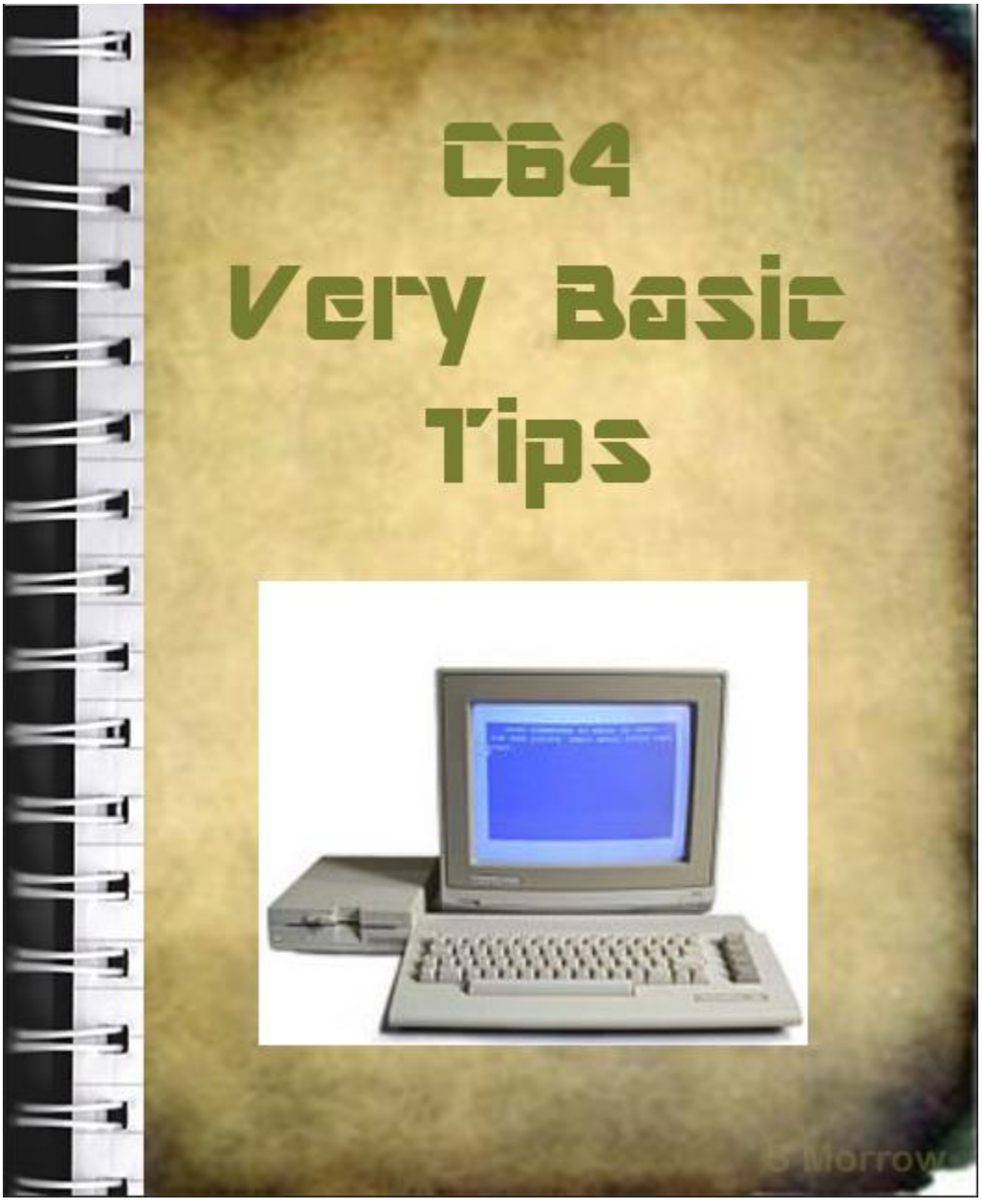
C64 Very Basic Tips



Dimensioning Arrays

It is time to learn about a two-dimensional array. An example is DIM (10,5). This allocates space for 10 rows and 5 numbers in each row. These can be handy for budget sheets, calculation algorithms, and keeping track of game data. Here is an example

DIM X (10,5)

The above statement allows the computer to reserve space in memory for such a two dimensional array under the name of X. In total this allows a preservation of 50 numbers (10 x 5 = 50).

In the example below, the third element in the 10th row is referred to as X (10,3). Rows are down and columns are across.

FOR EXAMPLE (R): FOR EXAMPLE (C)

ROW (R)	1 (R=0)	2 (R=1)	3 (R=2)	4 (R=3)	5 (R=4)
1 (C=1)	0 X(0,0)	1 X(0,1)	2 X(0,2)	3 X(0,3)	4 X(0,4)
2 (C=2)	1 X(1,0)	2 X(1,1)	4 X(1,2)	6 X(1,3)	8 X(1,4)
3 (C=3)	2 VL(2,0)	4	6	8	10
4 (C=4)	3	6	9	12	15

The program below puts the numbers in the above chart into a two dimensional array (X) and then gets an average calculation of each row.

```
10 DIM X(10,5),A(10)
20 FOR R=1 TO 10
30 T=0
40 FOR C=1 TO 5
50 READ X(R,C)
60 T = T + X(R,C)
70 NEXT C
80 A(R) = T/5
90 NEXT R
100 FOR R=1 TO 10
110 PRINT”QQROW #”;R
120 FOR C=1 TO 5
130 PRINTX(R,C):NEXT C
140 PRINT”AVERAGE = “;A(R)
150 FOR D=1 TO 1000:NEXT
160 NEXT R
170 DATA 1,3,5,7,9
```

```
180 DATA 2,4,6,8,10

190 DATA 5,10,15,20,25

200 DATA 10,20,30,40,50

210 DATA 20,40,60,80,100

220 DATA 30,60,90,120,150

230 DATA 40,80,120,160,200

240 DATA 50,100,150,200,250

250 DATA 100,200,300,400,500

260 DATA 500,1000,1500,2000,2500
```

GOSUB-RETURN

There is an instruction that is similar to the GOTO statement, but with an additional benefit called GOSUB. This statement will jump to a particular line depicted by the GOSUB parameter (value), jump to that line and wait for a RETURN statement before returning back to the GOSUB. This is known as a subroutine. An example should clarify this.

```
10 A$=”SUBROUTINE”:B$=”PROGRAM”

20 FOR J=1 TO 5

30 INPUT”ENTER A NUMBER”;X

40 GOSUB 100

50 PRINT B$:PRINT

60 NEXT

70 END

100 PRINTA$:PRINT

110 Z=X^2:PRINT Z

120 RETURN
```

This example will square five numbers and show the result on the screen. Messages are printed after each INPUT statement to indicate that statement is being executed. As mentioned earlier, whenever the computer encounters a RETURN statement it will force direction back to the line after the last GOSUB that was executed within that subroutine. These can be positioned anywhere in your program. If you fail

to include a RETURN and try to execute a GOTO you will receive an error message and the program will fail.

ON GOTO/GOSUB

This statement is interesting since it allows your program to do something called “branching”. What this means is that the computer can reuse a GOSUB statement upon each RETURN that is found. So if you execute a ON GOSUB X1, X2 then it will search the statement for the additional subroutines, execute them sequentially, and force a redirection to the same line everytime a RETURN is evaluated somewhere in your program. Once again a demonstrate should illustrate this in your mind.

```
10 ? “ENTER A NUMBER BETWEEN ONE AND FIVE”

20 INPUT X

30 ON X GOSUB 100,200,300,400,500

40 END

100 ?”YOUR NUMBER WAS 1”:RETURN

200 ?”YOUR NUMBER WAS 2”:RETURN

300 ?”YOUR NUMBER WAS 3”:RETURN

400 ?”YOUR NUMBER WAS 4”:RETURN

500 ?”YOUR NUMBER WAS 5”:RETURN
```

As the computer encounters each ON X GOSUB instruction and executes them from left to right the program is sent to that corresponding line and a series of instructions are carried out (it prints “YOUR NUMBER WAS x, where “x” represents 1-5. So the program will travel through lines 100-500 before the program ends.

RAM/ROM ACCESS-PEEK AND POKE

We are finally getting to the more powerful commands. Your computer’s memory contains a RAM (Random Access Memory) and ROM (Read Only Memory). It also houses over 64,000 memory locations within the unit. Each portion of memory is designated to carry out specific tasks (also known as functions). There are areas in memory that store your program and its variables.

As you learn more about your machine you will discover that your computer can control the clock, sound registers, graphics, I/O, and many other functions hidden deep below the surface.

We will now discuss the statements used to access these functions and memory pockets. The first one will allow you to peer (or peek) inside to see a value found in a specific memory location. If it is easier imagine the boxes again lined up from 0-65535 and each one containing a number within from 0-255. Below is a way to look inside a memory location.

P=PEEK(650):PRINT P

Enter the statement and press the return key. The computer will respond with a 0. This is similar to the variables you learned about earlier accept now instead of assigning a constant value, you are referencing a value contained in your computer's memory location 650. Think of it visually as peeking inside Box 650 and finding a zero there.

In essence this memory location controls if the computer will repeat keys when you hold them down continuously. We will now show how this works by changing the zero in memory location 650 (Box 650).

POKE 650,96

After you press return, the computer assigns the number 96 to box 650 (or memory location 650). This is known as POKEing a memory address into memory. This area of memory allows this since it is contained in RAM. You can write values directly into RAM, but if you turn off your computer they are reset. Notice that if you keep your thumb on the space bar it will not move over consistently. You are forced to release the space bar and press it again. Only then does the cursor slide to the right upon each keystroke.

Enter the command below to restore that memory location back to its default state, which contains the zero that you peeked into earlier.

POKE 650,0

Also recall when we discussed ROM. ROM does not allow a value to be POKEd into that memory location since it is already hard wired into the chip. In order to POKE into ROM, you have to move memory into earlier that are accessible for writing to RAM. An example of this is to move the computer's character set (all the characters you can type on the screen) into RAM so that you can change that data to your own character set. This is known as redefining a character set. It is common to see this in games. An example is changing a character, such as the letter A to resemble a space ship by altering the bits in that data.

ASC AND CHR$

Since we already discussed the character set the next example should be easier to comprehend. Just understand that each character you can type on the screen is linked to data that draws it on the screen and assigned a value. For example, the letter A is assigned to the PETASCII value 65. Recall how we used PRINT CHR$(147) in our program examples. That line is looking into the character set for the value of 147 to perform a specific action. So when you execute a PRINT CHR$(147) it will obviously clear the screen. Let's see an example of how to look inside at the number found in the PETASCII area.

? ASC("Q")

Once the return key is pressed the computer spits on the value of 81. This means that the computer data value for the letter Q is assigned to slot 81, just as the print clear screen special character (we learned about in Chapter 1) is assigned to slot 147. If it helps imagine the boxes again filled with values and a card. That card contains a function that allows a task to be performed.

FUNCTION KEYS

Next we are going to access the character data that controls the function keys on a standard Commodore 64 keyboard. For the beginners, the function keys are the keys titled F1, F2, F3, and F4, usually located on the right hand side. They are much larger than the standard keys used for typing.

Now also when you access a function key from a specific character slot (using PEEK) you will not see a printed value since they do not contain any. However, they are still assigned a container slot and perform a specific function. Here is a list of each of the function keys assigned to their specific slot.

F1 CHR$(133)

F2 CHR$(137)

F3 CHR$(134)

F4 CHR$(138)

F5 CHR$(135)

F6 CHR$(139)

F7 CHR$(136)

F8 CHR$(140)

Now you will notice that the Commodore 64 only shows the function keys F1,F3,F5,F7. In order to access F2,F4, F6, and F8 you will need to press the SHIFT key while pressing that specific function key. The function keys can be programmed for special tasks such as starting a game, pausing a game, performing a specific task (like for a Utility program), turning off a sound tone, and so on. Your imagination and the sky is the limit. Here is an example utilized to accept entry for the F1 key.

```
10 ?"PRESS F1 TO CONTINUE"

20 GET A$:IF A$="" THEN 20

30 IF A$<>CHR$(133) THEN 20

40 ?"YOU HAVE PRESSED F1"
```

The program will first print a message that instructs the user to press the F1 key. In line 20 the keyboard is evaluated to check for an empty keypress and freezes on that line if no key is selected. Then line 30 performs the real magic. Since the string variable of A$ has received a keypress it is then compared to see if it matches the character slot in 133 before it allows line 40 to be executed. However, if A$ fails to receive the correct keypress then the program is sent back to line 20 to have the user try again. Therefore, line 40 will never execute unless the correct key is found as assigned in character slot 133.

CONVERTING STRINGS AND NUMBERS

If you want to convert a string to a number or a number to a string, then there are commands assigned for these tasks. Here are a few examples.

```
10 A$=”64”

20 A=VAL(A$)

30 ? “THE VALUE OF”;A$”IS”;A
```

Let us evaluate the above program. Line 10 appoints the A “string” to a 64. Remember that string data can contain both characters and numbers. Line 20 next designates the variable A to a numeric value that was listed in the string. So this is actually allowed the 64 to be used as numeric data, which will then accept calculations. Line 30 finishes the program by printing a message that shows that numeric value now converted from the A$ (A “string”).

The next example does the reverse and takes a number and transforms it into a string, which cannot be used for calculations.

```
10 A=64

20 A$=STR$(A)

30 ? A “ IS THE VALUE OF “;A$
```

As you execute the program it will produce the same message. However, keep in mind that the number variable data contained in line 10 has now been transformed into a string and it will not allow calculations.

RANDOM NUMBERS

We have finally arrived at the end of this chapter and can have some fun with random numbers. A random number allows the computer to pick a value located within a specific range, usually starting at zero. Keep in mind also that the numbers list as nine digits in decimal form. These values range between 0.000000001 and 0.999999999. Here is such an example.

```
? RND(0)
```

The result will be a decimal value. Be sure to always use the parenthesis and the value contained within or you will receive a syntax error. What the computer is doing below the surface (recall the PEEK/POKE commands) is evaluating a real time clock and printing values based on that synchronization. The next example shows how to calculation a regular number (known as an Integer) without the decimals.

```
? INT(RND(0) * 6) + 1
```

This example will display a value between 1 through 6. The plus one added on the end forces the statement to skip over zero. However, to generate a value between 0 in a specific number, try the next example.

```
? INT(RND(0) * 6)
```

This time the value is selected between 0-6 since it has not added to the total random area.

I hope you have enjoyed these inside tips from Commodore 64 Brain!

If you have then I welcome you to subscribe to my YouTube channel at:

https://www.youtube.com/user/SeetheTruth4Yourself/featured

My Twitter channel is at:

https://twitter.com/C64Brain