in the WRITE file we generated 10 letters and STORDED THEM AS 10 RECORDS. Having stored (FILEd) them, they are not going to change!

At this point type **NEW** again. Load "alpha file" from the disk. And on the line which reads

```
OPEN FILE 7, "alpha", WRITE
```

type

```
OPEN FILE 7, "@0:alpha", WRITE
```

Don't do anything more as yet!

Now, when it is time, the --@0:-- will allow the original "alpha" file to be over-written. So, at this point, type

```
save "@0:alpha file"
```

Here, also, the --@0:-- will over-write our "alpha" save statement on the disk. (There is no point in "cluttering" our disk with repetitive files.)

RUNning the "@0:alpha" (please do) will create another 10 random letters to be stored.

Type NEW and LOAD the "getalphafile". RUN it and note there is a different output.

25.6 HERE IS ANOTHER simple program. Copy it to the computer.

```
// another sequential file

// storing a name

dim name$ of 40
open file 2, "name", write
input "type your name":name$
print file 2: name$
close
```

SAVE the program first, call it

```
SAVE "name$ file"
```

Then RUN it. When asked for your name, type it in and tap RETURN.

To retrieve, type (after the "NEW" command),

```
// read sequential file, "name"

// name file

dim name$ of 40
open file 2, "name", read
input file 2, names$
```

```
print name$
close
```

RUN this, and the name entered, will be retrieved to screen.

Let's modify this a bit, so we can type in the names of five students sitting in this computer class.

```
// sequential file to
// list 5 student names

dim name$ of 40
open file 2, "@0:name", write
for recnum:=1 to 5 do
input "type the names ":name$
print file 2, name$
endfor recnum

close
```

And a small program to retrieve them.

```
// sequential file to
// print names of students

dim name$ of 40
open file 2, "name",read
for recnum:=1 to 5 do
input file 2: name$
print name$
endfor recnum

close
```

25.7 **AN EXPANSION OF** this basic concept can lead to a file containing several bits of information in the same record.

We shall create a file of the starting five basketball players for a high school. In the record we would like

1. jersey number
2. name
3. height
4. shooting percentage

We will call them, respectively,

1. jrsynum$
2. name$
3. tall$
4. shotp$

Because we know exactly how many there are, we can enter them as DATA

line items.

```
// Basketball player file

dim jrsynum$ of 2,name$ of 20,tall$ of 7,shotp$ of 3

open file 3,"teamplayer",write
for rec:=1 to 5 do
read jrsynum$,name$,tall$,shotp$
write file 3:jrsynum$,name$,tall$,shotp$
endfor rec

close

data "11","ted short","6ft11in","80%"
data "15","bob tall","7ft3in","74%"
data "21","lou taller","7ft8in","87%"
data "32","sam shorter","6ft3in","92%"
data "41","ray tallest","7ft11in","98%"
```

And, to retrieve them

```
// basketball player file

page

dim jrsynum$ of 2,name$ of 20,tall$ of 7, shotp$ of 3

open file 3,"teamplayer",write
for rec:=1 to 5 do
read file 3:jrsynum$,name$,tall$,shotp$
print jrsynum$;name$,tall$,shotp$
endfor rec

close
```

Remember, even though we only may want one line of data, the whole file (in a sequential file) must be read before that information can be found and printed.

Delete the line that calls for

```
print jrsynum$;name$;tall$;shotp$
```

and in its place, using AUTO increments of 1, add

```
if jrsynum$="21" then
print name$;shotp$
elif jrsynum$="41" then
print name$;shotp$
endif
```

RUN the program and note the printout.

25.8 **A RANDOM FILE** would be extremely useful for a school librarian. If a student wanted books on a certain subject, the librarian --through the use of a computer-- could call out all the books that contained that word in their title. And, with further ramification of the library file system when it is first installed (and kept up to date as new books are accessioned) could list other titles that addressed the subject matter in question. This is why files are so powerful --but, also, why they take a great deal of work when designed to accept information, and to search the file for the purpose of retrieving information. A SEQUENTIAL FILE for a 50,000-volume library would make tedious work of even the simplest of tasks. A RANDOM FILE, on the other hand, is the tool for the job.

25.9 **THE LINE TO** open a random file takes the format of

```
OPEN FILE 4, "STUDENT LIBRARY",RANDOM 45
```

We recognize the first part as being the same as required for a sequential file, namely:

```
OPEN FILE 4, "STUDENT LIBRARY",
```

The next part

RANDOM

is the COMAL command for setting up a RANDOM FILE for both **WRITE**ing to, or **READ**ing from a FILE.

The

45

indicates the size of each entry. The longest title (+ 2) would be the guide for entry length. (Keep in mind though, provisions for the size of the entries must be increased if --say, the author and the Dewey number are included in each record. We will talk to that in a bit.)

Commercial library programs that already exist for the purpose of shelf lists and the daily checkout of titles (complete with provisions for entering the due date and name of the borrower) have designs that allow for entries on a do-as-you-can time basis. The following illustrations are for demonstrating the essential characteristics of a random file, and do not pretend to offer the sophistication needed for the typical school library. Indeed, the space provisions required, exceed the memory capacity of many micro-computers.

Enter the following program.

```
// mini random file demonstration

// using library books
```

```
        input "number of books in data lines? ":number
-£-     dim title$ of 45
-££-    open file 5, "books", random 45
        for num:=1 to number do
-£££-   read title$
-££££-  WRITE file 5,num:title$
        endfor num
        close
```

Before SAVEing this, enter the following titles as DATA statements --one to a program line.

```
        Understanding Pascal
        Computer Studies with Comal
        Basic for Students with Applications
        Pascal
        Okidata Personal Printer User's Manual
        Structured Programming with Comal
        The Amazing Adventures of Captain Comal
        Using Basic
        Introduction to Pascal
        Comal 80
        Commodore 64 Programmer's Reference Guide
        Pascal A Considerate Approach
        Beginning Comal
        The Comal Handbook
        Easy Script, Commodore 64
        Compute!'s First Book of VIC
```

SAVE the program first, before RUNning it. This way we can retrieve it if there are any errors.

When we list the CAT (or DIR), note the file is saved as a **REL** FILE. RANDOM files are saved as RELATIVE files.

*****

-£- We are making provision for 45 bytes of information in the name of each title.

This, of course, would be for the longest name. If a name "gets out of hand", then a meaningful abbreviation is certainly called for.

In general, integers are counted as 2 bytes, real numbers as 4 bytes, and $trings (including the spaces between each $tring) two more than the actual number of characters.

At this point, however, 45 is a generous allowance. For hundreds of titles, space allocation becomes quite critical.

-££- Note there is no comma after RANDOM. The name of the file is BOOKS.

-£££- The READ is not the same as in sequential files; here it is to get the information put on the DATA lines.

-££££- WRITE is the command to put the information into the file. Also observe that each entry is preceded by an integer --specified by "num" from the variable of the FOR-DO loop.

25.10 HAVING CREATED THE file "BOOKS", let's write a program to retrieve the entire file to the screen.

Enter the following. Be certain to SAVE it first, before RUNning.

```
        // mini random file demonstration

        // getting a printout of library books

        input "number of books ";number

        dim title$ of 45

-£-     open file 5,"books",random 45

-££-    for num:=1 to number do

        input file 5,num:title$
-£££-   print num;title$

        endfor num

        close
```

*****

-£- When the computer "sees" this command, it goes to the disk, searches for the file "books" and opens it up for retrieval of information --or to add more to it if that is what the user wants to do.

-££- We could change the varibles in the FOR-DO loop to start at a number other than one.

-£££- It prints both the record (num) number and the title.

25.11 **IF A STUDENT** wanted to know what books there were on COMAL, the librarian could step over to the computer, enter a program similar to the following and get the selected list.

```
         page

         // limited library file

         // getting title of books containing "COMAL"

         input "how many records to search? ": number

         dim title$ of 45, wordintitle$ of 15

         count:=0

         open file 5, "books",random 45

-£-      repeat
         count:+1

         input file 5,num:title$
-££-     searchtitle("comal",title$)

         until count=number

         close

         end "end search for titles"

-£££-    proc searchtitle(wordintitle$,bookname$)

-££££-   if wordintitle$ in bookname$ then
         print
         print count;booknames$
         endif

         endproc searchtitle
```

*****

-£- A FOR-DO could be used just as well.

-££- We recognize this as a PROCEDURE call. We have not studied about parameters listed in parentheses after a procedure name.

What we have is parameter passing. There is a great deal of power in this, but a discussion will have to be left to an advanced text.

Meanwhile, note the statement format in terms of how it is entered.

-£££- If we put the two lines one above the other, we have

```
     searchtitle("comal"       ,title$)
                     †             †
proc searchtitle(wordintitle$,bookname$)
```

-££££- Here WORDINTITLE$ corresponds to "comal" and BOOKNAME$ corresponds to "title$".

What this command does, is to say, "If you find COMAL in any of the names of the books (title$) then print out the number (count) of the book and its title (bookname$)."

After this is RUN, we will get the following titles to the screen.

2. computer studies with comal

6. structured programming with comal

7. the amazing adventures of captain comal

10. comal 80

13. beginning comal

14. the comal handbook

25.12 **IF THE PROGRAM** in 25.10 listed the entire file to screen, and the program in 25.11 printed the names of the books containing the word "COMAL in their title, then it is apparent each record had to be searched --just like in a sequential file.

To gain direct access to an entry in the random file "books", a program similar to the following could be used.

```
page

// limited library file

// getting a specific book by entry number

input "what book entry do you want? ":number

dim title$ of 45

open file 5,"books",random 45

input file 5,number:title$

print
print number;title$

close

cursor 21,1
end "end search for book name"
```

If we enter "13" to the inquiry,

    what book entry do you wish?

The title,

    13. BEGINNING COMAL

will be put on the screen.

In a library system containing thousands of book, we can modify the above program to accept a title entry and return the Dewey catalog number.

25.13 CERTAINLY, THIS IS, as stated earlier, only a cursory discussion on files. And in terms of the example of a school library, it must be evident a great deal more elegance is needed in the programs to make the file of any real value. Book titles should contain the DEWEY number. Searching a whole library for books on COMAL is not sensible. Major divisions of a book cataloging system would be recorded on separate discs. For example, if computer languages are listed in the 500's, then a disc with just those listings makes sense.

RECAP
*****

SEQUENTIAL files must be searched one record at a time, beginning with first record.

RANDOM files may be accessed in any order.

ALL files must be OPENed to either WRITE to or READ from them.

AFTER WRITEing to or READing from a file it must be CLOSEd.

problems
****

PROBLEMS FOR SEQUENTIAL FILES

1. Create a file to store 15 letters selected randomly from the alphabet. Print the letters in sequence as filed and then in reverse sequence.

2. Create a file to store 20 random numbers from 20 through 80.

   2.1 Print a table of two rows.

   2.2 Print the numbers in the first row and a running sum in the second.

3. Create a file to store the names of ten teachers --and retrieve the list.

4. Create a file to store your school schedule (or your morning routine on Sunday). Retrieve it.

5. From the file in problem (4), print only your 4th and 7th period data (or what you do from 9-10 and 11 -12 on Sunday).

7. Get the roster of a football team. Place 11 junior and/or senior players in a file. Have the following data in the file:

   grade as a non-string item
   name as a $tring
   weight as a non-string item
   position as a $tring item.

7.1 List the entire file.

7.2 List those in grade 12

7.3 List those who weigh more than 150 pounds.

PROBLEM FOR A RANDOM FILE

8. Create a RANDOM file of 25 entries. The subject matter is left to you --a record file, tape file, birthday file with dates, boy/girl-friend file with phone numbers, or something else that lends itself to being filed.

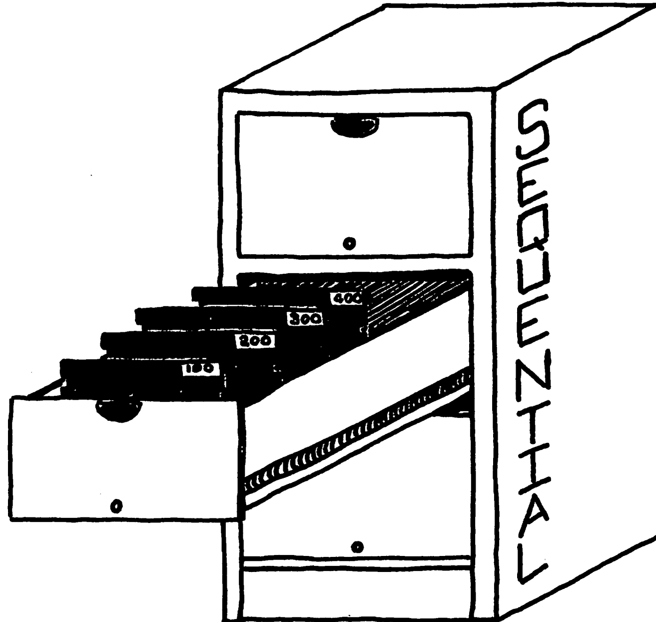
Write four programs to-

8.1 Create the file;

8.2 Get the entire file printed;

8.3 Get selected printouts that reflect some one word common to several items; and,

8.4 Get a specific file by entry number.



chapter twenty-six

the graphic turtle
****

objectives

At the completion of the chapter the student will be able to:

1. Move the turtle forward, backwards, at a right angle --either to the left or to the right;
2. Move the turtle at any angle;
3. Hide the turtle and recover the turtle;
4. Change the colors of the border, the background and the pencolor;
5. Write a program to produce a polygon of any number of sides;
6. Write a program to produce any portion of a circle and at any place on the screen;
7. Paint (fill); and,
8. Write a program to create an abstract "painting."

----

comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal

TURTLE SIZE

The turtle can be set to 11 different sizes. The command for this is

```
turtlesize(<size>)
```

<size> runs from 0 (smallest) through 10 (default size).

comalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomalcomal

26.1 OF THE ELEVEN packages the COMAL 80 cartridge has, the TURTLE is a great deal of "fun" and interest. Access to this package is by one of two methods.

1. Type USE TURTLE, or

2. Tap the **f3** key.

A triangular "turtle" appears and is "sitting" in the middle of the screen (which is its HOME position) facing north.

The turtle draws lines. And to make it do so, we have to give it commands.

We shall start with ten commands. After we have demonstrated how they work, take time to produce more pictures. With the background we now have, we know there must be an easier way to get these same pictures without having so many individual commands. However, time spent on these fundamentals will pay off when we introduce some programming techniques later.

Turtle Graphics (or LOGO) is another discipline which can be studied in great detail. The purpose of this chapter is to introduce the subject. For further in-depth study, a good tutorial is recommended.

26.2 TO GET THE turtle to move forward, we will tell it to do just that. (Enter these commands. Be certain to + <RETURN> after each entry.)

```
FORWARD(40)
```

It will also move backwards.

```
BACK(50)
```

It will turn a 90 degree angle --either right or left.

```
RIGHT(90)
```

and then,

```
FORWARD(10)

LEFT(90)
```

and then,

```
FORWARD(50)

LEFT(90)
```

and then,

```
FORWARD(10)
```

To hide the turtle so that we may see our final figure clearly, type just that.

HIDETURTLE

The turtle disappears, and we have a narrow rectangle.

To clear the screen, type

CLEAR

Then, to get the turtle back to its HOME position, type

penup

home

showturtle

and we are ready to start another picture.

When TURTLE is first loaded, the pen is down --i. e., when the turtle is moved, it will draw a line. If we want to move the turtle and not draw a line, then we must give the command

penup

To start drawing again, don't forget to issue the command

pendown.

Because we cleared the screen after we hid the turtle (hideturtle) we must give the command

showturtle

26.3

THERE ARE SOME abbreviations for these commands which will ease the typing chores a bit.

FORWARD	=	FD
BACK	=	BK
RIGHT	=	RT
LEFT	=	LT
PENDOWN	=	PD
PENUP	=	PU
HIDETURTLE	=	HT

SHOWTURTLE = ST

CLEARSCREEN = CS

Now, take these commands and create several pictures. Be certain to employ as many of them as possible. By the way, we are not restricted only to 90 degree angles. We can issue commands of any angle from 0 through 360 degrees.

RIGHT(72)

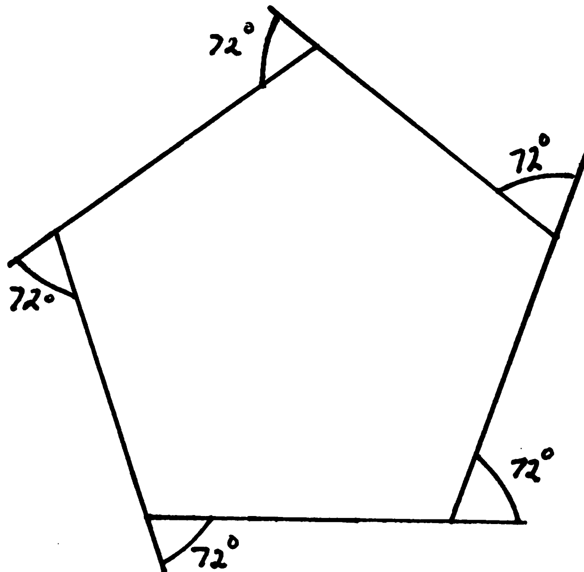
LEFT(105)

LEFT(84)

are valid commands.

26.4 IN DRAWING POLYGONS an essential concept is the sum of the exterior angles. No matter how many sides a polygon has, the sum of the exterior angles is always equal to 360 degrees.

Study this figure of a regular polygon (equal sides and equal interior angles) of 5 sides --a pentagon.



Therefore, when drawing polygons, the angle specified for the turtle to turn is the value of one EXTERIOR ANGLE.

Enter these two lines (after typing NEW and tapping f3).

```
fd(50)
rt(72)
```

Now, rather than type them four more times, take the CURSOR up to the **fd(50)** command and tap RETURN. Do the same for the **rt(72)**. Repeat the procedure until the pentagon is drawn. (**ht** will remove the turtle.)

If it is a regular pentagon, then why does the shape look distorted? It is because of the pixels that make up the screen. They are not actually square. The "up-and-down" dimension is greater than the "right-and-left" dimension. To demonstrate this even further, type these two lines (don't forget NEW and f3) --which should give one side of a square-- and employ the same technique (using the CURSOR) as above, with

```
fd(50)
rt(90)
```

We should have ended up with a square, 50 units on a side. Well, we did --but the left-and-right pixel dimension is "skinnier" so we ended up with a rectangular square (a contradiction in terms, of course). To get a "square" on the screen, enter this (remember, NEW and f3).

```
fd(37)
rt(90)
fd(50)
rt(90)
fd(37)
rt(90)
fd(50)
ht
```

In addition to pixel dimensions, the distortion present in the monitor tube also contributes to making a square somewhat rectangular in shape.

CLEAR the screen, HOME the turtle, and show (ST) it.

26.5 **THE TURTLE ALSO** can draw arcs and circles. To draw a circle from any current position that the turtle is in, type

```
arcl(60,360)
```

Again, the circle is not a "circle", looking more like an ellipse because of the pixels.

The first number in the parentheses is the length of the radius; the second, the size of the angle. If we had wanted a semi-circle, we would have made the second number 180.

Without clearing the screen, take the cursor up to the first figure (the radius) and put in different values. If we specify enough, the resulting figure looks either like we are peering into a tunnel or taking a skew look at the outside of a cone --depending on what "trick" our eyes play.

If we want the circles drawn to the right instead of to the left, then issue the command

```
arcr(60,360)
```

If we wish to specify a circle or arc be drawn at a position different from where the turtle currently is, then we issue the following command.

```
arc(<centerX>,<centerY>,<radius>,<start angle>,<size>)
```

Before analyzing the parameters of the command, let's lay out the monitor screen in terms of the pixel location values.

```
-159,99                  159,99
     +------------------------+
     !                        !
     !                        !
     !                        !
     !           *            !
     !          0,0           !
     !                        !
     !                        !
     +------------------------+
-159,-99                 159,-99
```

If, in using the above command, we want to draw a circle of radius 60 in the middle of the screen, the command would be

```
arc(0,0,30,0,360)
```

Enter the command and run it. Change the value of the radius until several concentric circles are drawn.

Without clearing the screen, move the center of the circle to the left by typing

```
arc(-50,0,30,0,360)
```

Again, change the value of the radius several times.

Now move the center to the right.

```
arc(50,0,30,0,360)
```

And up on the Y-axis, but in the middle of the X-axis,

```
arc(0,50,30,0,360)
```

And, finally down the Y-axis, but still in the middle of the X-axis.

```
arc(0,-50,30,0,360)
```

The **START ANGLE** can be anything we wish. "0" starts at 3

o'clock; 90 at 12 o'clock, and so on. **SIZE**, refers to how much of an arc we wish drawn. 360 is a full circle; 180, half a circle (180/360 = 1/2); and 60 = 1/6 of a circle.

Experiment with these three commands until they are mastered.

26.6

COLOR IS POSSIBLE with three other commands. We can choose from 16 different colors.

colour	number
black	0
white	1
red	2
cyan	3
purple	4
green	5
blue	6
yellow	7
orange	8
brown	9
pink	10
dark grey	11
medium grey	12
light green	13
light blue	14
light grey	15

BACKGROUND (BG) can be selected by typing that word or its abbreviation, plus a color number in parentheses; the edge around the screen, by typing BORDER(#) --the "#" being a number from 0 through 15.

The lines can also be in color. The command for this is PENCOLOR(#).

Enter the following program.

```
use turtle
clear
home
st
pencolor(1)
arc(0,0,30,0,360)
ht
fill(40,0)
```

(When the picture is complete, tap **f5**.)

This introduces the FILL command. It will fill the screen until it reaches the edge of the screen or encounters the edge of a drawing. And it will use the color of the pencolor --which in this instance is white. Repeat the above program and change the pencolor to some other number.

26.7 **WE ARE NOW** ready to expand into some simple programming.

```
use turtle

// ** main control sequence **

fullscreen
colors
arcleft
home
arcright
paint
end

proc colors
border(3)
bg(13)
pencolor(8)
endproc colors

proc arcleft
for circle:=5 to 35 step 5 do
arcl(circle,360)
endfor circle
endproc arcleft

proc arcright
for circle:=5 to 35 step 5 do
arcr(circle,360)
endfor circle
endproc arcright

proc paint
ht
pencolor (4)
fill (100,0)
endproc paint
```

RUN the program. **FULLSCREEN** allows us to see the turtle while it is working. When it is finished, the picture will disappear and the program will come back on the screen. To get back to the picture, tap **f5**.

Tapping **f1** will bring back the program listing. Experiment with the program, putting in different numbers. Change the 360 to 180, or 120 or some other value. Change the values of the colors.

Here is a program to draw a 10-sided figure --a decagon. Remember, each angle to be turned must be 36 degrees (360/10)).

```
use turtle

// a decagon
```

```
fullscreen
for sides:=1 to 10 do
fd(25)
rt(36)
endfor sides

ht
pencolor(1)
fill(10,0)

end " "
```

When the cursor starts to flash, tap **f5** and when finished with looking at the drawing, **f1**.

It takes a good deal of space on the disc, but we can save our screen drawings by using the command

```
savescreen("hrg.decagon")
```

We place the "hrg." so we will recognize it in a listing of the DIR as being a graphic.

LOADSCREEN("HRG.DECAGON") will take it off the disc, and

```
PRINTSCREEN("HRG.DECAGON")
```

coupled with

```
PRINTSCREEN("LP:80")
```

will make a hard copy.

Here is an interesting drawing.

```
use turtle
fullscreen
for sides:=100 to 1 step -1 do
fd(sides-10)
rt(sides+10)
endfor sides
ht
```

and three more

```
use turtle

// nested loop

fullscreen

for lines:=1 to 10 do
for arcs:=1 to 5 do
arcl(5*arcs,180)
arcr(8*arcs,180)
```

```
 endfor arcs
endfor lines

ht
```

```
use turtle

// nested loops

fullscreen

count:=0

for radius:=0 to 99 step 9 do

 count:+1

 for circles:=1 to 3 do
  pencolor(0+count)

  arc(0,0,radius,90-radius,360)

 endfor circles

endfor radius

ht
```

```
use turtle

// fiddling around

bg(7)

fullscreen

for lines:=1 to 15 do
 for arcs:=1 to 20 do
  fd(10)
  rt(40 + lines)
 endfor arcs

 pencolor(0+lines)

 arcl(2*lines,360)

endfor lines

ht
```

Experiment with different programs and with differing functions.

RECAP
*****

THE command FULLSCREEN allows the programmer to observe the TURTLE creating a drawing.

TAPPING f5 will bring the completed drawing back to the screen.

TO return to the program listing, tap f1.

problems
****

1. Using the samples in the chapter, write programs to produce and paint the following polygons.

    1.1 hexagon

    1.2 octogon

    1.3 duodecagon (12 sides)

    1.4 triangle

2. Create an abstract design.

3. Create a design exclusively of

    arcl

    arcr

    arc

    commands.

4. Write a program that will accept as an INPUT, the number of sides wanted for a polygon. (Hint: Remember, the value of the angle to turn the TURTLE is 360 divided by the number of sides.) Make fd = 50.

    Put in a ten-second delay before the picture leaves the screen.

    Use the FILL command in several places and with different colors.

CHAPTER TWENTY-SEVEN

for further study

****

```
comalcomalcomalcomalcomalcomalcomalcomalcomalcomal
c                                                c
o  SIZE                                          o
m  ----                                          m
a                                                a
l  Returns the amount  of  free memory left  in  l
c    terms of bytes and, at the same time,  the  c
o    the number  of  bytes used in the data and  o
m    the program.                                m
a                                                a
l  It is a direct command.  It can  be  used at  l
c    any time but not in a program.  Type        c
o                                                o
m               size                             m
a                                                a
l    and tap the RETURN key.  The screen  will   l
c    return the following format                 c
o                                                o
m               prog  data  free                 m
a               02937 00541 27236                a
l                                                l
c  (This printout is  at  the conclusion of the  c
o    author's solution  to  problem number 7 in  o
m    chapter 20.)                                m
a                                                a
comalcomalcomalcomalcomalcomalcomalcomalcomalcomal
```

27.1 HAVING COMPLETED THIS introductory text we are now in a position to do some in-depth study of COMAL. COMAL is a professional language in every sense of the word. With our present foundation from which to build, we can benefit from the many excellent tutorials which are in print.

Because new material is always being published, it would be well to contact

COMAL Users Group
USA, Limited
6041 Monona Drive
Madison, WI 53716

(Ph: 608 222-4432)

for an up-dated list of what is currently available.

27.2 THERE ARE TWO reference manuals which are indispensible

THE COMAL HANDBOOK
Second Edition

by: Len Lindsay

publisher: Reston Publishing Company
(c, 1984)

and

COMAL for the Commodore 64

by: Bason & Hojsholt-Poulsen

publisher: Tekst & tryk, Denmark
(c, 1985)

Both are available from COMAL Users Group --though the former is sometimes seen in local bookstores.

27.3 IN PREPARING THIS text, the author consulted four other publications, each of which merits a place in a computer library. This list also serves as a bibliographic listing.

Christensen, Borge. "Beginning Comal". Chichester, West Sussex, England: Ellis Horwood Limited, 1985.

COMAL TODAY (A bi-monthly magazine). COMAL Users Group, USA, Limited: Madison, WI 53716, 1984-1985.

Kelley, John. "Foundations in Computer Studies with COMAL". 2nd edition. Dublin, Ireland: Cahill Printers Limited, 1984.

Atherton, Roy. "Structured Programming with COMAL". Chichester, West Sussex, England: Ellis Horwood Limited, 1982.

INDEX

ABS 55,56
Absolute value 55,56
Algorithm 9
Alphabetical listing 221ff
AND 107,109
Array 181ff
  cells 184ff
  columns 205ff
  counting in, 193
  DIM 184ff
  DIM, multi 206,207
  INPUT 190
  rows 205ff
  summation line 186,187
  summation, columns 206ff
  summation, rows 206ff
  two-dimensional 204ff
ASCII
  code 79, 99ff
  table 101
AUTO 5

BASIC 232
Binary decision 87ff
  digits 99
  ELIF 90ff
  ELSE 90ff
Binary notation 99ff
BIT 99
Bits off/on 100ff
Boole, George 89
Boolean
  false 89,90
  true 89,90
  variables 89,90
Border color 13
BYTE 99
  description 104

CASE 158ff
CASE-OF 158ff
CAT (or DIR) 31
Cells, array 184ff,205ff
Central processing unit 2,98
CHAIN 33,35
CHR$ 70,102ff,137
  table 103
Christensen, Borge 143
Clearscreen 6
CLOSED variable 167,168

CLOSEing file 234
CLR/HOME key 3
Coin Flipping 144ff
Color chart, CURSOR 254
Colors
  border 13
  cursor 13,254
  default 13
  screen 13
Columns, Arrays 205ff
  arrays, summation 206ff
COMAL 1
COMAL HANDBOOK, THE 32
COMAL TO BASIC 232
COMAL TODAY 134
Comment lines 24
Count, array 193
  initializing 81
  statements 81-83
CPU 98
CRSR key 2
CURSOR 2
  back one word 55
  color 13,254
  chart 254
  control 173ff
  forward one word 55
  positioning 129,130

DANSK 1
DATA 113 ff
Decisions, binary 87ff
Default colors 13
DEL 25
DELETEing 25,72
DIM 184,206-207
  arrays 184
  multi-arrays 206,207
  strings 68ff
DIMensioning arrays 184
  multi-arrays 206,207
DIR (or CAT) 31
DISC
  CHAIN 33
  DELETEing programs 72
  formatting 31
  initializing 31
  LOAD 33
  MERGE 33
  PASS 32
  RENAMEing file 152

DISPLAY 158
  procedures 164
DIV 56
division 13,14
EDIT 42ff
  procedures 164
ELIF 91
ELSE 90ff
End of file 236
ENDIF 90ff
Ending messages 112
ENDWHILE 153ff
EOD 153ff
EOF 236
Exponent 13,14

f1 key
  renum 31
  turtle 255,258
f3 key
  turtle 249
f5 key
  list 88
  with turtle 255,258
f6 key 142
f7 key 67,80
f8 key 181
file
  CLOSEing 234
  create 233
  EOF 236
  name 233
  open 233
  RANDOM 233, 244ff
  READ 234
  RENAMEing, disc 152
  WRITE 233,234
files 231ff
  RANDOM 233,240ff
  sequential 232ff
FILL 254
FOR-ENDFOR
  loop 72ff
  multi-line 74
  one-liner 74

Global variable 168
Greater than (>) 88

Heading logo 73

IF-THEN-ELSE 87ff
Inequalities 88
Initializing count 81

INPUT 126ff
  array 190
  at 130
  numerics 126ff
  strings 126ff
INST/DEL key 3
INT 58,59
Integer function 58,59

Less than (<) 88
Lindsay, Len 129,130
Line errors, correcting 217
Line printer 36
LIST 33, 40ff
  procedures 164
  sequential file 73
  with f6 142
LOAD 33
Logic operator AND 106
  operator NOT 106,109
  operator OR 106
Logo heading 73
Loops, nested 142ff
LP 36

Main control sequence 164ff
Mathematical operations 13,14
Mathematics
  addition 13,14
  division 13,14
  exponents 13,14
  multiplication 13,14
  operations 13ff
  paratheses 13,14
  PI 21
  rules of order 13-15
  scientific 16ff
  subtraction 13,14
Maximum number in a list 217ff
MCS 164ff
MERGE 73,135
  from disc 33
  seq files 135
Messages, ending 112
Minimuum number in a list 218ff
MOD 57
Multi-line FOR-DO's 74
Multiplication 13,14

Name, files 233
Nested loops 142ff
NEW 6
NOT 106

NOT EOD 153ff
Not equal to (<>) 88
NULL 77
Numerics, INPUT 126ff

One-liner FOR-DO 74,76
OPEN, file 233
Operands 4
OR 106,109
OTHERWISE 160ff
Overwriting program 34

PAGE 35
Paint (see Fill)
Parameter passing 243,244
Parentheses 13,14
PASS 32
PI 21
Pixel value graph 253
Postioning
  CURSOR 129,130
  INPUT 130
PRINT 5,50,52
  AT 200
  comma 50
  for spacing 50
  semicolon command 49,50
Printer
  command 36
  copy 40
Procedure
  parameter passing 243,244
Procedures 163ff
  DISPLAY 164
  EDIT 164
  LIST 164
Program 9
  blank lines 40
  overwriting 34
  SAVE 33ff
  scrolling 97

RAM 2
RANDOM
  access memory 2
  coin flipping 149
  file 233,240ff
  file, READ 240
  file, size of entry 240
  file, WRITE 240-242
RANDOMIZE 63
READ 113ff
READ-DATA 113ff
  array 214
READ, files 234
Read only memory 2
READ, random file 240
Record, files 232
  sequential file 232
RENUM 5,24
REPEAT 136ff
REPEAT-UNTIL 135ff
  for files 236
Replace with 7
RESTORE 154
RESTORE key 3
RETURN key 3
Reverse on/off 196
RND 61-63
ROM 2
Rounding real numbers 59-61
Rows, arrays 205ff
Rows, arrays summation 206ff
RUN 7
  f7 67
  turtle with f3
  with f7 80
RUN/STOP key 3

SAVE 33ff
SCAN 106
  with f8 181
Scientific notation 16ff
Screen color 13
Scrolling program lines 97
Selection sort 223ff
Sequential file 232ff
  CLOSE 234
  create 233
  LIST 73
  MERGE 73
  OPEN 233
  READ 234
  Record 232
  WRITE 233
SHIFT key 2
Sort, selection 223ff
Sorting 217ff
Spacing with PRINT 50
SQR 57,58
Square root 57,58
STEP 76,77
STRING 46ff
  arrays 196ff
  DIM 68
  dimensioning 68ff
  INPUT 126ff
  with + sign 68
Structures, procedures 167
Subscripted variables 183ff

subtraction 13,14
Summation
  arrays 186-187,209ff
  loops 83-84,186-187,209ff

Trailing punctuation
  comma 50ff
  semicolon 49ff
Truncation 58,59
Turtle 248ff
  arc 252,253
  angles 249ff
  background color 254
  circle 253
  f1 key
  f5 key
  FILL 254
  FULLSCREEN
  HIDE 250
  move 249ff
  pendown 250
  penup 250
  size 248

Two-dimensional arrays 204ff

UNTIL 136ff

Variable
  CLOSED 167,168
  dynamic 7
  global 168
  static 7
Variables, subscripted 183ff
Variables, use of apostrophe 204
VERIFY 127

WHILE 152ff
WHILE-DO 152ff
WHILE-DO ENDWHILE 153ff
WHILE NOT EOD 153
Williams, Ted 146ff
WRITE, files 233,234
WRITE, RANDOM files 240-242

ZONE 50
ZONE and semicolon 75

Other COMAL books available from COMAL Users Group, U.S.A., Limited include:

TEXT BOOKS

Beginning COMAL by Borge Christensen
An informal text book for the beginner. Upper Elementary or Junior High level. Written by a Danish professor, the founder of COMAL.

Foundations in Computer Studies With COMAL by John Kelly
A text book for the beginner. Junior High or High School level. Written by an Irish professor.

Starting With COMAL by Ingvar Gratte
A text book for the beginner. Junior High or High School level. Written by a Swedish professor.

Structured Programming With COMAL by Roy Atherton
A text book for intermediate programming. High School level. Written by a British professor.

REFERENCE BOOKS

COMAL Handbook by Len Lindsay
A detailed reference book for Commodore 64 and PET COMAL. Junior High or High School level. Written by the Editor of COMAL Today magazine.

Commodore 64 Graphics With COMAL by Len Lindsay
A detailed reference book for the graphics and sprites in Commodore 64 COMAL version 0.14. Junior High or High School level. Written by the Editor of COMAL Today magazine.

TUTORIAL BOOKS

Cartridge Tutorial Binder by Frank Bason & Leo Hojsholt
A good overview of using the COMAL 2.0 Cartridge. Upper Elementary through High School level. Written by an American and a Dane for Commodore Denmark.

COMAL Workbook by Gordon Shigley
Provides a nice introduction to COMAL version 0.14 when used together with the *Tutorial Disk*. Upper Elementary through High School level.

ADVANCED BOOKS

COMAL 2.0 Packages by Jesse Knight
Complete instructions on how to make your own packages for COMAL 2.0. Comes with a disk.

Packages Library volume 1 by David Stidolph
A collection of 17 packages ready to use with the COMAL 2.0 Cartridge. Comes with a disk that includes the source code for most of the packages as well as a smooth scrolling editor.