Up to now we've performed some simple operations by entering a single line of instructions into the computer. Once RETURN was depressed, the operation that we specified was performed immediately. This is called the IMMEDIATE or CALCULATOR mode.

But to accomplish anything significant, we must be able to have the computer operate with more than a single line statement. A number of statements combined together is called a PROGRAM and allows you to use the full power of the Commodore 64.

To see how easy it is to write your first Commodore 64 program, try this:

Clear the screen by holding the SHIFT key, and then depressing the CLR/HOME key.
Type NEW and press RETURN. (This just clears out any numbers that might have been left in the computer from your experimenting.)
Now type the following exactly as shown (Remember to hit RETURN after each line)

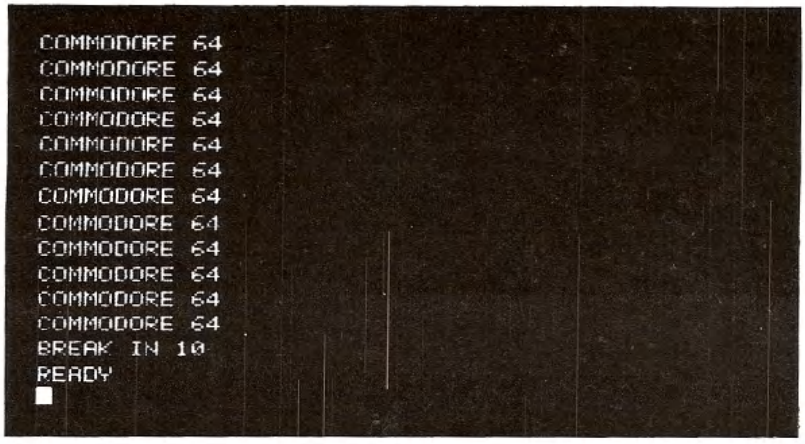
```
10 ?"COMMODORE 64"
20 GOTO 10
■
```

Now, type RUN and hit RETURN—watch what happens. Your screen will come alive with COMMODORE 64. After you've finished watching the display, hit RUN/STOP to stop the program.

A number of important concepts were introduced in this short program that are the basis for all programming.

Notice that here we preceded each statement with a number. This LINE number tells the computer in what order to work with each statement. These numbers are also a reference point, in case the program needs to get back to a particular line. Line numbers can be any whole number (integer) value between 0-63,999.

```
10 PRINT "COMMODORE 64"
↑   ↑
|   └── STATEMENT
└────── LINE NUMBER
```



It is good programming practice to number lines in increments of 10—in case you need to insert some statements later on.

Besides PRINT, our program also used another BASIC command, GOTO. This instructs the computer to go directly to a particular line and perform it, then continue from that point.

```
10 PRINT "COMMODORE 64"
20 GOTO 10
```

In our example, the program prints the message in line 10, goes to the next line (20), which instructs it to go back to line 10 and print the message over again. Then the cycle repeats. Since we didn't give the computer a way out of this loop, the program will cycle endlessly, until we physically stop it with the RUN/STOP key.

Once you've stopped the program, type: LIST. Your program will be displayed, intact, because it's still in the computer's memory. Notice, too, that the computer converted the ? into PRINT for you. The program can now be changed, saved, or run again.

Another important difference between typing something in the immediate mode and writing a program is that once you execute and clear the screen of an immediate statement, it's lost. However, you can always get a program back by just typing LIST.

By the way, when it comes to abbreviations don't forget that the computer may run out of space on a line if you use too many.

EDITING TIPS

If you make a mistake on a line, you have a number of editing options.

1. You can retype a line anytime, and the computer will automatically substitute the new line for the old one.
2. An unwanted line can be erased by simply typing the line number and RETURN .
3. You can also easily edit an existing line, using the cursor keys and editing keys.

Suppose you made a typing mistake in a line of the example. To correct it without retyping the entire line, try this:

Type LIST, then using the SHIFT and ⇑ CRSR ⇓ keys together move the cursor up until it is positioned on the line that needs to be changed.

Now, use the cursor-right key to move the cursor to the character you want to change, typing the change over the old character. Now hit RETURN and the corrected line will replace the old one.

If you need more space on the line, position the cursor where the space is needed and hit SHIFT and INST/DEL at the same time and a space will open up. Now just type in the additional information and hit RETURN . Likewise, you can delete unwanted characters by placing the cursor to the right of the unwanted character and hitting the INST/DEL key.

To verify that changes were entered, type LIST again, and the corrected program will be displayed! And lines don't have to be entered in numerical order. The computer will automatically place them in the proper sequence.

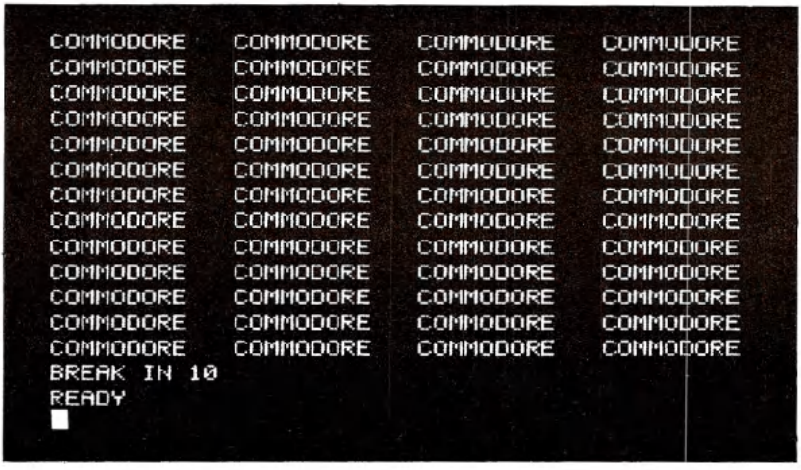
Try editing our sample program on page 33 by changing line 10 and adding a comma to the end of the line. Then RUN the program again.

```
10 PRINT "COMMODORE",
```

DON'T FORGET TO MOVE THE CURSOR PAST LINE 20 BEFORE YOU RUN THE PROGRAM.

VARIABLES

Variables are some of the most used features of any programming language, because variables can represent much more information in the computer. Understanding how variables operate will make computing easier and allow us to accomplish feats that would not be possible otherwise.



Imagine a number of boxes within the computer that can each hold a number or a string of text characters. Each of these boxes is to be labeled with a name that we choose. That name is called a variable and represents the information in the respective box.

For example, if we say:

```
10 X% = 15
20 X = 23.5
30 X$ = "THE SUM OF X%+X = "
```

The computer might represent the variables like this:

```
X%    15

X     23.5

X$    THE SUM OF X%+X =
```

A variable name represents the box, or memory location, where the current value of the variable is stored. As you can see, we can assign either an integer number, floating point number, or a text string to a variable.

The % symbol following a variable name indicates the variable will represent an integer number. The following are valid integer variable names:

A%
X%
A1%
NM%

The '$' following the variable name indicates the variable will represent a text string. The following are examples of string variables:

A$
X$
MIS

Floating point variables follow the same format, with the type indicator:

A1
X
Y
MI

In assigning a name to a variable there are a few things to keep in mind. First, a variable name can have one or two characters. The first character must be an alphabetic character from A to Z; the second character can be either alphabetic or numeric (in the range 0 to 9). A third character can be included to indicate the type of variable (integer or text string), **%** or **$.**

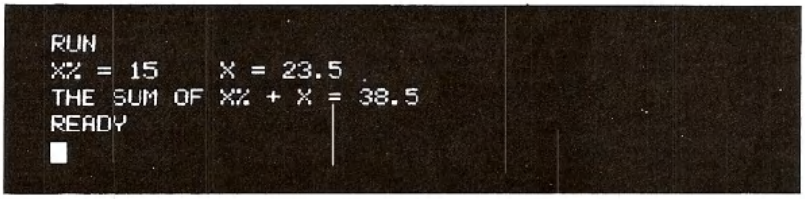
You can use variable names having more than two alphabetic characters, but only the first two are recognized by the computer. So PA and PARTNO are the same and would refer to the same variable box.

The last rule for variable names is simple: they can't contain any BASIC keywords (reserved words) such as GOTO, RUN, etc. Refer back to Appendix D for a complete list of BASIC reserved words.

To see how variables can be put to work, type in the complete program that we introduced earlier and RUN it. Remember to hit RETURN after each line in the program.

```
NEW
10 X% = 15
20 X = 23.5
30 X$ = "THE SUM OF X% + X = "
40 PRINT "X% = "; X%, "X = "; X
50 PRINT X$; X% + X
```

If you did everything as shown, you should get the following result printed on the screen.



We've put together all the tricks learned so far to format the display as you see it and print the sum of the two variables.

In lines 10 and 20 we assigned an integer value to X% and assigned a floating point value to X. This puts the number associated with the variable in its box. In line 30, we assigned a text string to X$. Line 40 combines the two types of PRINT statements to print a message and the actual value of X% and X. Line 50 prints the text string assigned to X$ and the sum of X% and X.

Note that even though X is used as part of each variable name, the identifiers **%** and **$** make **X%, X,** and **X$** unique, thus representing three distinct variables.

But variables are much more powerful. If you change their value, the new value replaces the original value in the same box. This allows you to write a statement like:

```
X = X + 1
```

This would never be accepted in normal algebra, but is one of the most used concepts in programming. It means: take the current value of X, add one to it and place the new sum into the box representing X.

IF . . . THEN

Armed with the ability to easily update the value of variables, we can now try a program such as:

```
NEW
10 CT = 0
20 ?"COMMODORE 64"
30 CT = CT + 1
40 IF CT < 5 THEN 20
50 END
RUN
COMMODORE 64
COMMODORE 64
COMMODORE 64
COMMODORE 64
COMMODORE 64
```

What we've done is introduce two new BASIC commands, and provided some control over our runaway little print program introduced at the start of this chapter.

IF . . . THEN adds some logic to the program. It says IF a condition holds true THEN do something. IF the condition no longer holds true, THEN do the next line in the program.

A number of conditions can be set up in using an IF . . . THEN statement:

SYMBOL	*MEANING*
<	Less Than
>	Greater Than
=	Equal To
< >	Not Equal To
> =	Greater Than or Equal To
< =	Less Than or Equal To

The use of any one of these conditions is simple, yet surprisingly powerful.

```
10 CT = 0
20 ?"COMMODORE 64"
30 CT = CT + 1
40 IF CT < 5 THEN 20
50 END
```

In the sample program, we've set up a "loop" that has some constraints placed on it by saying: IF a value is less than some number THEN do something.

Line 10 sets CT (CounT) equal to 0. Line 20 prints our message. Line 30 adds one to the variable CT. This line counts how many times we do the loop. Each time the loop is executed, CT goes up by one.

Line 40 is our control line. If CT is less than 5, meaning we've executed the loop less than 5 times, the program goes back to line 20 and prints again. When CT becomes equal to 5—indicating 5 COMMODORE 64's were printed—the program goes to line 50, which signals to END the program.

Try the program and see what we mean. By changing the CT limit in line 40 you can have any number of lines printed.

IF . . . THEN has a multitude of other uses, which we'll see in future examples.

FOR . . . NEXT LOOPS

There is a simpler, and preferred way to accomplish what we did in the previous example by using a FOR . . . NEXT loop. Consider the following:

```
NEW

10 FOR CT = 1 TO 5
20 PRINT "COMMODORE 64"
30 NEXT CT

RUN
COMMODORE 64
COMMODORE 64
COMMODORE 64
COMMODORE 64
COMMODORE 64
```

As you can see, the program has become much smaller and more direct.

CT starts at 1 in line 10. Then, line 20 does some printing. In Line 30

CT is incremented by 1. The NEXT statement in line 30 automatically sends the program back to line 10 where the FOR part of the FOR . . . NEXT statement is located. This process will continue until CT reaches the limit you entered.

The variable used in a FOR . . . NEXT loop can be incremented by smaller amounts than 1, if needed.

Try this:

```
NEW

10 FOR NB = 1 TO 10 STEP .5
20 PRINT NB,
30 NEXT NB

RUN
1             1.5           2             2.5
3             3.5           4             4.5
5             5.5           6             6.5
7             7.5           8             8.5
9             9.5           10
```

If you enter and run this program, you'll see the numbers from 1 to 10, by .5, printed across the display.

All we're doing here is printing the values that NB assumes as it goes through the loop.

You can even specify whether the variable is increasing or decreasing. Substitute the following for line 10:

```
10 FOR NB = 10 to 1 STEP -.5
```

and watch the opposite occur, as NB goes from 10 to 1 in descending order.