# *QBASIC Tutorial Table of Contents*

# Just a quick word (or 120) before we begin.

This tutorial was originally written in 1991. Although many things have changed between then and now, the fundamentals remain the same. It is still a good way to learn BASIC, and is a good launching point to go into Visual Basic.

**What has changed?** In 1991, MS-DOS was the only operating system you needed. 640k of memory was all you needed. A 20 Mb hard drive would never get filled up. The (***very*** slow by today's standards, 1/100th the speed of current computers) 33MHz 80386 Processor was king. And QBASIC was included free with DOS. Windows 2.0 was just coming on the scene (If I remember correctly), but nobody really used it much. DOS was more stable, and faster. (Still is, in my opinion. Especially with my old Assembly Language code.)

BASIC has evolved and changed, too. The language now contains objects (and the methods, events, and properties that go with them), there are over twice as many data types (Boolean, which was needed a long time ago, as well as new numeric types), the LET keyword is now obsolete, the GOTO keyword is now considered obsolete (although these 2 do still work), GOSUB and RETURN are obsolete, line numbers are gone (They were optional, but still supported in QBASIC), no more BSAVE or BLOAD, PEEK, POKE, INP, OUT, KEYn Function key support, KEY ON and KEY OFF, LOCATE, INKEY$, and WAIT are just some of the functions that are no longer part of BASIC (mostly because they are DOS-dependent).

As far as LET being obsolete, you still assign variables the same way, you just don't put the word "LET" in front of it any more. Instead of

```
LET NEWAMOUNT = 75
```

You simply use

```
NEWAMOUNT = 75
```

In 1991, this was written for GW-BASIC, and in 1995 it was rewritten for QBASIC, which is the form you see it in now. Starting around 1996 or so, I switched from QuickBASIC (the compiler version of QBASIC) to VBDOS (Visual Basic for DOS compiler). That was a big jump! It was a wonderful compiler, though, especially for DOS development that had basic Windows-like properties (listboxes, pushbuttons, menus, drop-down lists, etc). About a year later I switched to Visual Basic 3.0 for Windows and never looked back (too much). Now It's VB6, which is an even bigger jump. Classes, Dictionaries, Objects, OOP, dot-notation, auto-completion (Yaay!!), big difference from the old BASICA days of the 1980s!! (Anybody remember the Commodore PET from the late 1970's?)

The sorting chapter uses the Exchange Sort, because it's a little easier to understand, especially for Beginners. The Shell Sort (and the Recursive Sort, a.k.a. the Quick Sort) are much faster, but more complicated. Bubble Sort and Insertion Sort are too slow, so I didn't introduce them.

*Enjoy!!!!*

# Introduction to MS-DOS QBasic Computer Programming Language

So you've finally acquired that IBM PC-compatible computer, and want to know how to program it. Or perhaps you've had it for a while and are getting tired of running everybody else's programs. Or maybe you want it to do something very specific, but nobody seems to have a program that does what you want. That is why you have surfed to these pages. They will teach you, step by step, command by command, a large part of the the QBASIC language. There is one small catch, however, and it is quite minor. I even hesitate to call it a "catch". *You must be willing and eager to learn the language*.

In the QBASIC programming language series, we will start out relatively fast at the beginning, introducing three commands and explaining constants and numeric variables. We start out relatively fast so that you can begin writing small programs right away, instead of having to wait for two or three web pages to learn all the necessary commands. In fact, the first command that is discussed, the `PRINT` command, is not fully explored. This way, you will have the tools necessary to write programs, yet not be overwhelmed with the flexibility of the commands. Because of that, many commands will be "introduced" twice - once early on to teach you the basic command, and then again later on to explore the more advanced options that the command allows. We will also introduce more advanced features of QBASIC near the end of the series, including Boolean operators, trigonometric functions, etc. Although we will touch on it briefly, it is beyond the scope of this series to thoroughly teach concepts of flowcharting, recursive programming, etc.

Before we begin, I would first like to wish you luck with learning this new language (but does luck really come into play?), and to tell you that it is Very Important that you read these chapters while sitting in front of your computer actually doing the exercises. Print these pages out so you can read them while using QBASIC! You won't learn just by reading. Also, don't be afraid to try things on your own. Try changing the numbers around that are presented in the examples, and see how the computer responds! That is the best way to learn!! Experiment! As you go along, try changing around something in the program and then predict what the outcome will be - that is a great way to really test your knowledge of a command. In short, learn by doing! It is the best way to retain knowledge! You only remember about 25% to 30% of what you hear, about 50% of what you read or see, but about 75% to 80% of what you do!

Notice that these pages are called "*Chapters".* Since the chapters are all laid out in a logical manner, each chapter building on the commands and functions learned in previous chapters, it makes a great "quick-reference" guide, but goes one up of most other quick references by including a full explanation of the command and includes examples of how to use the command. It bridges the gap between an overly brief quick reference guide and a very dry, not too understandable and sometimes still sparsely-explained QBASIC reference manual.

Because these chapters do build upon the ones that come before it, you should page through all of the chapters in order.

Just a word about printing conventions used in these chapters. The regular body of the text is in this font, with emphasized words in *italics.* Information that you will type in will be in a **`bold font.`** Program output will be in `this font.` New BASIC key words will be printed in a **bold face serif font**. There is one exception to this general rule. Program listings of more than five lines that you will type in will be printed in the `serif font.`

Again, we wish you well in your venture to become just a bit more computer literate. While this is not overly hard work, it will take a little bit of persistence, or stick-to-it-iveness, if you prefer. Above all, however, have fun!! Don't just try the examples in this series, but see if you can write some interesting programs of your own! Since QBASIC is a strong engineering language (that means it is loaded with mathematical functions), you will be able to solve complex mathematical problems. With the power of looping and the quick speed of the computer, many redundant and recursive math functions will be performed quickly and with very little effort. In fact, one example program in the booklet calculates a square root of a number simply by repeated division that zeroes in on the answer. This is an example of recursive programming. That means the program keeps running through the same set of instructions continuously until a certain condition is met (In this case, the accuracy of the solution). There is no way of knowing how many times the instructions will be cycled through, and in fact will be different for almost every different value that is plugged into the formula!

Now sit down in front of your computer, turn it on, get Chapter One ready, stretch out your fingers over the keyboard, and let's have some fun!

# Chapter One
## Getting Started
KeyWords: END, LET, PRINT

### Starting QBASIC
To fully utilize the QBASIC system with all the options makes it quite a complex command. Here is the command in full:

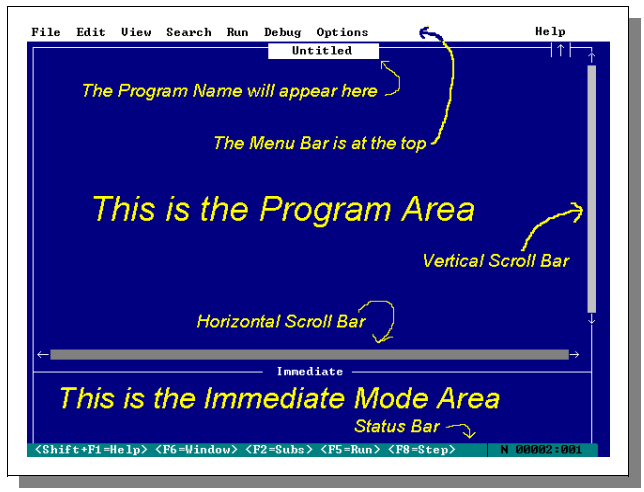
**QBASIC /B /G /H /MBF /NOHI /RUN [filename]**

Quite a command, huh? I know you're thinking "How in the world am I gonna remember all that?" Well, do we have some good news for you! We will now tell you the slightly easier way to start QBASIC. Simply type in:

**QBASIC**

and press Enter. Voila! You are now in QBASIC! See? Wasn't that easy? Oh, by the way, we won't tell you what all that other seemingly unnecessary stuff is for the QBASIC command. We will just say that it is for telling the system how to handle different video configurations. By the way the Q in QBASIC stands for QUICK. QBASIC was derived from Microsoft's highly successful and incredibly powerful QUICKBASIC compiler series. QUICKBASIC has many more capabilities than QBASIC, and allows much larger programs to be written, including easier interfacing with other high-level languages (such as C++) and low-level languages (like MACRO ASSEMBLER). I have even combined all three of those languages into a single program using QUICKBASIC. It can't be done with QBASIC. Oh, well. QBASIC doesn't cost anything, so I really can't complain!

Let's take a look at what is on the screen. At the top, you have a line of words. This is the menu system. We will be using it very shortly. Most of the rest of the screen is blue, except for that white box in the middle that says Welcome to MS-DOS QBasic, and gives the copyright notice. Press the ESC key to clear this box.
You will see that the screen is divided into two sections. The top section has a title above it that says Untitled. This is where the program name will appear when you are working on a program. The bottom window has a title of Immediate. We will be using this section when we write and test programs. The very bottom line shows a handfull of function key shortcuts, and your cursor position.




Enough chatter, let's get this computer to do something interesting!!! Yes, it's time already to learn our first command!!! What is it, you

ask? Well, it' the (drum roll, please....) **PRINT** command!! Can you guess what it does? Well, let's try it and see. First, press the F6 key until the word Immediate (near the bottom of the screen) is highlighted. That will allow us to type in QBASIC commands directly.

Type in the word

```
PRINT
```

(it can be in upper or lower case - the computer doesn't really care) and press the Enter key. Oops! It looks like we are viewing the screen that we had when we entered our QBASIC command! In fact, we can still see the command we entered, with a line at the bottom that says Press any key to continue. Go ahead and press any key, and you will be back in the QBASIC environment. Let's get this thing to do something a little more fun. Now type

```
PRINT 4
```

and press ENTER. Notice that there is now a number 4 at the bottom of the output screen? That's because we used the print command to print the number 4.

Let's recap. You told Mr. Computer to print 4. What did it do? It printed the number 4 on the output screen! Wait, it gets better. Now type this:

```
PRINT 7
```

Press the Enter key. Did you see that? Now it printed the number 7 on the output screen! Isn't this great? Let's see if we can print words, too! Type this:

```
PRINT LUNCH
```

and press ENTER. Wait a minute! It didn't print the word lunch, it printed a 0! This thing must be broken! Why didn't it print out what I told it to do? Hold on - how did we that it printed a 0 instead of the word lunch? Do you think maybe it's supposed to do that? The answer, of course, is yes. In the above example, the word "LUNCH" is what is known as a numeric variable. That simply means that it is a name which can represent just about any numeric value. You could call it a storage place for data. For example, we could assign the word LUNCH to have a value of 10. How do we do that in QBASIC? That is our second command we will learn. It is the LET command, and it looks like this for our particular case (go ahead and type this in ):

```
LET LUNCH = 10
```

and then press the ENTER key. Hmmmm, It didn't show us the output screen this time. That's good! That means that QBASIC understood the command that we typed in, and acted upon it. Now type in:

```
PRINT LUNCH
```

Look at that! It printed out the number 10! Let's try to change the value of lunch. Type:

```
LET LUNCH = 49
```

No response - good. Now type

```
PRINT LUNCH
```

You should get a 49 on the output screen. Neat, huh? We can use the same variable name to represent different things! Of course, when we told LUNCH to store the value of 49, it threw out the previous value of 10. A variable can only hold one thing at a time. If you tell it to hold something new, it has to let go of what it was holding before. It only has one hand, you know!

Now we start to cook. Here we are going to use three numeric variables, and a math function. Type in this series of lines (press Enter after each line):

```
LET FIRST = 9
LET SECOND = 6
LET ANSWER = FIRST + SECOND
PRINT ANSWER
```

Did you get what you expected? If everything went correctly, there should be a 15 on the output screen.

Let's explore what just happened. The first two lines you should be familiar with. The variable FIRST was assigned a value of 9. The variable SECOND was given a value of 6. The variable ANSWER was then given the value of SECOND added to the value of FIRST (6+9 in this case). Then the value of ANSWER was printed. That third line is where the power of computer programming begins to become apparent. You can put this line in a program, and it will add whatever those two variables (FIRST and SECOND) happen to be at the time. Let's look a little closer at the third line. QBASIC operates on the LET command by solving any math problems that is to the right of the equal sign using the standard Algebraic Order of Operations. If pain persists, see your mathematician. It simplifies this expression down to a single value, and (assuming no errors like dividing by zero) assigns it to the variable on the left side of the equal sign. There can only be one variable on the left side of the "equation", and no math operators. It can only be a single variable. However, you can have a highly complex formula on the right side of the equation. As long as it can be reduced (or solved) to a single number, anything is fair game on the right side of the equal sign!

Now let's get right into some real programming. All we have been doing so far is using QBASIC's "immediate mode". What that means is QBASIC performs the command we type in immediately after we press Enter. The other mode of QBASIC is (can you believe it?) the "Program mode". In the program mode, the instructions or commands we type in are not performed as we enter them, but instead are stored in the computer's memory as a program. After we get the program entered into the memory, we can then execute the program at a high rate of speed. How do we get into the program mode? Press the **F6** key until the word **Untitled** (at the top of the screen) is highlighted. You will also see the cursor (that flashing underline thing) jump up to the program window. Now type this in:

```
LET FIRST = 14
LET SECOND = 8
LET THIRD = FIRST + 6
LET ANSWER = FIRST + SECOND + THIRD
PRINT ANSWER
END
```

Let's go through this line by line. Lines 1, 2, and 5, you have seen before. However, the third line looks a little different. It is straightforward, however. It adds 6 to the current value of FIRST (14 in this case) and assigns it to the variable THIRD. The fourth line simply adds the values of all three variables and assigns it to the variable ANSWER. Incidentally, all the variables that are on the right side of the equal sign in a LET command are not altered. In other words, the FIRST, SECOND, and THIRD in line 4 do not change or lose their value when the line is executed. The 6 in the third line is called a constant. That is because its value doesn't change when you run the program. QBASIC allows you to define words as constants so that the language become more symbolic. For example you could just type in PI instead of 3.14159 every time you needed that value. We'll show you how in the Advanced series.

The last line contains a new command, END. It obviously is the end of the program. If there happens to be more program lines after the END command, they are not executed, because END tells the computer to stop. Simple, really. Now we need to tell the computer that we want this program to run. How do we do that? Take a look at the very bottom of the screen. Notice that it tells us that the F5 key is a shortcut key to run the program. Go ahead and press F5 to run it. Oh, look! You get a 42 as a result. Can you figure out why? Right! FIRST is 14, SECOND is 8, and THIRD is 20 (14+6). Add the three up, and you get 42. Now it is your turn to write a few programs. If you want to do more than add, here are a few more operators:

Addition is "+"
Subtraction is "-"
Multiplication is "*"
Division is "/"

To erase the existing program so that you can start with a new one, select the File menu either with the mouse (*by clicking on it*) or the keyboard (*by pressing* `Alt-F`). Then from the pull-down menu, select New either with the mouse (*by clicking on it*) or the keyboard (*by pressing* `N`). You will get a box asking if you want to save the changes. We really don't need to save this, so answer No. You will then be presented with a clean slate.

## Introduced In This Chapter:

*Keywords:* `END, LET, PRINT`
*Concepts:* Constants, Numeric Variables.

# Chapter Two

## Program Looping

### KeyWords: IF...THEN

We certainly covered a lot of ground in that first chapter, didn't we? Well you can relax just a little bit in this chapter. Here we will only introduce two commands, but boy, one of them sure is a doozey! It is a command that can make the computer appear to think. It is the decision making command of QBASIC. We will also let you in on a couple of shortcuts for some commands. But all in due time.

Let's get right into it. Sit in front of your computer, turn it on, and get yourself into QBASIC if you're not already there. We are going to have the computer do some counting for us. Let's enter the program now (in the program {upper} window), and we'll explain it afterwards.

```
LET COUNT = 1
MORE:
PRINT COUNT
LET COUNT = COUNT + 1
IF COUNT < 21 THEN GOTO MORE
END
```

Now press F5. Watch closely, it will go pretty fast. Let's explore the program. You should already understand lines 1 and 3, but can you explain line 4? Try to figure it out for yourself. Line 5 contains our new, powerful command. It is called the IF...THEN command. Here is what it does. The expression between the words IF and THEN is evaluated to find out if it is true or false. In this case it tests to see if the current value of COUNT is less than 21. If it is, the program executes the command after the word THEN. In this case, the program goes to the label MORE. If the expression is false, the program skips down to the next line, ignoring everything after the THEN on that line. In our case, the next line is 5, which has the END command, which stops the program.

Let's talk about labels for a little bit. Line 2 is a label in our little program. A label is a single word that is not a QBASIC command (we call those reserved words because they are reserved for a special purpose) followed immediately by a colon. A label is a way to tell the computer where things are. If you put a bunch of stuff in cardboard boxes and would want to be able to find some of that stuff later (without having to open all the boxes to find it), you yould put a label on that box. We are doing something similar here. We use a label to tell the computer where we will want to find something later on. In line 5, we look for that label, so that the computer program can jump to the next line (line 3) in the program and continue on from there.

Would you like to know a couple of secrets? No, they're not really secrets. They are shortcuts for two of the commands we've already learned. They make typing programs in a little easier. The first one is for the LET command. The word LET is actually optional! In other words the line

```
LET VALUE = 17
```

is identical to the line

```
VALUE = 17.
```

The other shortcut is for the PRINT command. Since it is used so often, the writers of QBASIC decided to use a shortcut for typing it in. It is the question mark (?). It is conveniently located next to the right shift key, which makes printing things quite fast and easy. Using these two shortcuts, the above program would have been typed in like such:

```
COUNT = 1
MORE:
? COUNT
COUNT = COUNT + 1
IF COUNT < 21 THEN GOTO MORE
END
```

Notice that when you press the ENTER key after typing line 3 that the ? is automatically converted to PRINT, but the words LET will not be added. They are not considered necessary. Just a couple of ways to make typing programs easier. We will continue to use the word LET in all our future program listings, as well as the full word PRINT. You can leave out the LET and use the question mark character to save some typing, though.

Have you noticed that when you print something that it only prints one thing on a line? Seems a waste of space, doesn't it? What if you wanted two pieces of data on one line? Here is our first advanced function of the PRINT command. It's called the comma, and it works like this. Press F6 to switch to the Immediate window. Type this in:

```
PRINT 2,4,6
```

(Don't forget to press ENTER - We won't be telling you to do that anymore) and notice what happens. You get those three numbers back on the same line, separated by about 7 spaces. What the comma does is perform like a Tab key on a typewriter. The only thing about the computer is the tabs are automatically set for every fifteen character positions on most computers, and they cannot be altered; you're stuck with them. It comes in handy for doing columns, though. Let's write another program now. This one will display a list of numbers from 1 to 15, and show each number multiplied by 2, and multiplied by itself (that would be its square). Press F6 to switch back to the program window, erase any existing program by selecting New from the File menu (if you need to), and type this program in:

```
LET NUMBER = 1
AGAIN:
LET DOUBLE = NUMBER * 2
LET SQUARE = NUMBER * NUMBER
PRINT NUMBER, DOUBLE, SQUARE
LET NUMBER = NUMBER + 1
IF NUMBER < 16 THEN GOTO AGAIN
END
```

Now Press F5 and watch the program fly! Incidently, the PRINT command can end with a comma, such as PRINT RESULT, and the next time you run across a PRINT command in the program, it will pick up in the next column on the same line. In other words, it defeats the carriage return function of the PRINT command. Flexible, huh? Just wait until later, when we tell you even more about the PRINT command!

Do you want to spice up that last program with column headings? Sure you do. Why? Because this is where you are going to learn how to print words out! Add this line to the very top of the program (Put the cursor at the top of the screen by pressing CTRL-HOME):

```
PRINT "#" , "#*2" , "#*#"
```

and then press ENTER to move the rest of that line down. Make extra sure that you have those quotation marks in the right place! Now press F5 and notice the program output. See how everything that is inside a pair of quotation marks in a PRINT command gets displayed on the screen exactly the way it is typed into the program.

Now that we have a decent running program, you may want to save it for posterity. Select "Save As..." from the File menu. You will be presented with a box that asks for a filename. Type in any legal DOS filename that you want. You are limited to 8 characters, with no spaces. You may use letters, digits, dashes, underlines, and a handful of other symbols. Don't put a filename extension (like .BAS) because QBASIC will add it automatically.

With the cursor in the File Name box, type in

```
MULT
```

and press ENTER. If everything went OK, you will receive no error messages, and the word "Untitled" at the top of the program window will change to "MULT.BAS", indicating the name of the program.

We need to talk about typing errors. Erase any existing program, and type this in:

```
PRIUNT ANSWER
```

Obviously, the word PRINT is spelled wrong. Something about large fingers, I think. You may notice that if you typed the program in lower case that QBASIC did not automatically convert the word into capital letters. That's because QBASIC doesn't know what the word PRIUNT means; It thinks you are going to use it as a variable. However, when you press F5 to run the program, the computer scans the program for errors, highlights the unknown word, and presents a white box that says "Syntax Error". Syntax is a tax that you pay on syn. Just kidding. That means that it doesn't quite understand what you are trying to do. To correct this

error, click on OK in the white box, move the cursor under the U, and press the Delete key once to remove the offending letter. QBASIC behaves just like a text editor! DO NOT PRESS ENTER AFTER DELETING THE "U", OR YOU WILL SPLIT UP THE LINE! Just press the Down Arrow once, and you will see the word "print" converted to upper case. Go ahead and press F5 again, and the program will run correctly. The answer will be zero, of course.

Let's try another IF...THEN program. Clear out any existing program and enter:

```
LET A = 1
MORE:
LET B = A*A
PRINT A, B
LET A = A+0.1
IF B<2.4 THEN GOTO MORE
END
```

See if you can predict what will happen here. You don't have to know on just what number the program will end (unless you can figure square roots of fractional numbers in your head!), but see if you can figure out what value that the variable B will not go over. Then RUN the program to see if you are correct. I'll bet you're wrong! I was! It's because the PRINT comes before the IF...THEN.

Here are all of the relational operators that can be used with the IF...THEN statement:

"=" is equals; the expressions on both sides of the equal sign must be numerically equivalent
"<" is less than; just like in grade school math class.
">" is greater than; grade school again.
"=<" or "<=" is less than or equal
"=>" or ">=" is greater than or equal
"<>" or "><" is not equal; if the two sides are different, the THEN GOTO label will be jumped to.

Try each relational operator in an IF...THEN statement and see what happens. Now you can write lots of programs! You've learned how to put output into columns using the comma in the PRINT command. By the way, if you want to move over two columns, just put two commas together like this:

```
PRINT FIRST,,SECOND
```

and that will do the trick! You also know how to print words or letters by using quotation marks in the PRINT command. You know how to create a program loop using the IF...THEN command. A loop is a section of program that gets executed over and over again (usually changing the variables each time through) until a certain condition is met (that is what the IF... THEN tests for). By the way, if something goes wacky, and your loop keeps going and going and going and going and going and going, you can stop the program by holding down the Ctrl key and pressing the Break Key. The word Break is usually printed on the front of a keycap on the keyboard. It is usually either on the Scroll Lock (sometimes abbreviated ScrlLk) key or the Pause key on keyboards that have it. Both keys are toward the right side of the keyboard. You may release the Ctrl key after releasing the Break key. QBASIC will stop running the program, and return you to the program window. The next line that would have been executed will be displayed in bright white. To continue the program from where you left off, just press F5 again.

To start over from the top, press SHIFT-F5. You may make changes to the program after stopping it with the BREAK key. Some changes will require you to start the program from the top again. If so, QBASIC will warn you about that.

Next chapter, we'll show you how to load a QBASIC program that you saved earlier, as well as another way to set up a loop without using an IF...THEN command. But in the meantime, have fun using and learning about what can be done with that pesky little IF...THEN command!

## Introduced In This Chapter:

*Keywords*: IF...THEN

*Concepts*: Program Labels, Typing Shortcuts, Basic Output Formatting, Printing String Constants, Program Line Editing, Relational Operators, Stopping a Program Manually, Continuing and Restarting a Stopped Program.

# Chapter Three

## Program Looping, Continued

### KeyWords: FOR...NEXT

Let's kick off this chapter by learning how to load a program that was saved to disk at an earlier time. This is where the wonderful menu system comes in. See that line at the top of the screen that has the words "File Edit View Search" and so on? Those are all menus. Since we are dealing with something that is on the disk (that's where we store our program as a file - get it?), open the File menu by either clicking on the word "File" with the mouse, or holding down the "Alt" key while pressing the "F" key (This will be referred to as "Alt-F" in the future). Notice that a menu pops down, with the words "New, Open", etc. Click on "Open". A big box will pop up in the middle of the screen that has a handful of other smaller boxes inside it, labeled "File Name", "Files", and "Dirs/Drives". If you are in the same directory that you were in when you went through Chapter 2, you should see a file called MULT.BAS in the Files box. To load it, just double-click on that file name with the mouse, or use the Tab key to move the highlight bar to the "Files" window, and then the arrow keys to move the highlight bar to the desired file (if there's more than one - If it's the only file, you will have to press the spacebar to get the highlight to show up). Press Enter to load the file, and it will appear in the upper window. Neat, huh?

Notice that you have the program that you wrote in Chapter Two. You may run the program if you wish. Incidently, anytime you load a program into memory, any existing program is automatically erased! Be sure that the current program is saved (if you want to keep it) before loading in a new one!!! If you try to load a program in without saving the current one, QBASIC will tell you about the error of your ways.

As promised last chapter, we are now going to show you a new way to set up a program loop. This new method is much easier, albeit at a very slight expense of flexibility. For the most part, however, this new method will work just about anywhere that the one you are using now works. For those rare cases where this new form wouldn't work, there is even another type of looping statement that usually will. That will come in the Advanced section, though. Enough chatter, let's get to a program. Erase any existing program in memory by selecting New from the File menu, and enter the following program:

```
FOR NUMBER = 1 TO 30
     PRINT NUMBER
NEXT NUMBER
END
```

It seems that we only understand two of the lines here. Are we learning two new commands? Actually, no. It is one statement that happens to take up two lines. The first part of the statement is in the first line, and has the basic form: FOR variable = initial value TO final value. Let's explain that in regular English, using the example we have here. What happens is that the variable NUMBER is given a value of 1, as shown in line 1. It's almost like a LET command saying LET NUMBER = 1. Now the program drops down to line 2 (the indented line), and the

variable NUMBER is printed. The program then goes on to line 3 and sees the command NEXT NUMBER. What does that mean? Well, it adds 1 to the variable NUMBER, and checks back with the FOR statement to see if it has exceeded (in our case) 30. That 30 in the first line is called the final value. Anyway, if the variable NUMBER has not exceeded that final value, the program jumps to the command right after the FOR statement, in our case, line 2. The new value of NUMBER is again printed (it is 2 this time around). Then we get to NEXT NUMBER again. Once more, the value of NUMBER goes up by 1, and is checked with the final value. Now let's assume that we have gone through it 29 times already, and are at line 2. The value of NUMBER is printed (29), and we get to NEXT NUMBER again. NUMBER is incremented by one, so it is now 30. The program checks this value against the final value, which is also 30. In the rules of mathematics, 30 is not greater than 30 (they are equal), so the loop is executed once again, with NUMBER having a value of 30. That value gets printed out in line 2, and then we get to NEXT NUMBER again. Number goes up by one again, and is checked against the final value. Now that NUMBER is 31, QBASIC knows not to go back through the loop again.

Let's try some interesting things with the FOR...NEXT statement. Erase the current program (by selecting New from the File menu), and type this program in:

```
PRINT "Number","Square"
FOR NUM = 1 TO 50
     PRINT NUM, NUM * NUM
NEXT NUM
END
```

Now go ahead and run the program. Zips by quickly, doesn't it? Let me throw a monkey wrench into the works here. What happens if you wanted to count backwards from 50 to 1? Well, it sounds like all we have to do is change the second line. Well, that's half right. So far our new line 2 would look like this (don't type it in; it's wrong!):

```
FOR NUM = 50 TO 1
```

You already know that is wrong, but why? Let's trace through the program. On the first time around, NUM will have the value of 50. Ok, so it prints 50 and 2500 on the first pass of line 30. Now we get to line 40, and NUM goes up by one, now with a value of 51. Let's see, 51 is greater than 1, so we are finished with the loop. Execute the next line, which says END. Program finished. Hold on, here! It only went through it once! How do we get it to decrement the variable instead of incrementing the variable? Well, that is when we use an optional part of the FOR...NEXT statement. It is called the STEP size. Let's type in the correct line 2 first Replace the second line with:

```
FOR NUM = 50 TO 1 STEP -1
```

Normally when you get to the NEXT statement, the variable is incremented by 1. By using the STEP statement, we can set that increment to anything we want. In this case we tell the computer to add minus one to the variable at the NEXT statement, which causes it to go down by one instead of up by one. When a negative number is used for the STEP (as it is here), the rule for the final value changes just a bit. Instead of checking to see if the current value is greater than the final value, it checks to see if the current value is less than the final value. All else is unchanged. Go ahead and run the program. Notice that it counted down from 50 the

way we wanted it to. Let's try something else.

Let's have the computer print out all the even numbers from one to 100. Well, we know that 1 is not an even number, but that 2 is. So we know that the loop will start with the value of two. We also know that every other number is even. Can you guess the STEP size? Don't look at the answer below until you guess!

Here's the new program:

```
FOR NUM = 2 TO 100 STEP 2
     PRINT NUM,
NEXT NUM
END
```

Did you notice our use if the comma in line 2? That way, you can see the entire output of the program on one screen. Notice that it printed out all of the even numbers from 2 to 100. What do you think would happen if you changed the final value in the first line to 99? What about a final value of 101? Type them in and see for yourself!.

Why do we have all of the stuff between the FOR and the NEXT lines indented? That makes it easier for us to see exactly what part of our program will be repeated many times! It doesn't mean much now, but you will find out that it is possible to put a FOR...NEXT loop inside of a bigger FOR...NEXT loop, which can be inside of an even bigger FOR...NEXT loop! You can see how this can get to be a bit confusing; the indentation just makes it a little bit easier to see just where you are.

That's all we'll do for this chapter. We thought you deserved an easy one for now, the last couple were kind of rough. Keep practicing on your PRINT command, your IF...THEN command and now your FOR...NEXT loops. Here are a few programs for you to try:

Print all the odd numbers from 1 to 250;
Compute the factorial of 9 using a FOR...NEXT loop;
Add the numbers from 1 to 750 using a FOR...NEXT loop.

We'll give you the answers to the last two problems in the next chapter. Meanwhile, you are wondering what a factorial is. Nine Factorial is 9 times 8 times 7 times 6 times 5 times 4 times 3 times 2 times 1. If you write that program correctly, you can find the factorial of any positive number up to 33 (the answer of which is in the numerical capability of the computer; 34 and over will give you an overflow error) by simply changing one number in the FOR...NEXT statement. The correct result for the second problem is 362880, and for the third problem is 281625.

Next chapter, we will show you how to place and use data in a program, and if you're nice, how to have the program ask you to type in data from the keyboard while it's running!! See you next time!!

## Introduced In This Chapter:

*Keywords*: FOR...NEXT
*Concepts*: Retrieving a file from floppy or hard drive, a more efficient way of setting up a loop.

# Chapter Four

## Loading and Processing Data

### KeyWords: READ...DATA, GOTO

As promised at the end of Chapter Three, here are the answers to the two problems you were supposed to try. You did try them, didn't you? Here is the program to compute the factorial of 9 (The names for your numeric variables will probably be different):

```
LET PRODUCT = 1
FOR NUMBER = 9 TO 1 STEP -1
     LET PRODUCT = PRODUCT * NUMBER
NEXT NUMBER
PRINT "9 Factorial is ", PRODUCT
END
```

We will learn a slightly better way to print the output later in this chapter. All we do is get rid of the comma in line 5. Here is the program listing for the last problem, adding the numbers from 1 to 750 (1+2+3+4+5+6+...+747+748+749+750=???):

```
LET SUM = 0
FOR NUMBER = 1 TO 750
     LET SUM = SUM + NUMBER
NEXT NUMBER
PRINT "Sum Is", SUM
END
```

You may have noticed that the last program takes about 1/4 second to run. It may seem like it's pretty slow, but then ask yourself: can you count from 1 to 750 and keep a running total in your head in a quarter of a second? Probably not! In fact, go ahead and type in that last program listing above. We'll be using it to learn about that PRINT trick. I'll wait here while you type.

Oh, good! You're back! Go ahead and run the program. Now change line 5 To:

```
PRINT "Sum Is" SUM
```

Now run the program again. Notice how the answer is moved much closer to the words? Makes it easier to read! Incidently, QBASIC is assuming that there is a semicolon (;) between the string constant (the message in between the quotes in the PRINT command) and the variable SUM. In fact, you may have noticed that QBASIC automatically inserted a semicolon. If you want to print the values of two variables next to each other, you would have to use the semicolon. It would look something like this:

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

If you type that in the immediate window, you would get three zeros on the same line, with two spaces between adjacent zeros. One space comes before each digit, and is there to hold the place of the sign. If the number was negative, there would be a minus sign in that space. There is another space after the number just to separate it from other numbers that might be printed in the same PRINT command. It just makes the output look better by not running numbers together.

Now that we have that out of the way, let's process some data in our programs. Clear out the current program, and type this in:

```
READ L,W
LET P = 2 * L + 2 * W
PRINT "Perimeter =" P
DATA 6.5, 2.3
END
```

Go ahead and run the program if you wish. Can you figure out what lines 1 and 4 are doing? For those of you who are totally baffled, it goes something like this: First off, the program sees line 1: READ L,W. When the computer sees a READ command, it looks for the first DATA statement in the program. Ah-HA! There it is, down in line 4! Let's see. It wants us to read L and W. Oh, look! There are two numbers in that DATA statement (They are separated by a comma)! Ain't that handy? So what happens? Well the first piece of data in the DATA statement (6.5 in our case) is assigned to the variable L, and the second piece of data (2.3) is assigned to the variable W. Incidentally, it doesn't matter where the DATA statement appears in the program, as long as it's before the END command. In other words, all of these three program listings operate identically:

```
DATA 6.5, 2.3
READ L,W
LET P = 2 * L + 2 * W
PRINT "Perimeter =" P
END
```

```
READ L,W
DATA 6.5, 2.3
LET P = 2 * L + 2 * W
PRINT "Perimeter =" P
END
```

```
READ L,W
LET P = 2 * L + 2 * W
DATA 6.5, 2.3
PRINT "Perimeter =" P
END
```

The above program processes just one set of data. In our case, it computes the perimeter of only one rectangle. What if we have three rectangles we want perimeters for? Here is the program:

```
TOP:
READ L,W
LET P=2*L + 2*W
PRINT "Perimiter =" P
GOTO TOP
DATA 6.5, 2.3, 7.86, 6.03, 21, 17
END
```

Oh, Look! A new command: GOTO. It does just what it says. When the program reaches line 5, it is told to go to the label TOP. Now it sees that READ command again. What does it do? Well, it remembers where it left off in the DATA statement! Since it read in the first two numbers the first time around, it is going to assign the third piece of data (7.86) to the first variable in the READ command (L) and the next data (6.03) into W. Every time it is asked to READ another value, it takes the next one from the DATA statement. It doesn't even have to be the same READ command; it is for any READ. What happens when it runs out of numbers in the DATA statement? Well, it's not pretty. Run the program and find out. See how you get an Out Of DATA error? Then the program just stops. Hey, it's not great, but it works. We're just learning here, we're not trying to be experts right off the bat! To make the error message go away, click on the OK. If you want to see what the program displayed, press the F4 key to display the output screen. Press any key to return to QBASIC. By the way, a good way to eliminate the "Out Of DATA" error is to have your first DATA element indicate how many times the loop will run (if you know that). Then you could start off your program with something like:

```
READ CYCLES
FOR LOOP = 1 TO CYCLES
...
```

Another good way is to have two more data items of the number "-1" at the end of the DATA statement. Then after you READ the variables, immediately check to see if they are equal to -1 with an IF...THEN command. If it's true, place the label name before the END command after the THEN. Here's how it would look:

```
TOP:
READ L, W
IF L = -1 THEN GOTO NOMORE
LET P = 2 * L + 2 * W
PRINT "Perimeter ="P
GOTO TOP
DATA 6.5, 2.3, 7.86, 6.03, 21, 17, -1, -1
NOMORE:
END
```

This is the preferred method by many programmers, as they don't have to determine how many times the program will run through the loop. Also notice that there are two -1's at the end of the DATA statement. Why? Because the READ command has to fill 2 numeric variables in this example. If we only have one -1 in the line, it will be assigned to the variable L, and there won't be anything for the variable W to be assigned, so you'll still get the Out Of DATA error message. Got it? Good!.

Let's learn something new about our friendly neighborhood PRINT command! All along, we've been using a LET command to do all of our calculating for us. Did you know that you can

calculate in a PRINT command? Try it now: Type this in at the Immediate window:

```
PRINT 72/8
```

and press Enter. See? It gives you a 9 as a result. That's because 72 divided by 8 is 9. PRINT will automatically calculate anything you tell it to before it prints out. The above program would look like this (You may enter and run it if you wish, but don't forget to erase the current program from memory!):

```
MORE:
READ L, W
IF L = -1 THEN GOTO QUIT
PRINT "Perimeter ="; 2 * L + 2 * W
GOTO MORE
DATA 6.5, 2.3, 7.86, 6.03, 21, 17, -1, -1
QUIT:
END
```

That got rid of our LET command, didn't it? We'll let you off easy on this chapter again, but here are some programs that you have to (try to) write before we give you the answers in the next chapter. Also next chapter, we'll show you how to have the program ask the user for data from the keyboard. We will also talk about and use "Nested Loops". Meanwhile, here are some programs for you to try to write.

READ in a series of 20 positive numbers, and print out the largest number. Make sure the program works no matter where in the DATA statement the largest number appears, and use the "-1" method to know when to stop reading data.

Find the average of two numbers. Use at least 3 pairs of data. Do not bother to test for end of data; let the computer generate an error message.

READ a positive integer N and print the product of the four consecutive integers N, N+1, N+2, and N+3. If the program runs correctly, the answer plus one will be a perfect square each time. Use at least five integers in the DATA statement, and do not bother testing for end of data.

That's all for now, See you next time!!!

## Introduced In This Chapter:

*Keywords:* READ...DATA, GOTO

*Concepts:* Semicolons in the PRINT command, calculating in the PRINT command, positioning of the DATA statement in a program, how data is read from a DATA statement.

# Chapter Five

## User Data, Nested Loops

### KeyWord: INPUT

Let's begin this chapter by divulging the answers to the three problems at the end of the last chapter. Please remember that your variable names and the values in your DATA statements will be different. What is important is that the program gives you the correct answers. First was the "Largest Number" program. Here is it's listing:

```
MORE:
READ X
IF X = -1 THEN GOTO FINISH
IF X > LARGEST THEN GOTO UPDATE
GOTO MORE
UPDATE:
LARGEST = X
GOTO MORE
DATA 6,2,4,8,5,12,45,56,18,54,28,64,53,95,75,51,20,15,30,88,-1
FINISH:
PRINT "Largest is"; LARGEST
END
```

Notice how we used the "-1" method to test for the end of actual data.

Next was the "find the average of two numbers" program. It could look something like this:

```
MORE:
READ A, B
LET SUM = A + B
PRINT SUM / 2
GOTO MORE
DATA 1,3,2,4,57,122,6,-9
END
```

Here is probably how you did the "Consecutive Products" problem:

```
MORE:
READ N
O = N + 1
P = N + 2
Q = N + 3
PRINT N * O * P * Q
GOTO MORE
DATA 9,2,45,54
END
```

On to bigger and better things. Well, on to other things, anyway. The first thing we will learn in

this chapter is how to ask the person running the program for information. You probably noticed that up to now, every time time you run a program, you get the same answer(s). If you are using READ...DATA statements, the only way an operator could change the results is to actually go into the program and change the necessary data. The only problem with that is the program user is not usually the person who wrote the program, so they will have no idea what to change, assuming that the person even knows how to program! What we need is a way for the computer to ask the user for data directly. Would you believe that QBASIC has a function to do just that? It's called the INPUT statement. It would look something like this in a program: INPUT X. When the program gets to the INPUT statement, it types a question mark on the screen and waits for the user to type in the requested number. Incidentally, you can also have the statement accept more than one variable at a time, just like the READ statement. It would look like this: INPUT A,B,C. When the program gets to this line, the user must enter the three pieces of data, separated by commas. If only one or two are entered, the computer responds with "Redo from Start" and redisplays the question mark. This is also true if you enter too many pieces of data, or enter something other than numbers. Enough explaining, let's try it in an actual program, like our perimeter thing. Type this program in:

```
PRINT "This Program computes the perimeter of a rectangle."
PRINT "Please enter the length and width of the rectangle,"
PRINT "Separated by commas."
INPUT L, W
PRINT "The Perimeter is"; 2 * L + 2 * W
END
```

Go ahead and run the program. The computer will respond:

```
This Program computes the perimeter of a rectangle.
Please enter the length and width of the rectangle,
Separated by commas.
?_
```

and you will see the blinking cursor. Remember that the cursor means the computer is waiting for you to type something in. Go ahead and enter:

**7,6**

and press the Enter key. The computer will respond:

```
The Perimeter is 26
```

and you will get the "`Press any key to continue`" prompt, meaning that the program is done running. If you were to add program lines (The lines in green)

```
PRINT "This Program computes the perimeter of a rectangle."
PRINT "Please enter the length and width of the rectangle,"
PRINT "Separated by commas."
AGAIN:
INPUT L, W
PRINT "The Perimeter is"; 2 * L + 2 * W
GOTO AGAIN
END
```

then the program would type another question mark on the screen and ask for more data. This would make the program an "infinite loop", and would run forever. To stop an infinite loop, or a running program at any point. hold down one of the keys marked "Ctrl" (some keyboards have one; others have two) and while holding that key, press the "Break" key. The word "Break" is printed on the front of a keycap on some computers, and the top on others. Depending on which keyboard you have, it is either on a key labeled "ScrlLk" over on the numeric keypad section, or on a key labeled "Pause" on the right end of the very top row of keys. If you're not sure, check with the manuals that came with your computer.

The user must be somewhat careful what he types in response to the question mark for the INPUT statement. First, the number of entries must match the number of variables in the statement, or he will get the "Redo from Start" message. He will also get this message if he types in any letters (with 2 very specific exceptions - that's in the advanced section) or symbols. For example, fractions cannot be entered. Three-fourths must be entered as a decimal, not a fraction. In other words, the user must type in 0.75 and not 3/4 to enter the value. Incidentally, the zero before the decimal point is optional. The computer will take .75 without complaining, also.

Time to talk about Nested Loops. No, they're not for the birds! Simply put, it is just a loop that is inside another loop. Here is a simple example (if you want, you can type it in):

```
FOR OUTER = 1 TO 100
    FOR INNER = 1 TO 100
        PRINT "*";
    NEXT INNER
    PRINT "!";
NEXT OUTER
END
```

What happens? well, the OUTER loop is initialized in the first line. Then the INNER loop gets initialized in the second line. Line 3 prints out an asterisk, and line feed is suppressed (notice the semicolon). Line 4 increments the INNER variable and re-executes the inner loop. This happens 100 times, and then the INNER loop is done. We jump down to line 5, where a single exclamation point is printed. Then the OUTER variable is incremented, and tested back at line 2. That loop hasn't run out yet, so line 3 gets executed. The INNER loop is re-initialized all over again, and the whole process starts again. In this particular case, the OUTER loop goes through 100 "cycles", which means the INNER loop is executed 100 times. Each inner loop runs through 100 of its own "cycles", so some simple multiplication tells us (are you following all of this so far?) that the asterisk is printed a total of 10,000 times! It is printed in 100 sets of 100. In other words, the program prints out 100 "*"s followed by a single "!". It does this 100 times, for a total of 10,000 "*"s and 100 "!"s. Got it? I didn't think so. We'll do a couple of much smaller loops so you can see what is happening. Type this in:

```
FOR OUTER = 1 TO 4
    FOR INNER = 1 TO 4
        PRINT "Outer="; OUTER, "Inner ="; INNER
    NEXT INNER
NEXT OUTER
END
```

Now go ahead and run the program. Notice that you get 16 lines? Notice that the INNER loop goes through a complete cycle before the OUTER loop goes to the next value. Each time the OUTER gets another value, the INNER goes through another cycle. It's one dirty way to do multiplication by addition, but has much more interesting uses. One that we will do quite a bit later is for sorting.

Let's move on to some more mathematical functions in QBASIC. You (should) already know four: Addition (+); Subtraction (-); Multiplication (*); Division (/). Just to show you how these work if you haven't experimented with them yet (come on, pay attention - they have all been used in programs so far), type in these lines in the immediate window:

```
PRINT 27 + 55
PRINT 87 - 39
PRINT 12 * 11
PRINT 1215 / 45
```

The answers for each are 82, 48, 132, and 27. Let's learn some new functions. We'll need some of these for the next chapter. Here are all of the "non-trigonometric" mathematical functions:

Exponentiation is `^`
Absolute Value is `ABS(x)`
Natural Exponent is `EXP(x)`
Integer Truncation (what?) is `FIX(x)`
"Largest" Integer Truncation (huh?) is `INT(x)`
Natural Logarithm is `LOG(x)`
Random Number is `RND(x)`
Sign Determination is `SGN(x)`
Square Root is `SQR(x)`

Here's How to use them: To find a "generic" exponent, or in U.S. English, if you wanted to calculate seven to the third power, it would be something like 7^3. Try it now (in the immediate window):

```
PRINT 7^3
```

and see what the answer is. Absolute Value is pretty straightforward. It the number is positive, it stays positive. If it's negative, it is made positive. Try these:

```
PRINT ABS(16.2)
PRINT ABS(-27.3)
```

Notice that both answers are positive. Next up is natural exponentiation. That is simply the number *e* raised to the indicated power. *e* is the symbol used for the natural base. It's value is approximately 2.71828 or so. QBASIC operates using natural logarithms and antilogarithms. For example, to print *e* raised to the fourth power, type **`PRINT EXP(4)`** and voila! There's your answer!

Integer truncation just chops off anything after the decimal point. Type in

```
PRINT FIX(4.728)
```

and see what you get. Just the 4, of course. Now let's try the Integer function. Type in

```
PRINT INT(4.728)
```

and what do you have? 4 again. What's the difference, you ask? Type these in and notice the answers:

```
PRINT FIX(-4.728)
PRINT INT(-4.728)
```

AH-HA!! There's the difference! The FIX function just chops off the decimal point and everything after it. The INT function returns the next lower (or more negative) integer. It only makes a difference for negative numbers.

Natural logarithm is the inverse of the natural exponentiation discussed above.

We'll devote a major chunk of a future chapter to random numbers and their uses.

SGN(x) can be kind of nifty. Here's all it does (Type these in):

```
PRINT SGN(14)
PRINT SGN(0)
PRINT SGN(-244)
```

Notice that positive numbers give you an answer of 1, negative numbers give an answer of -1, and zero gives you back a zero. This can be useful for certain types of decision-making with IF...THEN statements. Can you think of any? I didn't think so

The last one is Square Root. Use it like this:

```
PRINT SQR(625)
```

and you get the correct answer of 25.

No "quizzes" this time. Instead, practice using some of these new math functions in programs of your own design. Try to think up of some kind of nifty math problem (borrow somebody's math book and have fun!) and try writing a program to compute the answer. Be sure to use INPUT statements for getting the values into the computer!! Next chapter, we will write a "Square Root" program that uses a "Divide and Average" method for finding the answer. If you know how this method works, see if you can write the program ahead of time. Be careful, though - even when written correctly, the computer will either give a wrong answer or "hang up"! We'll find ways to fix that by introducing "tolerance acceptance" as well as guarding against bad inputs (like negative numbers) using something called "input checking" or "error checking" or something like that. See you next chapter!

## Introduced In This Chapter:

*Keywords*: INPUT

*Concepts*: Basic User Interface, Infinite Loops, Stopping a Running Program, Nested Loops, More Math Functions,

# Chapter Six

## Input Range Testing, Tolerances

### KeyWord: BEEP

This chapter will start us off by learning how to check data that was input by an operator using an INPUT statement. Let's assume that you have a program that requires the user to input a positive number, and that zero can be allowed. What happens if the user enters in a negative number? How do you check it, and how can you correct it? Let's find out! Here is a section of program code that has our input checking property:

```
...
GETNUM:
PRINT "Enter a positive number";
INPUT A
IF A >= 0 THEN GOTO NUMOK
PRINT "You have entered a negative number! Try again"
GOTO GETNUM
NUMOK:
LET X=2*(A/D)
...
```

Let's trace through this program section to see what it does. When the program gets to the second line, it prints out the message indicating what the user needs to do. Line 3 then prints a question mark on the screen and waits for the user to type in a number. When the user enters the number, it gets assigned to the variable A, and the program continues to line 4. Here we have one of those wonderful IF...THEN commands. What it does is check the value of A to see if it is greater than or equal to (remember those from chapter 2?) zero. If it is, we have a valid response and the program jumps to line 7. If not, the program drops down to line 5 and prints out the "now look what you did" message. Then line 6 tells us to go to line 1, where the user is again asked to enter the positive number, and we do it all over again.

Now we get to write a "really neat" program. It actually calculates square roots. The method that is used is called "divide and average", and it works like this. Take the number you want to find the square root, and divide it by some arbitrary number. In our case, we'll divide it by 2. Now we take that answer, and divide the original number by it. Now we average the first answer with the new answer, and divide the original number by that result. We now take what we get and average it with the answer we got before. We continue this crazy cycle until the two answers are identical, in which case we will have the square root. Let's run through it using a real number so that you can understand what on earth is going on around here. We'll use 16 since it's a perfect square.

Take the 16 and divide it by 2. You get 8. Now take the 16 and divide it by the 8, and you get 2. Now average the 2 and the 8 to get 5. Now we divide the 16 by 5 and get 3.2. We average our new answer of 3.2 with our old answer of 5 to get 4.1. Now we divide 16 by 4.1 to get (approximately) 3.9. Now we average the 3.9 and the 4.1 to get 4. Now divide 16 by 4 to get 4. Since the new answer and the previous average are the same, we have arrived at the square

root. Let's put that into QBASIC. We will start out by asking for a number, so our first lines will be PRINT and INPUT commands to get a number. Be sure to clear out any program that is in memory:

```
PRINT "Enter Number to Find Square Root";
INPUT NUM
```

You will notice that we have to keep track of two answers. We will refer to them as new answer and old answer. We'll use the somewhat descriptive variable names NEWANS and OLDANS to keep track of them:

```
LET OLDANS = NUM / 2
```

Next, we divided the original number by the old answer to get a new answer. We will also need a label here for our loop. We'll use an abbreviation of "Calculate Answer" without spaces (you can't have spaces in label names).

```
CALCANS:
LET NEWANS = NUM / OLDANS
```

Next, we average the two answers together.

```
LET OLDANS = (NEWANS + OLDANS) / 2
```

After that, we go back and do it again.

```
GOTO CALCANS
```

Now, we need to check when the answers are the same. We'll do that right after we do the division, so insert this line right after the LET OLDANS= line:

```
IF NEWANS = OLDANS THEN GOTO FINISHED
```

When we finally get the answer, all we have to do is print it out and end the program. Add this to the bottom of the program:

```
FINISHED:
PRINT "Square Root Of "; NUM; " Is"; NEWANS
END
```

Just to make sure you have it in right, here's what we have so far:

(starts on next page)

```
PRINT "Enter Number to Find Square Root";
INPUT NUM
LET OLDANS = NUM / 2
CALCANS:
LET NEWANS = NUM / OLDANS
LET OLDANS = (NEWANS + OLDANS) / 2
IF NEWANS = OLDANS THEN GOTO FINISHED
GOTO CALCANS
FINISHED:
PRINT "Square Root Of "; NUM; " Is"; NEWANS
END
```

Now go ahead and run the program, using the value of 16. Notice that almost immediately it calculates the square root? Let's do it again! Run the program and type in 1234321 when it asks. Right away, you get back the correct answer of 1111. For those of you who have the slower PC's and XT's, you may detect about a tenth of a second delay between pressing the Enter key and the computer coming back with the answer. That is because the computer is going through that loop 13 times, doing all those calculations. If you have a 200 Mhz Pentium system, you almost get the answer before you ask the question because it's so fast! Anyway, let's try another number! Run the program again, and type in 24. Hey! Who told the computer to take a coffee break? When do I get my answer? Well, you will never get your answer! To stop the computer, hold down the Ctrl key while you press the Break key. This is called the "Control-Break" sequence. You probably never would have guessed that in a kazillion years. Here is where we need to talk just a little bit about how a computer sees numbers (and letters, for that matter). In our numbering system, we have ten digits, zero through nine. Well, a computer runs on electricity, and the electricity can only be on or off. Therefore, a computer can only use two digits, zero and one. For the most part, the conversion between these two numbering systems is pretty much accurate and hassle free. For example, what we see as 47, the computer sees as `00101111`. When you tell the computer 113, it is really thinking `01110001`. However, when you tell the computer 0.3, it has to think `0.010011001100110011001100110011001100110011001100110011001100110011001`... and has a headache, because that is a never-ending sequence. So, the computer has to chop it off at about 38 digits to keep from using too much memory, and yet still contain some accuracy. When it goes back and forth between the two numbering systems, it will sometimes generate this "rounding-off error", causing two things that are supposed to be exactly equal look like they aren't. This is the problem that our little program is having with the number 24. Well then, how do we fix that? Instead of checking if the two answers are equal, we will check to see if they are very, very, very close. How do we do that? Well, if two numbers are very close, when you subtract one from the other, you will get an answer that is very near zero. Let's fix up our program to take this "tolerance" into account. Replace the line that says

```
IF NEWANS = OLDANS THEN GOTO FINISHED
```

with:

```
IF ABS(NEWANS - OLDANS) < .00001 THEN GOTO FINISHED
```

Now if you run the program, it will work fine. Notice that we took the absolute value of the difference? That way, we would get a positive result no matter which way the numbers are. If the difference of these two numbers is less than 0.00001, a really really really small number,

then we will assume that they are the same. However, there is a minor flaw with this method. As we find the square roots of larger numbers, the tolerance become apparently much smaller, because the answers are bigger. For huge numbers, say 30 digit numbers, we may still get the computer to lock up. We could use a slightly bigger number on the right side of the less-than sign, but that would reduce the accuracy of the result for the smaller answers. So how do we fix that? We can make the tolerance change by not using an actual number, but instead by using a percentage of the original number. This will make the tolerance very tight for small numbers, and loosen it for huge numbers. To do that, we simply multiply our original number by a small constant. The corrected line will look like this:

```
IF ABS(NEWANS - OLDANS) < .00001 * NEWANS THEN GOTO FINISHED
```

Now the answer will be within .001% of the original number. That's pretty accurate, isn't it? There is still a major problem with the program. Run it again, and try to find the square root of -16. Oh, look! Another coffee break! Why? Each time it finds a new answer, the sign changes! First positive, then negative, then positive again, and so on. It will never zero in on a number but jump around wildly as it tries to average a positive and negative number. If the two numbers are close, but one is positive and the other negative, it will average out to be close to zero. When you divide a number by another very small number, you get a very large answer, and then it will average the new large answer with the old small number. How do we stop this? We don't let the user enter a negative number. How? We showed you at the beginning of the chapter. See if you can figure it out first, but first notice that entering a zero will also give you a "Division by zero" error - we don't want that, either.

Here is the program segment you need to enter to do the job. Between the lines

```
INPUT NUM
```

and

```
LET OLDANS = NUM / 2
```

enter:

```
IF NUM > 0 THEN GOTO ENTEREDOK
BEEP
PRINT"Number must be greater that zero - Try again"
GOTO TOP
ENTEREDOK:
```

and at the top, add:

```
TOP:
```

Whoa! Look at the line that says "BEEP"! what does that do? What do you think it might do? Would you like to find out? Then type in BEEP in the immediate window and listen closely! To be completely technical, it produces an 800 Hz tone for 0.27 seconds. In real English, it makes the computer beep. I guess in this case you could call it an error alarm! It alerts the user that he has made an input error, and is accompanied by the "nice going, stoopid!" message. The

user is then allowed to try again.

Just to make sure you have everything typed in correctly, here is the program in it's entirety:

```
TOP:
PRINT "Enter Number to Find Square Root";
INPUT NUM
IF NUM > 0 THEN GOTO ENTEREDOK
BEEP
PRINT "Number must be greater that zero - Try again"
GOTO TOP
ENTEREDOK:
LET OLDANS = NUM / 2
CALCANS:
LET NEWANS = NUM / OLDANS
LET OLDANS = (NEWANS + OLDANS) / 2
IF ABS(NEWANS - OLDANS) < .00001 THEN GOTO FINISHED
GOTO CALCANS
FINISHED:
PRINT "Square Root Of "; NUM; " Is"; NEWANS
END
```

If you want to see the intermediate answers, add the program line that's in color:

```
TOP:
PRINT "Enter Number to Find Square Root";
INPUT NUM
IF NUM > 0 THEN GOTO ENTEREDOK
BEEP
PRINT "Number must be greater that zero - Try again"
GOTO TOP
ENTEREDOK:
LET OLDANS = NUM / 2
CALCANS:
LET NEWANS = NUM / OLDANS
PRINT "Intermediate Answer is"; NEWANS
LET OLDANS = (NEWANS + OLDANS) / 2
IF ABS(NEWANS - OLDANS) < .00001 THEN GOTO FINISHED
GOTO CALCANS
FINISHED:
PRINT "Square Root Of "; NUM; " Is"; NEWANS
END
```

That's all for this time, but here are some programs for you to try your hand at:

INPUT a three digit number, and print it's reversal. In other words, given 275, print 572. You will need input range checking for this program to work. An original number ending with zero (420) can be printed out without leading zeros (24 is okay, since the computer won't print 024). If you would prefer, you may print it out as a three-digit number (024), but the PRINT command will automatically put spaces between each number (that's okay). HINT: Use division and either the INT(x) or FIX(x) functions.

INPUT a four digit number and print its reversal. In other words, given 9371, print 1739. Use the same hint as above.

Given a positive number less that 20,000, (Use INPUT and error checking) determine if it is a prime number. A prime number can be divided evenly only by 1 and itself. A word of caution here. On the lower speed computer, the larger numbers (over 10,000) can take over a minute to calculate.

Next chapter, we will learn about random number generation, and talk about a major problem: storing large amounts of data in variables. See you next time!!

## Introduced In This Chapter:

*Keyword*: BEEP

*Concepts*: Input range checking, tolerance, audible signaling.

Continues On the Next Page

# Chapter Seven

## Random Numbers

### KeyWords: RND, RANDOMIZE, TIMER

Here are three programs that will solve the problems at the end of chapter six. Keep in mind that your label and variable names will be different, and some of you more ingenious people may have found a more efficient method of doing them. The programs listed here work, but they use the simplest logic available, except for the Prime Number program, which has one little improvement. The reversal programs can be written with fewer variables. To test your programs, just run them and make sure they give you the correct answers. Here is the program that reverses a three-digit number:

```
GETNUM:
PRINT "Input a Three-Digit Number";
INPUT NUM
IF NUM - INT(NUM) <> 0 THEN GOTO GETNUM
IF NUM < 100 THEN GOTO GETNUM
IF NUM > 999 THEN GOTO GETNUM
LET QUOTIENT = NUM / 100
LET DIG3 = FIX(QUOTIENT)
LET NUM = NUM - 100 * DIG3
LET QUOTIENT = NUM / 10
LET DIG2 = FIX(QUOTIENT)
LET NUM = NUM - 10 * DIG2
PRINT NUM; DIG2; DIG3
PRINT DIG3 + 10 * DIG2 + 100 * NUM
END
```

Notice that we used a method that allows us to print out the answer in both methods suggested in the problem for the leading zero case. To turn it into a 4-digit program, we simply expand on the 3-digit algorithm (set of instructions) to handle one more digit, and adjust the input checking accordingly. Here is our program:

```
GETNUM:
PRINT "Input a Four-Digit Number";
INPUT NUM
IF NUM - INT(NUM) <> 0 THEN GOTO GETNUM
IF NUM < 1000 THEN GOTO GETNUM
IF NUM > 9999 THEN GOTO GETNUM
LET QUOTIENT = NUM / 1000
LET DIG4 = FIX(QUOTIENT)
LET NUM = NUM - 1000 * DIG4
LET QUOTIENT = NUM / 100
LET DIG3 = FIX(QUOTIENT)
LET NUM = NUM - 100 * DIG3
LET QUOTIENT = NUM / 10
LET DIG2 = FIX(QUOTIENT)
LET NUM = NUM - 10 * DIG2
```

```
PRINT NUM; DIG2; DIG3; DIG4
PRINT DIG4 + 10 * DIG3 + 100 * DIG2 + 1000 * NUM
END
```

Notice how the two programs are very similar? You may trace through the program listings yourself to discover the logic that we used. Chances are it is very close to the way you did it, although it is possible to do it backwards. Not easy, but possible. Instead of working with the quotient, you work with the remainder when you divide.

Let's move on to the Prime Number program. To find out if a number is prime, all we have to do is divide it by every integer between 2 and (itself-1). If we get an integer as an answer (no remainder), we have found a factor, and the number is not prime. However, to speed up the program a little bit, you will notice that factors always come in pairs, and the largest factor can never be more than half of the number. So to double the speed of the program, you only have to divide the number by every integer from 2 to 1/2 of the number. Here is the program (Notice how we test that the number is an integer in lines 5 and 10):

```
GETNUM:
PRINT "Enter an Integer between 2 and 20,000 and"
PRINT "I will tell you if it is a Prime Number"
INPUT NUM
IF NUM - INT(NUM) <> 0 THEN GOTO GETNUM
IF NUM < 2 THEN GOTO GETNUM
IF NUM > 20000 THEN GOTO GETNUM
PRIME = 0
FOR TRY = 2 TO NUM / 2
    IF NUM / TRY <> INT(NUM / TRY) THEN GOTO TRYMORE
    PRINT "Factor Found:"; TRY
    PRIME = 1
TRYMORE:
NEXT TRY
IF PRIME = 1 THEN GOTO NOTPRIME
PRINT "Number is Prime!"
GOTO DONE
NOTPRIME:
PRINT "Sorry, Number is not prime."
DONE:
END
```

There is one "trick" in this program, and you may have noticed it in the variable PRIME. This is what is called a FLAG variable. At the beginning of the program, this flag is set to zero, which in our case means we have not yet found a factor. We then divide the number by the first integer (2). If it does not turn out to be a factor, it jumps down to the NEXT command, and tries the next integer. On the other hand, if 2 is a factor, then the variable PRIME is assigned the value of 1, indicating we have found a factor, so we run it up the flag pole and see if anyone salutes. Got it? When the FOR...NEXT loop has finished, we drop down to line 15. If the entire loop has gone without finding a factor, then lines 11 and 12 would never have been executed, and PRIME would still be zero. We then drop down to line 16 and print out that the number is prime. If PRIME is 1, then line 15 is true, and the program branches to line 18, stating to the user that the number isn't prime. Notice also that line 11 prints out the factors as it finds them.

Let's get into some new material. We'll start out with the easier of the two ideas, random numbers. Random numbers are just that. You don't know what the next one will be. The function to generate a random number is RND. Type in this cute little program:

```
FOR X = 1 TO 5
    PRINT RND
NEXT X
END
```

Now go ahead and run the program. RND always returns a number between (but not including) zero and one. Also, you may be interested to know that these are not really random numbers. Run the program three or four more times. Notice anything? Yes! You get the same sequence of numbers every time!! We'll show you how to get around that later. RND can also be used with an argument, like such: RND(x). Different values of x will produce different results. If x is positive, you will get the next number in the sequence. If x is negative, you will start the sequence all over again. If x is zero, RND repeats the last number generated.

How do we get a different number each time we run our program? QBASIC has included a function that allows us to "seed" or initialize the random number generator. It is the RANDOMIZE function. Add this line to the top of our program:

```
RANDOMIZE
```

That's all there is to it. Now run the program. When the computer sees the RANDOMIZE function, it prints out a prompt:

```
Random number seed (-32768 to 32767)?_
```

and waits for you to enter an integer within the specified range. That allows you 65,536 different random sequences, which should be more than enough. Notice, however, that when you enter the same seed number, you get the same sequence of random numbers. Also, we probably don't want our user to have to come up with a number off the top of their head. How do we get around that? RANDOMIZE will also allow us to use an argument. QBASIC also has a function that returns a different number all the time, and only returns the same number once per day. It is the TIMER function, and all it does is tell you how many seconds it has been since midnight. It operates off of the DOS clock, so if you set the date and time correctly, the count will be pretty accurate. Let's see how many seconds have passed since midnight. Type in (in the immediate window)

```
PRINT TIMER
```

and press Enter. If it is in the morning, you will get a pretty small number. In the afternoon, you will get a good sized number, and late at night, you will get an absolutely huge number. Incidently, there are 86,400 seconds in a day. The only problem is, our random number seed can only be up to about 33,000. How do we fix that? Easy enough. Just divide TIMER by three, and the largest you will get is 28,800. That is easily within range, and you will get a new seed every time. Once the random number is seeded in a program, there is no need to keep on reseeding it. Let's fix up our top line to read:

```
RANDOMIZE TIMER/3
```

Now run the program. QBASIC will take into account the numbers after the decimal point for the seed and give you a different sequence of number no matter how fast you run the program.

The next problem we have is that RND only gives us numbers from 0-1. What if we want integers from 1 to 10?. It seems that we would want to multiply our random number by 10. This is true, because it will give us numbers between 0 and 10. But we want 1 through 10! If you take the integer part of the number, you will get numbers of 0 to 9. Remember that RND will never generate either a zero or a one. It will only be between these two numbers. Therefore, when we do either an INT(x) or FIX(x) we will get numbers from 0 to 9. Now we just add 1 to that result, and we will get numbers from 1 to 10. Here is the basic rule for generating a set of random integers:

*To generate a set of random integers in the range of x to y, first multiply the random number by (y-x)+1, take the integer portion, then add x.*

It works fine every time. For real numbers, just leave off the integer stuff. Let's put some of this into a program. Let's write a program that generates and prints out 20 random numbers in the range of 1 to 20 and see what happens:

```
RANDOMIZE TIMER / 3
FOR COUNT = 1 TO 20
    LET NUM = RND
    LET NUM = INT(20 * NUM) + 1
    PRINT NUM
NEXT COUNT
END
```

It's all pretty simple, really. Now why did we introduce all this stuff about random numbers here? Yes, there is a method to all this madness! Our next program will be a major sort program. We will be sorting a list of 50 numbers, and it is easier to let the computer generate the 50 numbers than to have either the programmer enter them with a DATA statement, or the user INPUT them. We'll let the computer do all the work. It's faster and easier that way.

Before we begin our sort program, we have to let you in on a little secret. Actually it is a big secret. No, really it's more of a powerful secret. As a matter of fact, it isn't really much of a secret at all! Here's the secret: Did you know that it is possible for a single numeric variable to have more than one value at the same time? How do we do that? It's by using subscripts. Those of your familiar with other languages will know them as arrays. We'll explain them next chapter, but to find out why we need them, let's average a set of numbers. Let's say we needed to average ten numbers using a READ...DATA statement. Our program would look like this:

```
READ A,B,C,D,E,F,G,H,I,J
LET X = (A+B+C+D+E+F+G+H+I+J)/2
PRINT X
DATA ...
END
```

What's wrong with that, you ask? Nothing! The program works fine. But suppose you needed to average 100 numbers? You would need to read in 100 numbers into 100 different numeric variables, and the READ and LET statements would become so long as to be ridiculous! I see some light bulbs coming on out there! Some of you are saying "Why not just add each piece of data to a variable as you read it in? It would only take two variables total for the program!" Yes, that would work absolutely perfectly. Let's take it one step further, though. Say we need to determine how far away from the average each piece of data is - kind of a poor man's standard deviation, if you please. Now we have a problem. We need to keep all of the pieces of data stored in a variable, because we first need them to determine the average, then we have to go back to them again to subtract them from the average. The logic would go something like this (this is what our flowchart would look like):

*1. Read and store a list of numbers.*
*2. Find the sum of the list.*
*3. Compute the average of the set.*
*4. Subtract the average from each number in the list.*
*5. Print each number and it's distance from the average.*
*6. End.*

Steps 1 and 2 could be done in a loop as talked about above, as could steps 4 and 5. If you have a list of ten numbers, this program would work fine:

```
READ A,B,C,D,E,F,G,H,I,J
LET X = (A+B+C+D+E+F+G+H+I+J) / 10
PRINT A,A-X
PRINT B,B-X
PRINT C,C-X
PRINT D,D-X
PRINT E,E-X
PRINT F,F-X
PRINT G,G-X
PRINT H,H-X
PRINT I,I-X
PRINT J,J-X
DATA ...
END
```

For a list of ten items, fourteen statements were required. No loop was used. For a list of 100 items, 104 lines would be needed! HELP! THERE HAS TO BE A BETTER WAY! There is. But we're not going to tell you about it yet, because I see that we have no more room to tell you about it. We will just say that this is where subscripted variables come in handy. Meanwhile, try to write some programs using random numbers. Here are a few suggestions:

Generate 1000 random digits (0-9) and count how many times each one occurs.

Flip a coin 100 times and count how many heads and how many tails you get. Here are two methods to use:

If the generated number is <0.5, it's heads; if the generated number is >0.5, it's tails.

Convert the random number to an integer in the range of 1-2. 1 is heads, 2 is tails.

Answers next chapter. See you then!

## Introduced In This Chapter:

*Keywords*: RND, RANDOMIZE, TIMER

*CONCEPTS*: Using variables as flags, Random numbers, Seeding the random number generator, Using the DOS clock as a seed value, Problems with storing large amounts of data.

Continues On the Next Page

# Chapter Eight

## Subscripted Variables

### Keyword: DIM

Before we dive into subscripts this chapter, we'll give you the answers to the problems at the end of chapter seven. Here is the program listing for the 1000 random integers from 0 to 9 (notice how we used blank lines to make it a little easier to read):

```
RANDOMIZE TIMER / 3
FOR N = 1 TO 1000
    NUM = INT(10 * RND)
    IF NUM <> 0 THEN GOTO TRYONE
    LET ZERO = ZERO + 1
    GOTO NEXTNUM

TRYONE:
    IF NUM <> 1 THEN GOTO TRYTWO
    LET ONE = ONE + 1
    GOTO NEXTNUM

TRYTWO:
    IF NUM <> 2 THEN GOTO TRYTHREE
    LET TWO = TWO + 1
    GOTO NEXTNUM

TRYTHREE:
    IF NUM <> 3 THEN GOTO TRYFOUR
    LET THREE = THREE + 1
    GOTO NEXTNUM

TRYFOUR:
    IF NUM <> 4 THEN GOTO TRYFIVE
    LET FOUR = FOUR + 1
    GOTO NEXTNUM

TRYFIVE:
    IF NUM <> 5 THEN GOTO TRYSIX
    LET FIVE = FIVE + 1
    GOTO NEXTNUM

TRYSIX:
    IF NUM <> 6 THEN GOTO TRYSEVEN
    LET SIX = SIX + 1
    GOTO NEXTNUM

TRYSEVEN:
    IF NUM <> 7 THEN GOTO TRYEIGHT
    LET SEVEN = SEVEN + 1
    GOTO NEXTNUM
```

There's More On the Next Page!!

```
TRYEIGHT:
    IF NUM <> 8 THEN GOTO ISNINE
    LET EIGHT = EIGHT + 1
    GOTO NEXTNUM

ISNINE:
    LET NINE = NINE + 1

NEXTNUM:
NEXT N

PRINT ZERO; "Zeros", FOUR; "Fours"
PRINT ONE; "Ones", FIVE; "Fives", EIGHT; "Eights"
PRINT TWO; "Twos", SIX; "Sixes", NINE; "Nines"
PRINT THREE; "Threes", SEVEN; "Sevens "
END
```

Notice that we used a different variable to represent each digit that could be generated, and the somewhat long series of IF...THEN statements to assign the digit to the appropriate variable. We will show you this program again later in this chapter, but using subscripts. It will bring the program to less than half of its current size. Also notice the size of our FOR...NEXT loop. The computer really doesn't care how long it is, as long as it has a FOR and a NEXT. Our indentation help us make sure that there is one of each.

Let's get to the 100 coin tosses. Here is the version for the 0.5 "breakpoint" method. Notice again our use of tabs and blank lines to enhance the readability:

```
RANDOMIZE TIMER / 3
FOR FLIP = 1 TO 100
    COIN = RND
    IF COIN < .5 THEN GOTO ISHEADS
    LET TAILS = TAILS + 1
    GOTO NEXTFLIP
ISHEADS:
    LET HEADS = HEADS + 1

NEXTFLIP:
NEXT FLIP

PRINT HEADS; "Heads, and "; TAILS; "Tails."
END
```

Notice that we don't do any calculations to the random number, just check its value. Here is the other method of converting the random number to either a 1 or 2:

```
RANDOMIZE TIMER / 3
FOR FLIP = 1 TO 100
    COIN = RND
    COIN = INT(2 * COIN) + 1
    IF COIN = 1 THEN GOTO ISHEADS
    LET TAILS = TAILS + 1
    GOTO MOREFLIP
ISHEADS:
```

```
    LET HEADS = HEADS + 1
MOREFLIP:
NEXT FLIP

PRINT HEADS; "Heads, and "; TAILS; "Tails."
END
```

At the end of chapter seven, we developed a program to find the average of a list of numbers, and then determine the distance of each element from the average. As you recall, the program we developed wasn't too bad for ten elements, but then we wanted to find out for a list of 100 elements! This requires a program that is 104 lines long and contains 101 different variables. To handle something like this, QBASIC has implemented what are called subscripted variables. A subscripted variable in QBASIC looks like this: A(1) and is pronounced "A Sub One". The subscript is placed in parentheses after the variable being subscripted. That may not seem too impressive, but the subscript itself can be a variable, and can be as large as necessary! For our first averaging program, we would need 10 numbers ranging from A(1) to A (10). In the second example, we could store the 100 numbers as A(1) to A(100). In both cases, the values are all stored in the variable A, and the subscript could be generated with a FOR... NEXT loop.

A variable that has its data stored in subscript form is called an array. We have been talking about one-dimensional arrays so far, but QBASIC allows multi-dimensional arrays. For example, a two-dimensional array would look like A(1,2), and a three-dimensional array would look like A(4,3,6).

Let's can the chatter and get into some programming. Here is the first finger exercise for this chapter:

```
FOR I = 1 TO 10
    READ A(I)
NEXT I
FOR I = 1 TO 10
    LET X = X + A(I)
NEXT I
LET X = X / 10
FOR I = 1 TO 10
    PRINT A(I), A(I) - X
NEXT I
DATA 51, 54, 56, 81, 79, 12, 60, 34, 67, 19
END
```

This program does the same thing as the one in chapter 7, but takes 8 fewer variables. Go ahead and run the program. Now we'll adjust the program to use 20 elements. The green lines are the changed or new ones:

```
FOR I = 1 TO 20
    READ A(I)
NEXT I
FOR I = 1 TO 20
    LET X = X + A(I)
NEXT I
```

Don't stop typing yet, there's another line on the next page!!

```
LET X = X / 20
FOR I = 1 TO 20
    PRINT A(I), A(I) - X
NEXT I
DATA 51, 54, 56, 81, 79, 12, 60, 34, 67, 19
DATA 16, 27, 80, 25, 13, 97, 52, 44, 5, 64
END
```

Now go ahead and run the program. Hey! What happened? Well, you get a "Subscript out of range" error! Why is that? It's because the computer needs to have a separate memory location for each element of the array, and large arrays can take up a lot of memory. The computer will automatically set aside ten "slots" for each subscripted variable, since ten does not take up too much memory. But what if we need, say, 100 slots? How do we tell the computer to set aside the necessary memory locations? We use the DIM command. DIM is short for Dimension. DIM tells the computer how much memory to reserve for an array. There is no harm in reserving more memory than is actually needed, provided you don't exceed the memory capacity of QBASIC. Here is what a DIM command looks like:

```
DIM A(100)
```

It looks just like a subscripted variable. The number inside the parentheses tells the computer the maximum number of subscripts you will use for that variable. If you try to use a larger one later on in the program, you will get that "Subscript out of range" error again. Let's fix up our little program to tell the computer to expect an array of 20 elements. Add this line at the very top of the program:

```
DIM A(20)
```

Now you may run the program. DIM commands are executed by QBASIC, so the DIM command must come before you try to put anything into the array. It is customary to put your DIM's at the beginning of the program.

Remember that we said we would rewrite our "1000 random digits" program using subscripted variables? Let's do it now. The first thing we need to do is clear out any program in memory. In our new program, we don't really need to tell the computer that we are using an array, since we are only going to use ten slots. However, it's a good idea to do it anyway so that you or anyone else who comes along later will know that arrays are being used. All we need to do is generate the digits and assign them to the array. Here is the first part of the program:

```
DIM A(10)
FOR NUM=1 TO 1000
    LET X = INT(10 * RND + 1)
    LET A(X) = A(X) + 1
NEXT NUM
```

Notice in line 3 how we generate the integers from 0 to 9 "on the fly"? Our random integer is assigned to the variable X. That X is then used as the subscript to the variable A. The A variable keeps a count of how many times each integer is generated. Just in those four lines, we have eliminated the long IF...THEN series that we had in the original program! Some of you may have noticed that zero can be used as a subscript, as it is here. All we have to do now is

to print out the results. Here is the rest of the program:

```
FOR NUM=0 TO 9
    PRINT NUM;"Appeared "; A(NUM); "Times."
NEXT NUM
END
```

Much simpler now, isn't it? Is the power of subscripted variables (also called arrays) becoming apparent? What took 46 lines to do without an array takes only 9 with it.

We will now introduce an algorithm that will probably be quite useful in many programs you write. Frequently, you will need to sort a set of numbers into either ascending or descending order. Usually, the sorting algorithm is part of a larger program. For example, one program that I have written in QUICKBASIC keeps track of all the employees in a medium-sized company. Each employee has his or her own employee number. As employees are added and removed, the list tends to get out of numerical order. To fix the situation, I added a menu option to re-sort the data either by employee number or by last name. The last-name sort is quite a bit more complex, but we will show you how it can be done when we discuss alphanumeric (or string) variables in future chapters. But for now, we have all of the necessary tools to generate and sort a list of numbers. We will show you how to sort a list of numbers in this chapter, and write the actual program in the next chapter.

Let's suppose we have a list of 7 numbers in an array:

14,2,5,12,19,11,17

We need to sort this list into decending order. You would probably scan the list from left to right to find the largest value, 19. You would then scan again for the next largest, 18. You would repeat this process until you had completely sorted the array.

Sounds simple, doesn't it? Well, there is only one teensy weensy problem. The computer can't "see" all of the values. It must follow a more detailed set of instructions, as it can only look at two numbers at a time. Here is how the instructions would go:

Compare the first number to the second. If the first is larger, leave the numbers where they are. If the first is smaller, exchange the two numbers. If they are equal, leave them alone. Now compare the first with the third. If the first is greater than or equal to the third. leave them alone. If the first is less than the third, swap them around. Now compare the first to the fourth in a similar manner. Continue until the first number has been compared to all of the following elements.

This compare and swap routine described is called one pass through the array. When this first pass is completed, the first element has the largest number in it. There may be other elements with the same value, but none are greater. After the first pass, our list would look like this:

19,2,5,12,14,11,17

You can see that the first number is the largest. Now we need to continue by comparing the second number to all remaining numbers, by doing the same thing. After our second pass, the array will look like this:

19,17,2,5,14,11,12

The computer continues the algorithm until we get to the end of the list, after which the array will look like:

19,17,14,12,11,5,2

Those of you familiar with sort algorithms will recognize this as the exchange sort.

Well, we've covered quite a bit in this chapter, so we'll break here and continue our discussion of sorting in chapter nine. In the meantime, here is a program for you to try:

Generate 100 random numbers ranging from 25 to 50, and calculate the standard deviation of the group. Standard deviation indicates how "spread out" or "scattered" the data is. To calculate standard deviation:

1.) Find the average of the list
2.) Subtract each item in the list from the average, and square the difference
3.) Add up all of the squared differences
4.) Divide the sum of the squared differences by the number of items in the list (100 for our program)
5.) Take the square root of the quotient.

*HINTS:* When the program is written correctly and you have a good random number generator, your answers should range between about 6.9 and 7.5 each time you run the program. Don't forget to use RANDOMIZE TIMER / 3 to seed the random number generator each time the program is run. To find the square of a number, use the following operator: (number)^2 or if necessary ((number)^2) and make sure you have the same number of closing parentheses as you do opening ones. To find the square root of a number, use SQR(number).

In our solution, we will show you a neat little trick with the DIM command and FOR...NEXT loops that allow you to change the number of items by either changing one line or using an INPUT statement.

If you want to try to write the sort program on your own, go ahead! Here is a hint or two: It uses only one array, and nested loops to cycle through the array. The initial value for the inner loop is the current value of the outer loop plus one.

See you next chapter!

Introduced in this chapter:

*Keywords*: DIM

*Concepts*: Arrays and subscripted variables, Dimensioning large arrays, multi-dimensional arrays.

# Chapter Nine

## Sorting

### Keyword: SWAP

Here is our solution to the standard deviation program:

```
LET ITEMS = 100
DIM A(ITEMS)
RANDOMIZE TIMER/3
FOR NUM = 1 TO ITEMS
    LET A(NUM) = INT(26*RND) + 25
NEXT NUM
FOR NUM = 1 TO ITEMS
    LET SUM = SUM + A(NUM)
NEXT NUM
LET AVE = SUM / ITEMS
FOR NUM=1 TO ITEMS
    LET SQUARE = SQUARE + ((AVE-A(NUM))^2)
NEXT NUM
LET SQUARE = SQUARE / ITEMS
PRINT"Standard deviation is";SQR(SQUARE)
END
```

Did you spot our little trick with the DIM command? We first defined how many items we would have in our list. Then we dimensioned the array with a variable in line 2. We also used that same variable (ITEMS) in all of our FOR...NEXT loops that cycle through the array. Now we can easily change the number of items that we are "standard deviating" simply by changing line 1. Or you could replace line 1 with an INPUT statement to have the user specify how many items to be generated. Also, if you want to see just what a standard deviation does, try changing the range of the random numbers generated in line 5. Instead of using 26*RND, try 51*RND and 6*RND and see what you get for a standard deviation. You will notice that as the data is less scattered, the standard deviation will drop way down.

Were you able to write any kind of a sort program? If you did, congratulations. If you didn't, don't feel bad; it isn't as easy as it sounds. In this chapter, we are going to walk through the development of the program piece by piece. The first thing we need to do is clear out any program in memory. Now we have a clean slate to start off with. The very first thing we need to do is tell the computer how many items we will be sorting. All of those items will be stored in one array. Let's use the trick that we used in the standard deviation program. First, we assign the number of items we will be generating and sorting to the variable ITEMS. Then we will dimension the array A using the variable ITEMS. Our first two lines will look like this for generating fifty items:

```
LET ITEMS = 50
DIM A(ITEMS)
```

Now that we have that out of the way, let's seed our random number generator:

```
RANDOMIZE TIMER/3
```

Next, we will generate the random numbers and place them into our array using a FOR...NEXT loop. Leave a blank line or two between this section and the previous one:

```
FOR NUM = 1 TO ITEMS
    LET A(NUM) = INT(100 * RND) + 1
NEXT NUM
```

Can you guess from that second line what range our random numbers will be in? They will range from 1 to 100. Now that we have all of our numbers stored into the array, we need to sort them. We will do it in ascending order. Do you recall how to do a pass through the array? You compare the first item in the array with all of the remaining items, one at a time. This sounds like a wonderful place for a FOR...NEXT loop, doesn't it? Yes it does. It looks like one loop ought to do the trick. It does, but only for the first pass. If you remember, the second pass compares the second element to all remaining ones, and the third pass compares the third element to all remaining ones, and so on. It sounds like another loop would come in handy here. That is correct. But you say we have two loops going at the same time? That is an ideal case for a pair of nested loops. Let's see, we will compare the first element of the array on only one pass, and our outer loop will always begin with the first element. So our loop would begin with FOR OUTER=1 TO something. When we get to the end of the array, do we have to compare the last element to itself to see if it is less than itself? No! So our complete loop statement would look like this (type it in):

```
FOR OUTER = 1 TO ITEMS - 1
```

Now what about our inner loop? As you recall, we compare the element in question with every succeeding element in the array. This means for our first pass, we start the inner loop with the second element. Hmmm, It sounds like we would use a FOR INNER = 2 TO something. That won't work, however, When we get to the second pass, we want to start with the third element, so our inner loop would look like FOR INNER = 3 TO something. Now what can we use for the changing starting point? Wait a minute! The outer loop's variable indicates which element is being tested for the largest! Why don't we start our inner loop with one more than what our outer loop is currently at? That is exactly what we need to do, so our inner loop will look like FOR INNER = OUTER + 1 TO something. Where does our inner loop end? Well, when we were scanning the array, we always went to the end of the array. What variable is holding the value for the length of the array? Right! It's ITEMS. Our complete FOR statement will look like this (Type it in, remember to indent):

```
    FOR INNER = OUTER + 1 TO ITEMS
```

Now we need to figure out what the comparison between our two variables needs to be for us not to swap the elements around (you'll see why we asked it this way later). We are sorting in an ascending order, so if the first element is less than or equal to the "later on down the line" element, there is no need to swap, so we will skip over our programming code to exchange the two elements in the array. Can you figure out the IF...THEN command that we will use (without the line label after the THEN, of course)? It will look like this (type it in):

```
        IF A(OUTER) <= A(INNER) THEN GOTO NOCHANGE
```

Now let's go ahead and finish up our loops with the necessary NEXT statements. Can you guess which will come first; INNER or OUTER? Try to guess without looking at the lines below. I know it's hard, what with them being in a nice bold type. Go ahead and enter these lines (we need the label for a certain reason):

```
NOCHANGE:
    NEXT INNER
NEXT OUTER
```

QBASIC has a command to exchange two variables, and we will be needing it here. Way back in the early days of BASIC (before the IBM PC's and such - Apple II, Radio Shack TRS-80, Commodore PET, any CP/M- based computer, etc.), such a command did not exist. The way to swap two variables was to put one in a temporary variable, put the second into the first, and then put the temporary into the second. That took up three lines. So what is this new, magical command, you ask? Well, it's the (hold onto your hat!) SWAP command! How does it work? Instead of explaining it, let's type in the actual program line (It's the one in color):

```
FOR OUTER = 1 TO ITEMS - 1
    FOR INNER = OUTER + 1 TO ITEMS
        IF A(OUTER) <= A(INNER) THEN GOTO NOCHANGE
        SWAP A(OUTER), A(INNER)
NOCHANGE:
    NEXT INNER
NEXT OUTER
```

Yes, there really is a Santa Claus! Basically, the command takes the form SWAP A, B where A and B are variables. In our case, we are using array elements - two different elements of the same array. Well, let's get back to the program. It is actually in a working form right now, but if you run it, the computer will go away for about a second and then give you the "Press any key to continue" message, meaning that it's finished. Why? We don't have any PRINT statements in it! Let's print out the sorted array using a FOR...NEXT loop (add this to the end of what we have so far):

```
FOR NUM = 1 TO ITEMS
    PRINT A(NUM),
NEXT NUM
END
```

Don't forget that comma at the end of the PRINT command! We will be printing out 50 numbers, and we want them all to fit on the screen! Now go ahead and run the program. Remember that the PRINT command prints from left to right, then top to bottom. The numbers will all be in ascending order. If they are not, check your typing carefully, especially for lines:

```
IF A(OUTER) <= A(INNER) THEN GOTO NOCHANGE
```

and

```
SWAP A(OUTER), A(INNER)
```

How do you know that the sort is really working? Maybe the numbers are being generated in the correct order mysteriously! Well, not so! Would you like to see what the array looks like before it is sorted? Add the green line, and don't forget the comma:

```
LET ITEMS = 50
DIM A(ITEMS)
RANDOMIZE TIMER / 3
FOR NUM = 1 TO ITEMS
    LET A(NUM) = INT(100 * RND) + 1
    PRINT A(NUM),
NEXT NUM
```

Now run the program. Do you see a problem? That's right, the sorted list starts right after the unsorted list. Do you know why? It's because of the comma at the end of line 7. It leaves the next print position on the same line, so when the PRINT command at the bottom comes along, it takes up where it left off earlier. In this case, however, there are 50 items that are printed out in 5 columns, so the comma does send the next print position down to the next line. How do we fix that? Do you remember the very first thing we did with the print command? If not, go back to chapter one and review. Now we are actually going to put that theory into a real program! Here is the line:

```
FOR OUTER = 1 TO ITEMS - 1
    FOR INNER = OUTER + 1 TO ITEMS
        IF A(OUTER) <= A(INNER) THEN GOTO NOCHANGE
        SWAP A(OUTER), A(INNER)
NOCHANGE:
    NEXT INNER
NEXT OUTER
PRINT
FOR NUM = 1 TO ITEMS
    PRINT A(NUM),
NEXT NUM
END
```

Now run the program again. See our blank line between the two lists? It kind of separates the two, doesn't it? Let's spruce up our output a little bit more. Right now, the computer operator would only see two lists of numbers. Why don't we print out what the computer is showing us? Let's add these PRINT commands:

```
LET ITEMS = 50
DIM A(ITEMS)
RANDOMIZE TIMER / 3
PRINT "Unsorted List:"
FOR NUM = 1 TO ITEMS
    LET A(NUM) = INT(100 * RND) + 1
    PRINT A(NUM),
NEXT NUM
PRINT "Sorting..."
FOR OUTER = 1 TO ITEMS - 1
    FOR INNER = OUTER + 1 TO ITEMS
```

"Continued Overleaf", as they say

```
        IF A(OUTER) <= A(INNER) THEN GOTO NOCHANGE
        SWAP A(OUTER), A(INNER)
NOCHANGE:
    NEXT INNER
NEXT OUTER
PRINT
PRINT "Sorted List:"
FOR NUM = 1 TO ITEMS
    PRINT A(NUM),
NEXT NUM
END
```

Now go ahead and run the program again. Looks a little nicer, doesn't it?

Let's do one more thing to this program. We'll print out the pass number that the sorting routine is executing. Can you figure out how to do it? Here are the changes to make:

```
    LET A(NUM) = INT(100 * RND) + 1
    PRINT A(NUM),
NEXT NUM
PRINT "Sorting Pass:"
FOR OUTER = 1 TO ITEMS - 1
    PRINT OUTER;
    FOR INNER = OUTER + 1 TO ITEMS
        IF A(OUTER) <= A(INNER) THEN GOTO NOCHANGE
        SWAP A(OUTER), A(INNER)
NOCHANGE:
```

Now run the program, and watch it go. Run it several times, and see how the list is correctly sorted every time? When you are through having fun with this program, make sure that you save it to disk (call it SORT), because we will be using it again in the next few chapters to learn more ways of formatting the screen's output.

For your reference, here is our entire sort program. You may use this listing to check your typing:

```
LET ITEMS = 50
DIM A(ITEMS)
RANDOMIZE TIMER / 3
PRINT "Unsorted List:"
FOR NUM = 1 TO ITEMS
    LET A(NUM) = INT(100 * RND) + 1
    PRINT A(NUM),
NEXT NUM
PRINT "Sorting Pass:"
FOR OUTER = 1 TO ITEMS - 1
    PRINT OUTER;
        FOR INNER = OUTER + 1 TO ITEMS
        IF A(OUTER) <= A(INNER) THEN GOTO NOCHANGE
        SWAP A(OUTER), A(INNER)
NOCHANGE:
    NEXT INNER
NEXT OUTER
```

```
PRINT "Sorted List:"
FOR NUM = 1 TO ITEMS
    PRINT A(NUM),
NEXT NUM
END
```

Here are a few things for you to practice your programming skills with:

The median of a list of numbers is that element such that when the numbers are sorted, half of the numbers lie above it, and half of the numbers lie below it. This happens only when there are an odd number of items in the list. If there are an even number of items, the median is the average of the middle two numbers. Have the program use an INPUT command to specify the number of items with a maximum of 200. Use input range checking. The only output necessary for this program is the median value.

Using the sorting program we developed this chapter, change it so that the numbers generated range between 1 and 1000.

Change the original sorting program so that it generates 150 items.

Change the original sorting program so that the list is sorted in descending order.

We'll give you the answers next time. Also in the next chapter, we will introduce some commands that will help you in making "custom" output displays for your program, including how to clear the screen, and how to print a character or number in a particular position. See you then!!

## Introduced in this chapter:

*Keywords*: SWAP

*Concepts*: Dimensioning an array from a variable, A sorting algorithm, Using PRINT commands to enhance a program's operation.

# Chapter Ten

## Output Formatting, Part One

### Keywords: CLS, LOCATE

Before we learn about screen formatting, here are the answers to the problems at the end of chapter 9. Here is the program to find the median of a list of numbers:

```
TOP:
PRINT "Enter Number Of Items to Generate";
INPUT ITEMS
IF ITEMS > 200 THEN GOTO TOP
IF ITEMS < 2 THEN GOTO TOP
DIM A(ITEMS)
RANDOMIZE TIMER / 3
FOR NUM = 1 TO ITEMS
    LET A(NUM) = INT(100 * RND) + 1
NEXT NUM
FOR OUTER = 1 TO ITEMS - 1
    FOR INNER = OUTER + 1 TO ITEMS
        IF A(OUTER) <= A(INNER) THEN GOTO NOSWAP
        SWAP A(OUTER), A(INNER)
NOSWAP:
    NEXT INNER
    PRINT ".";
NEXT OUTER
PRINT
IF ITEMS / 2 = INT(ITEMS / 2) THEN GOTO EVEN
PRINT A(ITEMS / 2); "Is the Median."
GOTO DONE
EVEN:
LET MID1 = INT(ITEMS / 2)
LET MID2 = MID1 + 1
MEDIAN = (A(MID1) + A(MID2)) / 2
PRINT MEDIAN; "Is the Median."
DONE:
END
```

You may have noticed that we added a line (in green) which will print out a single period for every pass. This is to let the operator know that the program is working, and the computer hasn't gone out to lunch. This also required adding the PRINT command after the sort (also in green) to give us a carriage return. Also notice that we handled the cases for an odd number of items and an even number of items separately.

Let's look at our sorting program again. Here it is:

```
LET ITEMS = 50
DIM A(ITEMS)
RANDOMIZE TIMER / 3
PRINT "Unsorted List:"
FOR NUM = 1 TO ITEMS
    LET A(NUM) = INT(100 * RND) + 1
    PRINT A(NUM),
NEXT NUM
PRINT "Sorting Pass:"
FOR OUTER = 1 TO ITEMS - 1
    PRINT OUTER;
    FOR INNER = OUTER + 1 TO ITEMS
        IF A(OUTER) <= A(INNER) THEN GOTO NOCHANGE
        SWAP A(OUTER), A(INNER)
NOCHANGE:
    NEXT INNER
NEXT OUTER
PRINT
PRINT "Sorted List:"
FOR NUM = 1 TO ITEMS
    PRINT A(NUM),
NEXT NUM
END
```

To change our original sorting program to generate values between 1 and 1000, simply change the green line:

```
FOR NUM = 1 TO ITEMS
    LET A(NUM) = INT(1000 * RND) + 1
    PRINT A(NUM),
NEXT NUM
```

To change the original program to generate 150 items, change the first line to read:

```
LET ITEMS = 150
```

As you may remember, we wrote the original program in such a way as to easily change the number of items. Why? It is because we used a variable for the final value in defining the FOR...NEXT loops. We don't have to rewrite the program every time we need to change the number of items.

To change the original program to sort in descending order, change the IF...THEN line to:

```
IF A(OUTER) => A(INNER) THEN GOTO NOCHANGE
```

You may have noticed that when you run a program, anything that is sent to the screen with a PRINT command goes right below the line that was just printed, assuming no commas or semicolons. When the output line gets to the bottom, you notice that the rest of the screen scrolls up a line to make room for the new stuff. That is all well and good, but it just seems like things get a little messy after a while. Take the sorting program from chapter 9, for example. Let's say you typed in the whole program. When you run the program, the unsorted list starts printing out below whatever was there before, with everything else scrolling off the top. Wouldn't it be neat if we could start off with a "clean slate"? It would be a little nicer to have the screen erased when the program starts. Well, QBASIC has a command to do just that. It is the clear screen command, and it looks like this: **CLS**. Let's get ready to try it. First, fill up the screen with something. If you have a program in memory, run it a couple of times. If you don't have any program in memory, type in `FILES` (a QBASIC command to display the files in the current directory) a couple of times in the immediate window to fill up the screen. Once your output screen is full, type in `CLS` in immediate window, and watch what happens. Bingo! Your output screen is cleared! Now suppose you have cleared your screen, and used a bunch of PRINT commands to print out some kind of screen display, such as a series of boxes or something. Now, how do you get your cursor to a certain position to print out a number or letter? We use QBASIC's command that positions the cursor to any part of the screen. It is the **LOCATE** command. How do we use it? Take a look at this program segment:

```
...
CLS
LOCATE 20
...
```

After the CLS command, your screen would be cleared, and your cursor would be at the top of the screen. After the LOCATE command, they would go to the twentieth line down. That's nice, but what if we wanted to type something way over in column 75? Then we would use `LOCATE 20,75`. Let's try using these in a simple program.

Here is a variation of our sort program that we have designed strictly for learning purposes. It is stuck with sorting only 20 items. Go ahead and type these lines in:

```
CLS
DIM A(20)
FOR LOOOP = 1 TO 20
    PRINT "Item number"; LOOOP; "is"
NEXT LOOOP
FOR LOOOP = 1 TO 20
    LET A(LOOOP) = INT(50 * RND) + 1
NEXT LOOOP
FOR LOOOP = 1 TO 20
    LOCATE LOOOP, 18
    PRINT A(LOOOP)
NEXT LOOOP
PRINT
PRINT "Sorting the List"
FOR OUTER = 1 TO 19
    FOR INNER = OUTER + 1 TO 20
        IF A(INNER) >= A(OUTER) THEN GOTO NOSWAP
        SWAP A(INNER), A(OUTER)
```

```
NOSWAP:
    NEXT INNER
NEXT OUTER
FOR EMPTY = 1 TO 50000
NEXT EMPTY
FOR LOOOP = 1 TO 20
    LOCATE LOOOP, 18
    PRINT A(LOOOP)
NEXT LOOOP
PRINT
PRINT "This Is the Sorted List"
END
```

Make sure that your variable LOOOP has 3 "O"'s in it, or you will get a bunch of error messages! The word LOOP is a reserved word in QBasic. Let's explain the green lines. What do they do? Nothing! All that they do it make the computer count from one to 50000. Why? Because it takes the computer a little bit of time to do that. This "holds up" the computer for about half of a second, so that you have a little time to look at the list while it gets sorted. The faster your computer runs, the less time it takes for it to count that high, and the shorter it waits. If you want the computer to wait longer, have it count up to a larger number, say 80000 or more. If you still need more time, have it count from -80000 to 80000. If you feel ambitious, you can use the TIMER command and an IF...THEN loop to have the computer wait a specific amount of time (in seconds). We'll show you that in the advanced series. The rest of the program is pretty straightforward. Go ahead and run it to see if it does what you expected.

The advantage of using the LOCATE statement is shown above, although it may not be too clear what with the screens popping up so fast. Let's take a closer look. Notice that we first cleared the screen, then dimensioned our array, and then printed our output screen? We haven't even generated the random numbers, yet! After we print the output screen, we then generate and assign our random numbers. Then using the LOCATE command followed by the PRINT command, we place the cursor where we want the number to print out. Then we print the number. See how we used our loop variable to position the cursor on the appropriate row? After that, we then sorted the list of numbers, and then had our empty FOR...NEXT loop to pause the computer for a bit. Then we printed out our sorted array, using the same technique for printing out the unsorted array. Did you notice that we didn't have to reprint out our "labels" again? Due to some technical reasons, this also makes the program run a bit faster. It takes a little bit of time o print out the labels, because the computer actually displays only one letter at a time. To print out each label takes up a lot of internal instructions in the computer. We won't go into it, but it is quite complicated!

For your information, most screens are 80 column wide and 24 lines high. That means your LOCATE command ranges from 1,1 for the top left, and 24,80 for the bottom right. There are a couple of exceptions to this, however, and they will be covered in the advanced series. Meanwhile, here are a couple of programs for you to try:

On the screen, have two headings, HEADS and TAILS. Flip a coin 2,500 times, and keep a tally of how many heads there were, and how many tails. Update the count on the screen after every coin flip, but only update the needed column. When you initially create the screen, go ahead and put a zero under each heading.

Using a FOR...NEXT loop and a LOCATE statement, draw a diagonal line of *'s on the screen from 1,1 to 22,22

Modify the above program to draw the "line" from 1,1 to 22,44. HINT: You can do calculations within the LOCATE command.

Next chapter, we will learn a very powerful option for the PRINT command that can do some really amazing formatting, especially when coupled with the LOCATE command we learned here. See you next time!!

## *Introduced In This Chapter:*

*KeyWords*: CLS, LOCATE

*Concepts*: Clearing the screen, Placing the cursor at a particular print position on the screen without erasing the screen.

# Chapter Eleven

## Output Formatting, Part Two

### Keyword: PRINT USING

As usual, here are the answers to the problems at the end of the previous chapter. This is the program that we came up with for the coin flipping problem. Of course, it won't be exactly like yours, but it will probably be similar. To find out if yours is correct, run both yours and ours, and see if they operate in a similar manner. The location of the displayed output on the screen may be different, too. Anyway, here is our program:

```
CLS
RANDOMIZE TIMER / 3
LOCATE 5, 19
PRINT "HEADS"
LOCATE 5, 39
PRINT "TAILS"
LOCATE 6, 20
PRINT HEADS
LOCATE 6, 40
PRINT TAILS
FOR COIN = 1 TO 2500
    LET FLIP = INT(2 * RND) + 1
    IF FLIP = 1 THEN GOTO ISHEADS
    LET TAILS = TAILS + 1
    LOCATE 6, 40
    PRINT TAILS
    GOTO NEXTFLIP
ISHEADS:
    LET HEADS = HEADS + 1
    LOCATE 6, 20
    PRINT HEADS
NEXTFLIP:
NEXT COIN
END
```

When we came up with the problem, we hadn't yet developed the program for the solution, and discovered that it would have saved four lines to not print out the zeros initially. Oh, well. That's life.

Here is the program to draw the diagonal line of asterisks:

```
CLS
FOR STAR=1 TO 22
    LOCATE STAR, STAR
    PRINT"*"
NEXT STAR
```

```
END
```

To modify this program to get the next one, simply change line 3 to read:

```
    LOCATE STAR, 2 * STAR
```

In this chapter, we will reintroduce the PRINT command, but this time it has a powerful extension to it. In fact, it does so much that we won't even discuss all of its features here. That's because some of them have to do with string variables, which we will introduce later on. Let's get right into it. Here is an example program for you to type in:

```
CLS
FOR NUM = 1 TO 20000
    LOCATE 5,10
    PRINT NUM
NEXT NUM
END
```

Now run the program, and notice how it behaves. If you didn't catch it, run it a few more times. Do you notice that as the number grows larger (has more digits) that it "grows" to the right? This means that the numbers are "left justified". Run it again, and notice especially at the start that the fastest changing number moves to the right by one position as the number grows larger. Now change line 4 to read:

```
    PRINT USING "#####";NUM
```

Now run the program a few more times. Notice that now the number "grows" to the left? In other words, the ones position does not shift over as the number becomes larger. This is called "right justified", and is useful for columns of numbers, because the digits will line up.

What do those number signs do, and why are there five of them in the PRINT USING command? The number sign represents a digit in the output format. In other words, the USING part of the PRINT command tells the computer just how to print out our numeric variable. Since we have 5 #'s, we are telling the computer to reserve 5 spaces to print out digits. If we needed to print a larger number, we would need to add another number sign. What happens if there are more digits than number signs? Let's find out. Change line 4 to:

```
    PRINT USING "####";NUM
```

Now run the program. You won't notice a difference until the number 10000 is printed. What do you get? That's right, it prints out %10000. That percent sign is QBASIC's way of showing us that we have a formatting error. It does not stop the program in its tracks, though. To fix that little formatting problem, simply add another number sign or two.

It sounds simple so far, but let's try something else. Change the program to (Changes are in green):

```
CLS
FOR NUM = 1 TO 2000 STEP 0.1
    LOCATE 5, 10
    PRINT USING "####"; NUM
NEXT NUM
END
```

Now run the program. Does it do what you expected? *NO!* We expected it to count from 1 to 2000, incrementing by 0.1 each time, but all we get are whole numbers! Why? We did not specify a decimal point in our format, so QBASIC assumes that it goes at the end. How do we fix that? Change line 4 to:

```
    PRINT USING "####.#";NUM
```

Now run the program again. Much better, wouldn't you say? What was happening to the number when the decimal point was being cut off? Let's find out. Type this in the immediate window:

```
PRINT USING "#";1.4
```

Do you see what the answer is? The computer spits back a "1". I guess we could expect that. Now try this:

```
PRINT USING "#";1.5
```

***AH-HA!!*** Now the computer gave us a "2". This thing is pretty smart! It knows how to round off the answers! Let's try something else. Change the program to:

```
CLS
FOR NUM = 900 TO -900 STEP -1
    LOCATE 5,10
    PRINT USING "###"; NUM
NEXT NUM
END
```

and run the program. Do you notice our little formatting error symbol? Why is that? It's because we specified 3 digit positions. It works fine for the positive numbers, but when we get to the negative numbers, we must specify a space for the minus sign. Change line 4 to:

```
    PRINT USING "####";NUM
```

Now RUN the program again. Ah, that's better! But suppose we are doing some kind of engineering work, and we would like the sign to be printed even for a positive answer. What do we do then? Well, one of the specifiers for our format is the plus sign, +. To see it in action, change line 4 to:

```
    PRINT USING "+###";NUM
```

Now run it and watch closely. Did you notice that the program printed a plus sign for the positive numbers, and a minus sign for the negative numbers? If you missed them, run the program a few more times. The numbers fly by fast, so watch closely. Change the STEP size to -0.1 or -0.01 to slow it down if you have to.

Let's switch to accounting type problems. Suppose we have some huge numbers we want to print out. Let's start a new program to help us with these next few examples:

```
CLS
LET NUM = 1
MORE:
LOCATE 5, 10
PRINT USING "#######"; NUM
LET NUM = NUM * 1.01
IF NUM < 10000000 THEN GOTO MORE
END
```

Go ahead and run the program. It goes kind of fast. Notice our final value. Is it nine hundred thousand something, or nine million something? How do you clean it up to read it easier? Most people write extremely large number by putting commas in them. Did you know that QBASIC can do that, too? Here's how to specify it. Change line 4 to:

```
PRINT USING "#######,";NUM
```

Now run the program again. Oh,look! We get that format overflow signal again! Why? It's because the commas take up a couple of character positions! Once again, we just need to add a couple more #'s to fix it:

```
PRINT USING "#########,";NUM
```

Now run it again. Much better. Another fix would be to write the program line this way:

```
PRINT USING "#,###,###";NUM
```

Next problem. Suppose our variable NUM represents a dollar amount. Can we put a dollar sign in front of the number? Yes. Let's change line 4 once again:

```
PRINT USING "$#########";NUM
```

Run it again. Now you have a dollar sign off to the left of the number. That's nice, but what if we want it right next to the number? QBASIC has a provision for that. To make the dollar sign "float", you simply place two of them at the front of the format, like this:

```
PRINT USING "$$#########";NUM
```

Now run it again. Looks a little better, but can we combine two things together? We like the floating dollar sign, but it would also help to have the commas in there, too. Here is our next line:

```
PRINT USING "$$########,";NUM
```

Go ahead and run it. One more problem with this format. When you specify dollars, you usually will want to display some cents to go along with it. To do that, all you need to do is specify two digit positions after the decimal point, like such:

```
PRINT USING "$#########,.##";NUM
```

Run it again. Do you notice something interesting about that final value? Even though the cents part is zero, the computer still prints out the zero digits. When you use PRINT USING, any specified digits after the decimal point are always printed, even if they are zero. Type this in the immediate window:

```
PRINT USING "#.####";0
```

Notice how all of the digits are printed? It makes for a more uniform look.

Back to the Program Window. Now let's say you are writing a program that will print out on pre-printed checks. To prevent tampering, many programs will fill in unused positions to the left of the number with asterisks. How do we do that in QBASIC? Like this:

```
    PRINT USING "**#########,.##";NUM
```

Now run it and see what happens. Neat, huh? But suppose your check doesn't have a dollar sign printed on it? We want to have our asterisk filler, followed by our floating dollar sign, followed by our number, complete with comma seperators. You guessed it, here is our new line:

```
    PRINT USING "**$#########,.##";NUM
```

Run it one last time. Neat, huh?

Well, that's about all we have room for in this chapter. In the next chapter, we will put some of these PRINT USING formats to use, displaying a quadratic equation. But for next time, here is a program for you to try. It is a little more difficult, so it will be the only one we will give you. If you figure it out, make up some of your own.

Write a "cash register" program. Have the user input the price of a product, and the quantity of that product. Display a running total at all times. When the user enters a price of zero, that will indicate that there are no more items. When all items are entered, display the sub-total, calculate and display a 5% tax, and then display the total. If you feel adventurous, have the user enter the amount of cash tendered, and display the amount of change to be returned. For the really brave, have the computer calculate the number of $10, $5, $1 bills and quarters, dimes, nickels, and pennies to be returned as change.

See you next time!!

*Introduced in this chapter:*

*Keywords:* PRINT USING
*Concepts:* Formatting numerical output for easier reading and "accounting" type output.

# Chapter Twelve

## Output Formatting, Part Three

### Keyword: AND

For those of you who tried to write the program at the end of the last chapter with all of the bells and whistles, but found it nearly impossible, don't worry. It was quite a difficult program to write. It took us about 30 minutes to do it, and we have been programming in QUICKBASIC for over ten years! Enough whining, however. Let's get to the problem. Here is the basic essentials of the cash register program. It totals up the purchases, adds the tax, and then prints out the total.

```
LET SUBTTL = 0
NEXTITEM:
CLS
LOCATE 5, 40
PRINT "Running Total:";
PRINT USING "$$####.##"; SUBTTL
LOCATE 2, 2
PRINT "Enter Price:";
INPUT PRICE
IF PRICE = 0 THEN GOTO TOTALS
PRINT "Enter Quantity:";
INPUT QTY
IF QTY = 0 THEN GOTO NEXTITEM
LET SUBTTL = SUBTTL + (PRICE * QTY)
GOTO NEXTITEM
TOTALS:
CLS
PRINT
PRINT "Subtotal:";
PRINT USING "$$####.##"; SUBTTL
LET TAX = SUBTTL * .05
LET TAX = INT((100 * TAX) + .5) / 100
PRINT " Tax:";
PRINT USING "$$####.##"; TAX
PRINT
PRINT " TOTAL:";
TOTAL = SUBTTL + TAX
TOTAL = INT((100 * TOTAL) + .5) / 100
PRINT USING "$$####.##"; TOTAL
```

Notice how we rounded our tax and final total to the nearest penny (it's the green line). Keep that formula handy. It is a universal formula for rounding. It basically goes like so: Multiply your number by a desired amount so as to move the decimal point after the digit to be rounded.

Then add 0.5 to that amount, and take the integer portion. Next, divide by the same amount that you multiplied by earlier. Simple, don't you think?

If you stopped here with your program, you really didn't need to round off your final answer. However, we went on with the program, and with the computer doing all of that addition and subtraction, some inaccuracies will creep in. If you have forgotten why we need to do this, go back and re-read chapter six. Anyway, here is the second part of the program. Adding this segment asks the cashier to enter the amount of cash used to pay, and then displays the change due:

```
PRINT "Enter Cash Received";
INPUT CASH
CHANGE = CASH - SUBTTL - TAX
CHANGE = INT((100*CHANGE) +.5) / 100
PRINT "Change Due:";
PRINT USING "$$####.##";CHANGE
```

Now that wasn't too bad, was it? Next is the killer. Again, notice that we rounded to the nearest penny. That is because this next section has a separate program loop for each denomination of bill or coin to be returned. We have tested the program, and it worked well for us. However, with very large amounts of change due, there may be a sufficient number of "trips" through the loops to accumulate an error of greater than one cent. The advanced series will introduce a way to declare all of our variables as integers, and bypass that problem. It does create another problem, however. As integers, we would have to represent everything in pennies, and not dollars. Also, the computer stores integer numbers in a much more compact way than "regular" numbers, and as such, can only go to about 32,000. That would limit our total to $320. The way around that is to use a different type of integer. Anyway, here is our "How to hand out the change" segment:

```
MORETENS:
IF CHANGE < 10 THEN GOTO FIVES
CHANGE = CHANGE - 10
TENS = TENS + 1
GOTO MORETENS

FIVES:
IF CHANGE < 5 THEN GOTO ONES
CHANGE = CHANGE - 5
FIVES = FIVES + 1
GOTO FIVES

ONES:
IF CHANGE<1 THEN GOTO QUARTERS
CHANGE = CHANGE - 1
ONES = ONES +1
GOTO ONES

QUARTERS:
IF CHANGE < .25 THEN GOTO DIMES
CHANGE = CHANGE - .25
QUARTERS = QUARTERS + 1
GOTO QUARTERS
```

```
DIMES:
IF CHANGE < .1 THEN GOTO NICKELS
CHANGE = CHANGE - .1
DIMES = DIMES + 1
GOTO DIMES

NICKELS:
IF CHANGE < .05 THEN GOTO PENNIES
CHANGE = CHANGE -.05
NICKELS = NICKELS + 1
GOTO NICKELS

PENNIES:
IF CHANGE < .005 THEN GOTO NOCHANGE
CHANGE = CHANGE -.01
PENNIES = PENNIES + 1
GOTO PENNIES

NOCHANGE:
PRINT "Return:"
IF TENS = 0 THEN GOTO NOTENS
PRINT TENS;"Tens"

NOTENS:
IF FIVES = 0 THEN GOTO NOFIVES
PRINT FIVES;"Fives"

NOFIVES:
IF ONES = 0 THEN GOTO NOONES
PRINT ONES;"Ones"

NOONES:
IF QUARTERS = 0 THEN GOTO NOQUARTERS
PRINT QUARTERS;"Quarters"

NOQUARTERS:
IF DIMES = 0 THEN GOTO NODIMES
PRINT DIMES;"Dimes"

NODIMES:
IF NICKELS = 0 THEN GOTO NONICKELS
PRINT NICKELS;"Nickels"

NONICKELS:
IF PENNIES = 0 THEN GOTO NOPENNIES
PRINT PENNIES;"Pennies"

NOPENNIES:
IF ABS(CASH-TOTAL)>.01 THEN GOTO CHANGE
PRINT "No Change"
CHANGE:
END
```

Wow! Can you see why it took us a while? 117 lines (including blank ones)!

Let's start to develop our quadratic equation program. This does not actually solve the equation, it merely displays it accurately. Going back to those good old high school days, you may remember that a quadratic equation looks like $ax^2+bx+c=0$. Right off the bat, we have a small problem. The computer does not allow us to show the exponent 2 for the first element (easily - it can be done). Therefore, we will use QBASIC's form of exponentiation, the "**^**" symbol. Let's develop the program.

The program will operate as such: The variables a, b, and c shall be input from the user, one at a time. The user shall input 0 for all three values to indicate that the program should terminate. After all three variables are received, the program will proceed to output the equation in the form "ax^2+bx+c=0", adjusting the addition signs to subtraction if negative values are entered. If a value of 1 is entered, only the x component should be displayed. if a value of -1 is entered, only the -x component should be displayed. If a value of 0 is entered, the particular term must not be displayed at all.

Sounds simple enough. Let's get our program started by clearing the screen and asking the user to enter the required values. So that the user has some idea as to what is going on, we will print the equation in the generic form at the top of the screen, and then ask for the values.

```
CLS
TOP:
PRINT "ax^2+bx+c=0"
PRINT
PRINT
PRINT "Please enter a ";
INPUT A
PRINT "Please enter b ";
INPUT B
PRINT "Please enter c ";
INPUT C
```

Now we come to our first problem. How can we check a, b, and c all at the same time? It's simple. We will use QBASIC's *Boolean* (logic) operators. Boolean functions are used in a very specialized form of mathematics, particularly in binary math. Since a computer is a very fast binary device, Boolean functions are very common. There are two quite common Boolean operators that are used in QBASIC. they are **AND** and **OR**, and they do just what they sound like they do. Instead of trying to explain it, we will show you our line that does it. Go ahead and type this in:

```
IF A = 0 AND B = 0 AND C = 0 THEN END
```

It looks innocent enough. Simply stated, in an AND situation, all of the comparisons have to be true for the entire thing to be true. In an OR situation, at least one must be true (but more than one can be true) for the statement to be true. We will discuss these and other Boolean operators in a future chapter. Yes, there are a couple more. By the way, see how we can end the program in an IF...THEN statement? Anyway, let's worry about displaying the ax^2 element. For this, we need to take care of, and eliminate the special cases first. We'll start out with a=1. If so, we only print out "x^2". Here's how:

```
IF A <> 1 THEN GOTO ANOTONE
```

```
PRINT "x^2";
```

Since that part was taken care of, we need to jump around the rest of the ax^2 part:

```
GOTO GETB
```

Now we will handle a=-1:

```
ANOTONE:
IF A <> -1 THEN GOTO ANOTMINUS1
PRINT "-x^2";
GOTO GETB
```

And the a=0. For this, we don't print anything, so we jump right to the beginning of our b section.

```
IF A = 0 THEN GOTO GETB
```

Now all we have left is numbers to actually be printed. We'll use a PRINT USING function to handle it. In this case, if the number is positive, we don't need to add a plus sign, since it is the first element in the equation:

```
PRINT USING "###"; A;
PRINT " x^2";
```

Now that we have the first element out of the way, let's work on the "b" part of the equation. It will be quite similar to the "a" part, but with a few exceptions. Let's start with a value of 1. Here, we need to print out a "+x" term, like this:

```
GETB:
IF B <> 1 THEN GOTO BNOT1
PRINT " +x";
GOTO GETC
```

If b is -1, we need to print out a "-x" term, like so:

```
BNOT1:
IF B <> -1 THEN GOTO BNOTMINUS1
PRINT " -x";
GOTO GETC
```

If b=0, we need to skip around the entire "b" code. We could have just as easily placed these lines up higher, speeding up the program slightly for b=0. Here's the skip:

```
BNOTMINUS1:
IF B = 0 THEN GOTO GETC
```

At this point, all we have are actual coefficients that need to be printed. Again, we will use a PRINT USING command to print it, but we will use it slightly differently. In the first term, we did

not need to print a plus sign. For this term, we do. If you wanted to get really tricky, you could write the program to print the plus sign only if the "a" coefficient were non-zero. We didn't feel like doing it that way, so it saved us about two or three program lines. Try to figure it out for yourself. Here is our last lines for the "b" term:

```
PRINT USING "+###"; B;
PRINT " x";
```

We are going to stop here, since we covered a lot of ground in a short chapter. Meanwhile, see if you can figure out the rest of the program for yourself. If not, the next chapter will continue the program development. If so, see how closely your program is to ours. See you then!!

## *Introduced in This Chapter:*

*KeyWords:* AND

*Concepts:* Specialized output formatting, comparing multiple equalities in a single IF...THEN statement.

# Chapter Thirteen
## String Variables

To start off this chapter, we will begin by completing the program we started in the previous chapter. As you may recall, we were printing out a quadratic equation. We have already printed out the a and b terms, and only had the c term to do. First off, let's see if the c term is zero. If it is, we will skip around the whole process:

```
GETC:
IF C = 0 THEN GOTO FINISH
```

Now that we are this far, we need to test for an error condition. If a and b are 0 and c isn't, we have an inequality. Something like "5=0". This can't be, so we will print out an error message, and have the user enter another set of numbers, like so:

```
IF A = 0 AND B = 0 THEN GOTO OOPS
GOTO SHOWC
OOPS:
BEEP
PRINT "How can you tell me that"; C; "=0 is a legitimate statement?"
BEEP
GOTO TOP
```

Now that we have that out of the way, we only need to print out the c term with it's sign. We can do that with a PRINT USING command:

```
SHOWC:
PRINT USING "+###"; C;
```

Now we will print out the "=0" part:

```
FINISH:
PRINT "=0"
```

From here, we go back and do it all again:

```
GOTO TOP
```

For your benefit, here is the entire program listing:

```
CLS
TOP:
PRINT "ax^2+bx+c=0"
PRINT
PRINT
PRINT "Please enter a ";
INPUT A
PRINT "Please enter b ";
INPUT B
PRINT "Please enter c ";
INPUT C
IF A = 0 AND B = 0 AND C = 0 THEN END
IF A <> 1 THEN GOTO ANOTONE
PRINT "x^2";
GOTO GETB
ANOTONE:
IF A <> -1 THEN GOTO ANOTMINUS1
PRINT "-x^2";
GOTO GETB
ANOTMINUS1:
IF A = 0 THEN GOTO GETB
PRINT USING "###"; A;
PRINT " x^2";
GETB:
IF B <> 1 THEN GOTO BNOT1
PRINT " +x";
GOTO GETC

BNOT1:
IF B <> -1 THEN GOTO BNOTMINUS1
PRINT " -x";
GOTO GETC
BNOTMINUS1:
IF B = 0 THEN GOTO GETC

PRINT USING "+###"; B;
PRINT " x";

GETC:
IF C = 0 THEN GOTO FINISH

IF A = 0 AND B = 0 THEN GOTO OOPS
GOTO SHOWC
OOPS:
BEEP
PRINT "How can you tell me that"; C; "=0 is a legitimate statement?"
BEEP
GOTO TOP
SHOWC:
PRINT USING "+###"; C;
FINISH:
PRINT "=0"
GOTO TOP
```

If you feel ambitious, fix the problem where a=0 and b is positive; i.e. supress the plus sign in that instance.

Let's jump into something new! We are now going to learn about a new type of variable. Up to this point, we have been using letters to represent storage spaces for numbers. With the addition of a symbol to the variable descriptor, it will have the capabilities of storing not numbers, but letters. They are known as string variables. You are already familiar with string constants. In a command like PRINT "I Am Learning To Program A Computer", the stuff inside the quotes is a string constant. That is because the same thing is printed out every time the program is run. To get the idea of string variables, let's review briefly numeric variables. Type in the following in immediate window:

```
CLS
LET A=45
PRINT A
```

The computer responds with 45. The number printed is the current value of the numeric variable a. Now type this in ***exactly*** as shown:

```
LET A$="This Is A String Variable"
PRINT A$
```

Notice how the computer responds? Now type this in:

```
LET A$="Hi."
PRINT A$
```

Notice that when you assign something to a string variable that is already occupied, the stuff that was in it gets cleared out. Otherwise, you would have seen:

```
Hi.s Is A String Variable
```

as a response. What if you want to clear out a string variable and not put anything in it? You assign a null string to it. What is a null string? It is a string that contains no characters at all. How do you do it in QBASIC? Like this:

```
LET A$=""
```

Two double-quotation marks right next to each other signifies a null string.

Let's put some of this together now. Here we will write a program that accepts a list of students and their test scores, and then sorts the list from highest to lowest, and then prints out the students and their scores in descending test-score-order. The first thing we will do is clear off the screen. From the Program window, enter:

```
CLS
```

Now, since we are sorting a list of test scores, we will need to use subscripted variables. This next program line will show us two things - how to dimension more than one variable at a time, and how to dimension a string variable:

```
DIM NAM$(100),SCORE(100)
```

At this point, we will have the user tell the program how many students there are:

```
GETCOUNT:
PRINT"Enter Number of Students";
INPUT NUMBER
```

Since we only allowed 100 spaces for student names and scores, let's make sure that we don't ask for more than 100 students. Using input range checking would go something like:

```
IF NUMBER<100 AND NUMBER>0 THEN GOTO GETSTUDENTS
PRINT" That's Too Many Students!!! Try Again"
GOTO GETCOUNT
```

Now we need to set up a loop to get all of our information entered into the program. Here is how we initialize the loop:

```
GETSTUDENTS:
FOR N=1 TO NUMBER
```

So that the person entering the information can keep track of where they are, we will print out the loop variable in our prompt like so:

```
    PRINT "Enter Student #";N;" Name";
    INPUT NAM$(N)
```

Now we will ask for the test score for that particular student, and continue until all students are entered:

```
    PRINT"Enter Test Score For ";NAM$(N);
    INPUT SCORE(N)
NEXT N
```

Now we need to sort the test scores. We will use the same algorithm (procedure) that we used in chapters 8 and 9. Here is the beginning of the sort segment:

```
FOR OUTER=1 TO NUMBER-1
    FOR INNER=OUTER+1 TO NUMBER
        IF SCORE(OUTER)>=SCORE(INNER) THEN GOTO NOSWAP
        SWAP SCORE(OUTER),SCORE(INNER)
```

When we swap the scores around, we have to remember to swap the corresponding name with it!

```
        SWAP NAM$(OUTER),NAM$(INNER)
```

Now we can finish the sort algorithm:

```
NOSWAP:
    NEXT INNER
NEXT OUTER
```

Now we will clear the screen and print column titles:

```
CLS
PRINT "Name","Score"
PRINT
```

That last line simply puts a blank line between the column titles and the first piece of data. Now we will use a final loop to print out the sorted list of names and scores:

```
FOR N=1 TO NUMBER
    PRINT NAM$(N),SCORE(N)
NEXT N
END
```

Before you run the program, write down a list of about 9 or 10 names, and associated scores. For speed, use only first names. It types faster that way. Make sure that the test scores are not in any order. Run the program, and enter the names and numbers as you have them written down. When the computer spits out the results, check to make sure that the scores didn't change for the students!

For the next couple of chapters, we will introduce a whole bunch of functions that are connected with string variables. But for now, write a program that accepts a list of words, places the words in alphabetical order, and prints out the list.

***HINT:*** Alphabetizing a list is done by sorting it. When you run the program, use either all upper case letters or all lower case letters. Don't mix and match! The computer considers a lower case A to come after an upper case Z!

Have fun! See you next chapter!!

## *Introduced In This Chapter:*

*Concept:* String Variables.

# Chapter Fourteen

## String Functions

### Keywords: LEN, LEFT$, RIGHT$, MID$

Did you have any luck with your alphabetizing program? If not, don't despair; we will go through the development of the program in this chapter. Along the way, we will show you another trick or two. Let's get right to it.
To start off, we want our program to clear any clutter off the screen:

```
CLS
```

We will then ask the user for the number of words they will want to sort:

```
PRINT"How many words to sort";
INPUT NUMBER
```

Wait a minute! Have we forgotten something? If we are going to store and sort some variables, won't we need to dimension a subscripted variable? Yes. Here is our first trick! We will only set aside as much variable space as we need! How do we do that? QBASIC allows us to use a variable as the limit indicator in a dimension statement. Here is how our DIM statement will look:

```
DIM WORD$(NUMBER)
```

Now we simply have to enter in all of our data. We will use a method similar to the test scores program in last chapter, that is, printing out the number of each word as it is entered. Here we go:

```
FOR N=1 TO NUMBER
    PRINT "Enter Word #"N;
    INPUT WORD$(N)
NEXT N
```

Now we have all of our words entered into the subscripted variable, and we are using memory in the most efficient way possible at the same time! Incidently, if you enter a number of words that exceeds the available memory of the computer, you will get an *Out of Memory* message. Use a smaller number of words! We are not trying to store the entire dictionary here! The actual number at which the error message occurs is determined by many factors, including how QBASIC was loaded, and any other programs that are taking up memory. It is not our intention to explain the memory management capabilities of QBASIC, so don't worry about it.

Back to our program. Now that we have the words in our array, we simply need to sort them. To do this, we will use the exact same algorithm that we have been using to sort numbers. This time, instead of comparing and swapping numbers, we will be comparing and swapping words. Here is our next piece of program code:

```
FOR OUTER =1 TO NUMBER-1
    FOR INNER=OUTER+1 TO NUMBER
        IF WORD$(INNER)<WORD$(OUTER)THEN SWAP WORD$(INNER),WORD$(OUTER)
    NEXT INNER
NEXT OUTER
```

Let's stop here and discuss how the computer "sees" words. obviously, it knows that "A" comes before "B". But what happens when it sees "APE" and "BUG"? It simply looks at the first letter, and sees again that "A" comes before "B". Let's change it around. What about "APE" and "ABE"? First, the computer compares the first letter of the two words. If they are the same (as they are here), it goes to the second letter of each word. Here it sees that "B" comes before "P". How about something tricky like "SMITH" and "SMITHE"? The computer actually looks at the first five letters, and sees no difference. When it goes to the sixth letter, it sees that it runs out on one of the words. The computer then knows that the one with fewer letters is less than the one with more letters. In the case of "SMITH" and "SMITH", the computer considers them to be equal. They are, aren't they? Smart computer!
In any case, we may now simply print out our sorted list:

```
FOR N=1 TO NUMBER
    PRINT WORD$(N)
NEXT N
END
```

The easy way to test the program is to run it, and tell it that you have 26 words. As it asks for them, just type in one letter of the alphabet at a time. To make it easy to scramble the letters, just type them in as they appear on the keyboard:

```
QWERTYUIOPASDFGHJKLZXCVBNM
```

You will see the sorted list printed out when the program finishes. Well, the first four letters will scroll off the top of the screen, but you get the idea.

What kind of fun commands can we use with string variables? Let's look at just a few of the things that QBASIC will let us do! First, switch into the immediate window. Enter:

```
LET A$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

Now we have A$ which contains the alphabet. To verify that it is in there correctly, type in:

```
PRINT A$
```

Make sure you get the alphabet back. We know that there are 26 letters in the alphabet, but does the computer know that? We can ask it! Type in:

```
PRINT LEN(A$)
```

And the computer gives us the number 26! The LEN(string) function return the length of a string! Type in:

```
PRINT LEN("LETTERS")
```

and the computer responds with 7. That's how many letters are in the word "letters". We can

also look at parts of the string. Type in:

```
PRINT LEFT$(A$,5)
```

and the computer will respond with the first five letters of the alphabet! This command will take the leftmost characters of the specified string (in our case, A$), and return the requested number of characters. If we request more characters than there are in the string, we just get the whole thing back. QBASIC will not add any spare characters to it. Now let's type this:

```
PRINT RIGHT$(A$,5)
```

Now we get the last five letters of the alphabet! This works almost identical to the LEFT$ function, except that it takes letters from the right side of the string instead of the left side. What if we wanted something in the middle? Type this in:

```
PRINT MID$(A$,10,1)
```

The computer responds with a "J". If you were counting, you will discover that this is the tenth letter of the alphabet. Can you figure what the 1 does? Let's find out by changing it:

```
PRINT MID$(A$,10,3)
```

Notice that the computer responds with "JKL". It tells us how many letters to return. If we ask for more letters than there are remaining, it will give us from the specified letter to the end. If we ask it for a starting position that is greater than the length of the string, the computer will give us a null string for an answer.

Here's a fun thing to try: Let's write a program that has the user type in a word or sentence, and we will print out the letters backwards! First, switch back to the program area, and clear out any program you may have. The first thing our program will do is clear the screen. After that, we will have the user enter the string:

```
CLS
PRINT "Enter the sentence to print out backwards:";
INPUT A$
```

From here, all we have to do is look at one character at a time, starting from the end, and print it out. Have we seen an easy way to look at one character in the middle of the string? Sure! we used the MID$ function earlier to look at the tenth letter of the alphabet! We aren't stuck using numeric constants in our functions, we can use variables! Getting back to the program, the first thing we need to do is figure out how many characters are in our string. We can do that with the LEN function. In fact we will use this function as our starting loop variable. We will need to step backwards through the string variable, so our STEP will be -1. Our ending point will be the first character in the string. Using this information, our FOR statement will look like this:

```
FOR LETTER = LEN(A$) TO 1 STEP -1
```

Now we need to display each character at a time. Don't forget to suppress the automatic carriage return on the print statement!

```
    PRINT MID$(A$,LETTER,1);
NEXT LETTER
```

Since we suppressed the carriage return, and we have completed printing out our backwards sentence, we need to put in a carriage return. Do you remember how?

```
PRINT
END
```

Now go ahead and run the program using this for input:

```
!SKROW MARGORP RUOY !SNOITALUTARGNOC
```

If you typed in everything correctly, you will see a neat little message.

Your project to try before the next chapter is a bit involved. The program will sort a list of names. Ask the user for the number of names to be entered. Then enter the names in first-name-space-last-name order, and then sort them by last name. To make the program easier, you are allowed to print out the names in last name - first name order.

***HINT:*** In our solution, we will convert the name that the user enters into last name - first name order immediately after he types it in. It will be stored in the array last name first. We will also show you an option for printing out the sorted list in first name first order. You may try it on your own first if you are feeling bold!
Good Luck! See you next chapter!

## *Introduced In This Chapter:*

*Keywords:* LEN(), LEFT$(), RIGHT$(), MID$()
*Concepts:* Dimensioning subscripted variables with a variable, rules for comparing strings, disassembling a string variable.

# Chapter Fifteen

## More String Functions

### KeyWords: DATE$, TIME$, STR$, VAL

We have a lot of ground to cover in this chapter, so we will dig ourselves right into solving the problem presented at the end of the last chapter! We will start off the program just like we did the last few. We'll clear the screen, ask the user how many items are to be entered, and set up our subscripted variable:

```
CLS
PRINT "Enter the number of names to sort:";
INPUT NUMBER
DIM NAM$(NUMBER)
```

Now we will have the user enter the names using a loop:

```
FOR N = 1 TO NUMBER
    PRINT "Enter Name #"; N;
    INPUT N$
```

Here is where we run into our first challenge! The person running the program will be entering the data not only in first-name last-name order, but possibly first-name middle-initial last-name order. We want to sort our list by last name! What do we do? First, we need to search our string containing the name for a space one character at a time, starting from the back. We are going to assume that a "Jr." will not be used for any name. Then we will split the name into two parts, using the LEFT$ and RIGHT$ functions. We will then reconstruct the name in last-name first-name order using a feature of QBASIC called ***Concatenation.*** This is simply connecting two string variables together to make one longer string variable. Here is an example. You may type them in (in the immediate window), if you wish. Suppose we have a program containing these lines:

```
LET A$ = "ABCDE"
LET B$ = "FGHIJ"
LET C$ = A$ + B$
PRINT C$
```

What do you suppose the result would be? If you went ahead and typed them in (in the immediate window), you know that the answer is "ABCDEFGHIJ". What if we had:

```
LET C$ = A$ + "SPACE" + B$
PRINT C$
```

Then the answer would be "ABCDESPACEFGHIJ". Using what we know, and making the code as compact as possible, here is how we converted the name entered to last-name first-name order:

```
    FOR CHARACTER = LEN(N$) TO 1 STEP -1
        IF MID$(N$, CHARACTER, 1) = " " THEN GOTO SKIPREST
    NEXT CHARACTER
SKIPREST:
    LET NAM$(N) = RIGHT$(N$, LEN(N$) - CHARACTER) + " " + LEFT$(N$, CHARACTER - 1)
```

Notice that we needed to use a nested loop to search for our space character. Also note the adjustments we had to make in the arithmetic concerning character placement to avoid having a space at the end of our new string. We also needed to add a space between last name and first name, since they were not actually entered by the user. At this point in the program, the name has been assigned to the subscripted variable, so we need to finish off the data-entering loop:

```
NEXT N
```

At this point, we simply sort the list of names. Remember, they are stored last name first. We will use the exact same method that we used in the last chapter. Here it is:

```
FOR OUTER = 1 TO NUMBER - 1
    FOR INNER = OUTER TO NUMBER
        IF NAM$(INNER) < NAM$(OUTER) THEN SWAP NAM$(INNER), NAM$(OUTER)
    NEXT INNER
NEXT OUTER
```

Now we simply have to print out our sorted list:

```
FOR N = 1 TO NUMBER
    PRINT NAM$(N)
NEXT N
END
```

Simple, wasn't it? It is when you get it down to bits and pieces! What we have in this program is three parts. The first part gets the data into the computer, and does some modifying as we enter it. Next, we sorted the list. Then, we printed out the data.

To print out the data in the form that it was entered in, we need to flop the name back around. You might think that we do it exactly the same way as when we entered the data. This would work for the first-name last-name method, but for first-middle-last, it would put the middle name at the front! In essence, we need to undo what we did earlier by doing it backwards. In other words, we must begin searching for the space character from the front of the string! Here are the lines required to do it:

```
...
FOR OUTER=1 TO NUMBER-1
    FOR INNER=OUTER TO NUMBER
        IF NAM$(INNER)<NAM$(OUTER) THEN SWAP NAM$(INNER), NAM$(OUTER)
    NEXT INNER
NEXT OUTER


FOR N=1 TO NUMBER
    FOR CHAR=1 TO LEN(NAM$(N))
        IF MID$(NAM$(N),CHAR,1)=" " THEN GOTO SKIPREST2
    NEXT CHAR
SKIPREST2:
    PRINT RIGHT$(NAM$(N),LEN(NAM$(N))-CHAR); " ";LEFT$(NAM$(N),CHAR-1)
NEXT N
END
```

Notice that the algorithm is nearly identical to the earlier routine, with the only difference being that we started the search from the front of the string instead of the back. The difference is in the inner FOR...NEXT loop.

On to bigger and better things! First, we will tell you about 2 special string variables. These variables are preassigned. Switch to the Immediate window, and type in:

```
PRINT DATE$
```

The computer responds with what it thinks is the current date. If it is incorrect, you may change it by typing in:

```
LET DATE$ = "03/01/97"
```

or whatever today's date happens to be. The other variable is similar. Type this in:

```
PRINT TIME$
```

The computer responds with what is thinks the current time is. The computer tells time in a 24-hour format. In other words, 2:45 pm is displayed as "14:45:00". As with the date, you can set the computer's time with this command:

```
LET TIME$ = "19:30:00"
```

if it is 7:30 pm.

Here are two interesting functions that convert a number to a string and back again. Type this in:

```
LET X = 6 * 8
LET B$ = "Is The Answer"
LET A$ = STR$(X) + " " + B$
PRINT A$
```

The STR$ function turns a number into a string! Can we go the other way? Can we have a string variable that is a number and convert it to a numeric variable? Sure! Type this in:

```
LET A$="12345"
LET B$ = "16 Is A Perfect Square"
LET C$ = "There Are 26 Letters In The Alphabet"
LET D$ = "This line contains no numbers"
PRINT VAL(A$)
```

The computer responds with the number `12345`, as expected. What about the other three string variables? How do they behave? Let's find out:

```
PRINT VAL(B$)
```

This only gives you a 16 for an answer. That's because the computer sees the 16, and then sees the space character. As soon as it sees a non-digit character, it quits looking. On to the next one:

```
PRINT VAL(C$)
```

Even though there is a number 26 in the string, the computer gives us a 0 result. That is because the first character in the string is not a digit, so the computer stops looking any further.

```
PRINT VAL(D$)
```

Of course, there are no digits at all here, so we get a zero response again.

Since the next chapter is the last one in the series, here is your last project program to try. We will attempt to incorporate a lot of what we have learned into this program.

Here is the final test: Have the user enter a number between 5 and 20. Generate the requested number of random integers in the range of 1 to 50. The output will consist of three column: The first column will be the unsorted list of random numbers. The second column will be the list of numbers sorted as numeric variables. The third column will be the list of numbers sorted as strings. The computer must emit a beep after each column has been printed. The first column must be printed out as each number is generated. The second column must be printed after sorting them as numbers. The third column will then be printed after sorting them as strings.

***HINT***: Be sure to brush up on the LOCATE command in chapter ten. After you sort the numbers numerically, you may convert them to strings without having to "unsort" them. They'll just get resorted anyway. Good luck with this review, and we'll see you for our final chapter!

## *Introduced In This Chapter:*

*Keywords:* DATE$, TIME$, STR$, VAL

*Concepts:* Converting numeric variables to string variables, converting string variables to numeric variables.

# Chapter Sixteen
## Putting It All Together

This is our last time together in this series, so by now (hopefully) this program should have been just a little bit easier. If you were smart, you might have even taken some bits and pieces of previous programs and glued them in! Enough chat, let's develop our multiple-sort program.

First, we will seed the random number generator and clear the screen:

```
RANDOMIZE TIMER / 3
CLS
```

Now we will ask for the number of digits to generate, and check that the response was within 5 and 20:

```
getcount:
PRINT "Enter Number of Random Digits to Generate, between 5 and 20 ";
INPUT NUMS
IF NUMS < 5 OR NUMS > 20 THEN GOTO getcount
```

Next, we will allocate the storage space needed for both the numeric and string versions of the random numbers:

```
DIM NUMBER(NUMS), NUMBER$(NUMS)
```

Let's clear up the screen and print out our first column heading:

```
CLS
LOCATE 1, 2
PRINT "Unsorted List"
```

Now we will generate our array of random numbers, and print them out as we generate them. We can do this all within the same loop. We will then beep the speaker when we are done, just like we were asked:

```
FOR N = 1 TO NUMS
    NUMBER(N) = INT(50 * RND) + 1
    LOCATE N + 2, 2
    PRINT NUMBER(N)
NEXT N
BEEP
```

First, we will sort the numeric array (as numbers):

```
FOR OUTER = 1 TO NUMS - 1
    FOR INNER = OUTER TO NUMS
        IF NUMBER(INNER) < NUMBER(OUTER) THEN SWAP NUMBER(INNER), NUMBER
(OUTER)
    NEXT INNER
NEXT OUTER
```

Next. we will print out our second column heading and the sorted list, then beep again:

```
LOCATE 1, 26
PRINT "Sorted As Numbers"
FOR N = 1 TO NUMS
    LOCATE N + 2, 26
    PRINT NUMBER(N)
NEXT N
BEEP
```

Now to sort them as strings, we first need to *convert* them to strings:

```
FOR N = 1 TO NUMS
    NUMBER$(N) = STR$(NUMBER(N))
NEXT N
```

Now we sort the string array, the same as the numeric array:

```
FOR OUTER = 1 TO NUMS - 1
    FOR INNER = OUTER TO NUMS
        IF NUMBER$(INNER) < NUMBER$(OUTER) THEN SWAP NUMBER$(INNER), NUMBER
$(OUTER)
    NEXT INNER
NEXT OUTER
```

And, just like the others, we will display the column title, and the sorted string array, then beep:

```
LOCATE 1, 52
PRINT "Sorted as Characters"
FOR N = 1 TO NUMS
    LOCATE N + 2, 52
    PRINT NUMBER$(N)
NEXT N
BEEP
```

We're done:

```
END
```

See how simple it can be when you break it down into smaller sections? That leads us to the next point. We are only using a small amount of the power of QBASIC. QBASIC allows us to write small programs like this and incorporate it into much larger programs! Not only is the program more organized, but it is also easier to debug (find and fix mistakes), and has the added benefit that if you use a certain function (like sorting) many times, you only need to write the sort routine once!

All of the programming we have done in this series is referred to as **top-down programming** - In other words, the program starts at the top, and just goes on down, doing what it needs to do, until it hits the end at the bottom. There is very little jumping around in the program. The type of programming technique described in the previous paragraph is called **structured programming**, and is the preferred way to write programs. In fact, in many of the more advanced languages, it is the only way that can be used!

QBASIC also has a very robust set of graphics commands and functions that we haven't even touch upon (or hinted at)! It is great for creating your own graphs, or you can use it for making your own graphics-based games. In fact, a train simulator program I wrote was originally developed in QBASIC! I had to move it over to QUICKBASIC for technical reasons to make it work (I usually write programs using multiple source files, which QBASIC does not allow, among other things), but it only uses regular QBASIC graphics functions! *(Some of the other functions use assembly language to make them work).*

Happy Programming!