Introduction

First of all, I am not original the writer of this material and this is the second edition. I came across it in a text file on my computer (as well as my Batch Programming material) a very very long time ago, so I decided to format it, edit it, convert it into an e-book and then publish it for free. I distributed this material amongst students (along with my Batch Programming material) in my class back in my diploma days because we had a course in BASIC, and the material proved itself to be very useful!

QBASIC is a very simple language to pick up, and yet it can accomplish a great deal. It is also important it a form of introduction to programming (honestly speaking, the first programming language I learned was BASIC!). It's simple to create games, business applications, simple databases, and graphics. The best aspect of the language is its close resemblance to English.

I hope that this little tutorial will introduce the basic concepts of programming to get you started with computer programming altogether. Even if you already know programming, this material could still be of interest to you, because I myself have learned a lot of new stuffs from this material!

Happy coding!

Kheme

About Me

My name is Okiemute Omuta, but I'm very well known as Kheme. I am a Nigerian freelance website designer and developer specialized in PHP & MySQL. I also have a good understanding of JavaScript, DHTML and CSS. I also have a considerable applicable knowledge of Flash, Action Scripts, AJAX, C/C++ and Oracle SQL.

I have gathered experience in designing websites commercially using PHP, MySQL, (X)HTML, CSS, JavaScript and DHTML since 2003. Hence, I am able to develop both static and dynamic websites, from small scale websites through medium scale to medium-large scale websites.

Besides developing websites, I run a free website design tutorials website where I have website design related lessons. The name of the website is HTML Online (www.htmlonline.tk); I also have a few free e-books on that website, such as this one you're reading right now.

If you noticed any spelling or typographical errors, or you just like my tutorials, or you have problems or questions to ask, feel free to email me at Kheme@htmlonline.tk or visit my website at www.khemeonline.tk. Also, I consider it consolation for my efforts when my readers e-mail in to say a thing or two about my e-books, or simply just to say "Thanks!" :)

Feel free to share this e-book with you friends and those you feel could benefit from it.

Thanks for downloading one of my e-books.

Kheme

Table of Content

Variables

A variable, simply defined, is a name which can contain a value. Programming involves giving values to these names and presenting them in some form to the user. A variable has a type which is defined by the kind of value it holds. If the variable holds a number, it may be an integer, floating decimal, long integer, or imaginary. If the variable holds symbols or text, it may be a character variable or a string variable. These are terms you will become accustomed to as you continue programming. Here are some examples of values a variable might contain:

```
STRING "hello, this is a string"
INTEGER 5
LONG 92883
SINGLE 39.2932
DOUBLE 983288.18
```

Variable Declaration

The first is a “string”. Strings contain text. The last four are number types. But the computer does not know what kind of value or data you are trying to give a variable unless you tell it! There are two methods of telling the computer what kind of variable you are using, by explicitly declare the variable as a type or by using predefined QBASIC variable symbols.

Explicit declaration is done by using the `DIM` statement. Say you wanted to make a variable called number which would contain an integer, you would do it like this:

```
DIM number AS INTEGER
```

Then you would use that variable as an integer. The word DIM actually originates from the word Dimension, but you won't see why until we discuss arrays.

Declaring variables using variable symbols is done by putting a symbol after the variable name which is defined as representing that data type. QBASIC has a set of symbols which represent each variable data type:

- $ = String
- % = Integer
- & = Long
- ! = Single
- # = Double

Appending one of these symbols to the name of a variable when you use it in the program tells the computer that you are using it as that data type.

This is actually a difficult concept to grasp for newcomers to programming. The most common error in QBASIC is the infamous “Type Mismatch” which you will see a lot! This means that you are trying to put a value into a variable of the wrong data type! You might be trying to put the letters "hi there" into an integer variable. If you don't define the type of the variable, then QBASIC assumes it is of the “Single” type, which can often yield unexpected results! I personally prefer to use the type symbols after variable names, but come explicitly declare them usually at the head of their programs.

Interacting with the Computer

You know what a variable is and how to control them, so it's time you learned some programming! QBASIC, like all other languages, is set up using pre-defined statements according to the syntax specified for that statement. It may be helpful to look in the help index to learn a statement but I've heard much complaint's that the help index is too hard! Indeed, it is too hard for new programmers, but as you learn more and more statements and their syntaxes, you'll become accustomed to the index and use it as a casual reference!

Writing to Screen

Let's make a program that prints some text on the screen. Type QBASIC at the DOS prompt and enter the following program:

```
CLS

PRINT "This text will appear on the screen"

END
```

Note: If you don't have QBASIC installed on your computer, you can download it free from http://www.htmlonline.tk/ebooks/qbasic.php

The first statement, `CLS` stands for "clear screen." It erases whatever was on the screen before it was executed. `PRINT` simply displays its argument to the screen at the current text cursor location. The argument in this case is the text enclosed in quotes. `PRINT` displays text within quotes directly, or it can display the value of a variable, like this:

```
CLS

a% = 50

b% = 100

PRINT "The value of a is "; a%; " and the value of b is "; b%

END
```

This will yield the output:

The value of a is 50 and the value of b is 100

The semicolons indicate that the next time something is printed it should be right after where the last PRINT statement left off. Remember that PRINT prints literally what is inside the quotes, and the value of the variable which is NOT in quotes. So in this example, a% & b% are integers containing values and their values are printed using the PRINT statement.

Input from Keyboard

Let's say you want to interact with the user now, you'll need to learn a statement called INPUT. INPUT displays a prompt and assigns what the user types in to a variable. Try these lines below:

```
CLS

INPUT "What is your name? ", yourName$

INPUT "How old are you? ", age%

PRINT "So, "; yourName$; ", you are "; age%; " years old.
That's interesting."

END
```

This first asks the user for their name and assigns it to the string variable “yourName$”. Then the age is requested and the result is printed in a sentence. So what happens if you input “I DON'T KNOW” at the age prompt, what do you think will happen? You'd get a weird message that says **REDO FROM START**. Why? This is because the program is trying to assign a string (text) to an integer (number) type, and this makes no sense so the user is asked to do it over again!

Condition Statement

Another cornerstone of programming is the conditional test. Basically, the program tests if a condition is true, and if it is, it does something. It looks like English so it's not as hard as it sounds.

```
CLS
PRINT "1. Say hello"      ' option 1
PRINT "2. Say nice tie"   ' option 2
INPUT "Enter your selection ", selection%
  IF selection% = 1 THEN PRINT "hello"
  IF selection% = 2 THEN PRINT "nice tie"
END
```

The user is given a set of options, and then they input a value which is assigned to the variable “selection%”. The value of selection% is then tested, and code is executed based on the value. If the user pressed 1, it prints hello, but if they pressed 2, it prints nice tie. Also notice the text after the single quote (') in the code. These are remark statements. Anything printed after a ' on a line does not affect the outcome of the program.

Back to the actual code; what if the user doesn't input 1 or 2? What if they input 328? This must be taken into account if you want to write a “smart” program. You usually can't assume that the user has half a brain, so if they do something wrong, you can't screw up the program! So the ELSE statement comes into play. The logic goes like this: IF the condition is true, THEN do something, but if the condition is anything ELSE, then do something else. You follow? The ELSE statement is used with IF...THEN to test if a condition is anything else, hence it is commonly called the IF THEN ELSE statement!

```
CLS

INPUT "Press 1 if you want some pizza.", number%

IF number% = 1 THEN PRINT "Here's your pizza" ELSE PRINT "You
don't get pizza"

END
```

That's a fairly simple example, but real life things will be much more complex! So let’s try a "real life" program! QBASIC is capable of fairly sophisticated math, so let’s put some of it to use.

Say your Algebra teacher tells you to find the areas of the circles with the following radiuses (or radii, whatever), and he gives you a sheet with hundreds of radii. You decide to boot up your computer and write the following program:

```
CLS

pi! = 3.1415

INPUT "What is the radius of the circle? ", radius!

area! = pi! * radius! ^ 2

PRINT "The area of the circle is ", area!

END
```

First, we're defining the variable pi. It's a single number, which means that it can be a fairly large number with some decimal places. The exclamation mark tells QBASIC that pi is of the single type. Next, the user is prompted for the radius of their circle, then the area is calculated. The star sign (*) means "times" and the carrot sign (^) means "to the power of" so, radius! ^ 2 means "radius to the power of 2" or simply "radius squared". This could also be written as pi! * radius! * radius!

Loop Structures

But there's a big problem with this program; the teacher gave you a sheet with hundreds of radii and so for every radius, you must run the program over again! This is not practical in anyway! But if we had some kind of loop until we wanted to quit, that just kept on repeating over and over, it would be much more useful! QBASIC uses Loop Structures for this. They start with a `DO` statement and end with a statement `LOOP`. You can `LOOP UNTIL` or `WHILE` or `DO UNTIL` or `WHILE` a condition is true. Another option is to break out of the loop manually as soon as a condition is true. Now let's adjust the previous code:

```
CLS

pi! = 3.1415

DO      ' Begin the loop here

 INPUT "What is the radius of the circle? (-1 to end) ",
radius!

 IF radius! = -1 THEN EXIT DO

 area! = pi! * radius! ^ 2

 PRINT "The area of the circle is ", area!

 PRINT

LOOP    ' End the loop here

END
```

Now we can end the program by entering -1 as the radius. The program checks the radius after the user inputs it and checks if it is -1. If it is, it exits the loop or else if it isn't, then it just keeps going its merry way. The PRINT with no arguments prints a blanks line so that we can separate our answers. So try the codes above and see for yourself!

Output Formatting

Say you want to print something in a certain pre-defined format, say you want to print a series of digits with only 2 places after the decimal point and a dollar sign before the first digit. Doing this requires the PRINT USING statement, which is very handy in business applications. The PRINT USING statement accepts two types of arguments. The first is a string which has already been defined. This is a special type of string, in that it contains format specifiers, which specify the format of the variables passed as the other arguments. Confused? You won't be. Here's a quick list of the most common format specifiers:

- ### digits
- & Prints an entire string
- \ \ Prints a string fit within the backslashes. Any thing longer is truncated
- $$ Puts a dollar sign to the left of a number
- . Prints a decimal point
- , Prints a comma every third digit to the left of the decimal point

And these can be combined in a format string to make a user defined way to print something. So "$$#,###.##" would print a number with a dollar sign to the left of it and if the number has more than two decimal places, it is truncated to two. If it is more than four digits long to the left of the decimal place, it will also be truncated to fit. To use a PRINT USING statement, you must first define the format string containing the format

specifiers, then the name of the format string, and variable values to fill the places defined in the format string. Here's an example to try:

```
CLS     ' get user input

INPUT "Enter item name: ", itemname$

INPUT "How many items?: ", numitems%

INPUT "What does one cost?: ", itemcost!

CLS     ' display inputs

format$ = "\            \   #,###      $$#,###.##
$$#,###,###.##"

    PRINT "Item Name        Quantity   Cost         Total Cost
   "

    PRINT "--------------   --------   ----------   ----------
----"

totalcost! = numitems% * itemcost!

PRINT USING format$; itemname$; numitems%; itemcost!;
totalcost!

END
```

First, we get the item name, number of items, and cost per item from the user. Then we clear the screen and define the format string to be used. It contains a static length string (text that will be truncated if it is too long), up to 4 digits for the quantity, 4 digits & two decimals for the item cost, and 7 digits & two decimals for the total cost. We then print out some column headers so we know what each value will represent, and some nice lines to go under the column headers. Then the total cost is calculated by multiplying the number of items by the item cost. And finally, the four variables' values are displayed under the column headers using the `PRINT USING` statement.

More Advanced Data Manipulation

There are numerous methods to manipulate data and present it to the user in QBASIC. One is called “array”. An array is a variable which can contain more than one value. For example, you might have an array called “a” and you can assign data to the members of that array. There might be a value for a(1) and a different value for a(6).

Arrays

Before an array can be used, it MUST be declared first! Arrays are declared with the `DIM` statement mentioned in the Variables lesson. Here is an example of an array declaration:

```
DIM a(1 TO 100) AS INTEGER
```

Now there are 100 different values that can be assigned to the array a, and they must all be integers. The array above could also be declared using the symbol % for integer, like this:

```
DIM a%(1 TO 100)
```

We call the different values for the array “members” of the array. The array “a” has 100 members. Array members can be assigned values by using a subscript number (within parentheses) after the array name, like this:

```
a%(1) = 10
a%(2) = 29
a%(3) = 39
```

and so on. Now you're probably wondering why the statement for declare is `DIM`. This comes from a term used in earlier programming languages

that means dimension. That still doesn't answer the question “why not use the statement DECLARE?” Well, an array can have more than one dimension. Arrays with multiple dimensions have y members in the second dimension for every x member of the first dimension:

```
DIM array(1 TO x, 1 TO y) AS INTEGER
```

So if the actual declaration looked like this:

```
DIM a$(1 TO 3, 1 TO 3)
```

you would have the following members to assign values to:

```
a$(1,1)     a$(1,2)     a$(1,3)
a$(2,1)     a$(2,2)     a$(2,3)
a$(3,1)     a$(3,2)     a$(3,3)
```

A two dimensional array is useful for tracking the status of each piece in a checkers game, or something of the like. Recall one of the last example programs in which we had a program that would ask the user for the item name, the item cost, and the quantity of that item, then spit out the data just given in a nice format with the total in the right hand column. Well, with only one item, this program isn't very practical. But now with our newfound knowledge of arrays and the knowledge we already have of loops, we can create a somewhat more useful application!

The process will start with the program prompting the user for the number of items that will be calculated, then the program will loop for the number of times that the user entered at the beginning, assigning we will declare each data to be entered into each member of the array. A variable called “netTotal!” will be displayed at the end of the program which will contain the total costs of the items. netTotal! will accumulate each time through the loop as the current totalCost! is added to it. Try the following code:

```
CLS

INPUT "How many items to be calculated? ", totalItems%

DIM itemName$(1 TO totalItems%)    ' Declare our arrays

DIM itemCost!(1 TO totalItems%)

DIM numItems%(1 TO totalItems%)

DIM totalCost!(1 TO totalItems%)

FOR i% = 1 TO totalItems%          'First loop: get inputs

  CLS

  PRINT "Item "; i%                ' Display the current item
number

  PRINT

  INPUT "Item name -- ", itemName$(i%)

  INPUT "Item cost -- ", itemCost!(i%)

  INPUT "Quantity --- ", numItems%(i%)

  totalCost!(i%) = itemCost!(i%) * numItems%(i%)

NEXT i%

CLS

PRINT "Summary"

PRINT

format$ = "\               \ $$#,###.##   #,###
$$#,###,###.##"

    PRINT "Item name         Item Cost    Quantity  Total
Cost    "

    PRINT "---------------- ----------   --------  ----------
----"

FOR i% = 1 TO totalItems%

  PRINT USING format$; itemName$(i%); itemCost!(i%);
```

```
numItems%(i%); totalCost!(i%)

  netTotal! = netTotal! + totalCost!(i%)

NEXT i%

PRINT

PRINT "Net Total = "; netTotal!

END
```

This program is much larger than anything we've done as of yet. It is kind of a review of everything we've done so far and one additional feature: the FOR...NEXT loop. This kind of loop loops for the number of times specified. A value is given to a variable and the program loops until that variable is equal or greater than the number specified after the TO.

This will loop 10 times, printing the numbers 1 through 10. The loop ends with a NEXT statement followed by the variable the loop increments for. So in our program, we have loops with index numbers (i%) starting at 1 and increasing for every number between 1 and the totalItems%, which is given by the user in the first part of the program. After the user inputs the number of items that will be calculated, four arrays are DIMensioned. They are all one dimensional arrays, so they aren't very complex. The first FOR...NEXT loop prompts the user for each item, then the format string is defined and the column headers are printed. The second FOR...NEXT loop cycles through the members of the four arrays and prints the data using the format string. The data for each member was assigned in the first FOR...NEXT loop. On each cycle through the second loop, the totalCost! of the current item being printed will be added to the variable “netTotal!”. The netTotal! is the total sum of the total costs. After the second FOR...NEXT loop, the net total is printed and the program ends.

File Processing

Let's say we have a game and when the user makes a new record score, they get to write their name on a list of the 10 best scores, but the next time the user plays the game, we want the name and position they recorded the last time they played to still be there. To do this, we must write to a "file" and then read it again later. Before you can do anything to a file, you must open it, and there are different ways you can open a file, believe it or not! A file can be opened so you can read from it or write to it, or it can be opened and split into records like a database. Here is a quick list of the different ways you can open a file:

- Input: Read data from the file
- Output: Write data to the file
- Append: Write data to the end of a file
- Binary: Read from or write to a file in binary mode
- Random: Read from or write to a file which is split up in records like a database

The syntax for the OPEN statement is quite peculiar; its arguments require us to specify a file name, an access type (one of the 5 types defined in variables), and a file number. When the file is open, QBASIC recognizes it by a number which we assign to the file when we open it. All references made to the file use this number. It can be any number from 1 to 255. An open statement may look like this:

```
OPEN "sample.txt" FOR INPUT AS #1
```

We will be reading data from this file because it was opened for input. Now back to our problem about the game scores; let's now set up a program that will ask the user for their name and give them a random score, then it will put their name on the list at the appropriate place on the top 10 (if it makes the list). We'll call the file "top10.dat." But say when the user buys the game, there are already 10 names and scores in

there, we then write the following program to put default names and scores into top10.dat:

```
CLS

OPEN "top10.dat" FOR OUTPUT AS #1

FOR i% = 1 TO 10

  playername$ = "Player" + STR$(i%)

  playerscore% = 1000 - (i% * 100)

  WRITE #1, playername$, playerscore%

NEXT i%

CLOSE #1

PRINT "Data written to file"

END
```

There are a couple strange features of this program that we have not seen yet. In the second line of the program, the file is opened for output so we can write to it. In the fourth line, we get into some light string manipulation; a name is generated from the word “player” and is concatenated with string form of the current loop number. You can concatenate two strings by using the + operator. The STR$ function returns the string representation of the number passed to it. The opposite of the STR$ function is the VAL function, which returns the numeric value of the string passed to it. Lastly, the WRITE statement writes to the file number specified as the first argument the values of the following arguments. Data is written to file in a format readable by the INPUT # statement which we will use in the actual program. We need this short program for the big one to work so we can give the program data to read from, or else we will get a nasty **INPUT PAST END OF FILE** error when we try to run it. Note that the file should be closed when we are done with it by using the CLOSE statement followed by the file number.

And now, we come to the big program itself! It is quite large and complex and I have not fully described all the statements used in it, so I have broken it down to five sections which I will describe in detail afterwards.

```
' section 1

CLS

RANDOMIZE TIMER

yourScore% = INT(RND * 1000)

PRINT "Game Over"

PRINT "Your score is "; yourScore%

DIM playername$(1 TO 10)    'Declare arrays for the 10 entries
on the list

DIM playerscore%(1 TO 10)

' section 2

OPEN "top10.dat" FOR INPUT AS #1

  DO WHILE NOT EOF(1)          ' EOF means "end of file"

    i% = i% + 1

    INPUT #1, playername$(i%)  'Read from file

    INPUT #1, playerscore%(i%)

  LOOP

CLOSE #1

PRINT


' section 3

FOR i% = 1 TO 10

  IF yourScore% >= playerscore%(i%) THEN
```

```
    FOR ii% = 10 TO i% + 1 STEP -1      'Go backwards (i% < 10)
      playername$(ii%) = playername$(ii% - 1)
      playerscore%(ii%) = playerscore%(ii% - 1)
    NEXT ii%
    PRINT "Congratulations! You have made the top 10!"
    INPUT "What is your name? ", yourName$
    playername$(i%) = yourName$
    playerscore%(i%) = yourScore%
    EXIT FOR
  END IF
NEXT i%

' section 4
OPEN "top10.dat" FOR OUTPUT AS #1
  FOR i% = 1 TO 10
    WRITE #1, playername$(i%), playerscore%(i%)
  NEXT i%
CLOSE #1

' section 5
PRINT
PRINT "Here is the top 10"
format$ = "\                         \  #### "
    PRINT "Player Name                 Score"
    PRINT "--------------------------  -----"
```

```
FOR i% = 1 TO 10

  PRINT USING format$; playername$(i%); playerscore%(i%)

NEXT i%

END
```

Section 1: The screen is cleared. The second line contains the statement `RANDOMIZE TIMER`. When dealing with random numbers, we must give the computer a number so it has something to base the random number it will create from. This number is called the random seed generator. A random seed generator can be specified with the `RANDOMIZE` statement. For the seed, we need a number that will not be the same every time we run a program, so we decide to use the number of seconds which have elapsed since midnight. The `TIMER` statement accesses a system device called the system timer and returns the current number of seconds which have elapsed since midnight. Since this number will change in each program we run, this can be used for the random seed. The variable "yourScore%" is given a random number from 0 to 1000. Lastly, we declare two arrays with 10 members each.

Section 2: In the first line we open the file for input, so we can read from it. The second line appears to be very weird at first because we are starting a loop with the `DO` statement, and then a condition to `DO WHILE`. The function `EOF` tests the file number passed to the function (in this case 1) and if the end of the file (EOF) has been encountered, it returns true. So `EOF(1)` is true if we are reading the end of the file. But we are using the Boolean operator `NOT`, so we want to loop while the end of file condition is false; we want to do the loop while we are not reading from the end of file 1. Then we assign the current data in the file to the arrays we declared in section 1. The `INPUT #` statement is used to read from the specified file into the specified variable(s) until a comma or carriage return is encountered.

Section 3: The main purpose of this section is to re-write the top 10 list if the player's randomly generated score places on the list. We do this by cycling through the list and testing if yourScore% is greater than or equal to (written in BASIC as >=) the current playerscore% being tested. If it is, then we have to shift each existing score below the current one down by one position to make room for the new score being added. The user is congratulated and prompted for their name. The loop is then exited using the `EXIT FOR` statement, which then goes to section 4.

Section 4: This short section simply opens the file for output so we can write to it, then we write each of the members of the array to file.

Section 5: Finally, we define the format string, print the headers and then print all the members of the top 10 and their scores, and that's the end of the program!

User Defined Data Types

Now, on to a new topic which will later become related to the previous, user defined data types. Recall that a data type is the type of value or data that a variable can have, such as integer, string, long, double, or single. You can create your own data types which contain one or more of the already defined data types. Here is an example of a user defined data type:

```
TYPE employeeType
  firstname AS STRING * 30
  lastname AS STRING * 30
  age AS INTEGER
  wage AS SINGLE
END TYPE
```

We have defined a new data type which consists of four members. We can now declare a variable of this type:

```
DIM employee AS employeeType
```

A variable of a user defined data type is like an array, in that it can have more than one value assigned to it. But you can have an array of a variable of a user defined data type as well, so things can get rather complex. Anyway, now that you have a user defined data type, you can assign values to the members of that variable. We use a period to access a member of a user defined data type like this:

```
employee.firstname = "Bob"
employee.lastname = "Foster"
employee.age = 24
employee.wage = 6.78
```

This could have been helpful in the last program we made with the top 10 list as we could have declared a user defined data type called "playerType" like this:

```
TYPE playerType
  name AS STRING * 20
  score AS INTEGER
END TYPE
```

and then declared an array of variables of that type,

```
DIM player(1 TO 10) AS playerType
```

That would have made our code more efficient, but not necessarily more readable. Notice that when we declare a string in a user defined data type, it seems as if we are multiplying it by a number. Actually, the number after the * defines the maximum length of the string and you

must define this because the size of a user defined type must be known by the computer. Any value assigned to this string data member that exceeds the length specified is truncated.

Random Access Files

User defined data types can serve more than to be efficient, they are the heart of the random access file mode, which is commonly used in database files. A database is a method of organizing large quantities of information in records and fields. In a record, there are a set of fields which are constant in every record. A field's value changes from record to record, however but only the name of the field remains constant.

So how does this relate to user defined data types? Well think of a variable of a user defined data type as a record in the database, and the members as fields of the records; employee may be a record, and firstname, lastname, age, and wage may be fields. Values can be assigned to the fields in each record, thus constructing a database-like structure.

A file opened for random access is organized with records split into fields; each record in the random access file is given a record number which can be convenient in a database environment. In the `OPEN` statement for opening a random access file, there is one extra argument! We must specify the length in bytes of how much space one record will occupy (or the record length). This can be easily taken by taking the LENgth of a variable defined as the user defined data type we are going to use. So back to our employee example, we could use the `LEN` function to get the size in bytes of the employee variable, which is an “employeeType”:

```
recordLen# = LEN(employee)

OPEN "database.dat" FOR RANDOM AS #1 LEN = recordLen#
```

Note: LEN stands for length. You can also use the `LEN` function to get the number of characters in a string.

So let's construct a simple “database” that will keep track of the employees of a business:

```
' Section 1
CLS
TYPE employeeType
  firstname AS STRING * 30
  lastname AS STRING * 30
  age AS INTEGER
  wage AS SINGLE
END TYPE
DIM employee AS employeeType


' Section 2
PRINT "1.) Create new recordset"
PRINT "2.) View existing recordset"
INPUT "Which option?  ", selection%
' Section 3
IF selection% = 1 THEN
  INPUT "How many employees are in the company?  ", numRecords%
  recordLen# = LEN(employee)
  OPEN "database.dat" FOR RANDOM AS #1 LEN = recordLen#
    FOR i% = 1 TO numRecords%
      CLS
      INPUT "First name:  ", employee.firstname
      INPUT "Last name:   ", employee.lastname
```

```
      INPUT "Age:          ", employee.age

      INPUT "Wage:         ", employee.wage

      PUT #1, ,employee

    NEXT i%

    CLS

  CLOSE #1

  PRINT "Recordset creation complete"

  END

END IF

' Section 4

IF selection% = 2 THEN

  recordLen# = LEN(employee)

  OPEN "database.dat" FOR RANDOM AS #1 LEN = recordLen#

    format$ = "\                \,\                \   ###
$$##.##"

        PRINT "Last name          First name           Age
Wage   "

        PRINT "----------------- ----------------   ---  -----
--"

    DO WHILE NOT EOF(1)

      GET #1, ,employee     'Sorry about the length of this
line!!!

      PRINT USING format$; employee.lastname; employee.firstname;
employee.age; employee.wage

    LOOP

  CLOSE #1

  END

END IF
```

Again, I've split this program into sections again because that seems to work well for the larger ones.

Section 1: We're defining the user defined data type and declaring a variable of that type.

Section 2: The first thing the user sees is a menu with the option to either create a new database (or recordset) or view the existing one. The user is prompted to make a selection which is stored in the variable “selection%”.

Section 3: If the user chose option 1 (to create a new recordset) then this code is executed. First we prompt the user for how many employees are in the company so we know how many times to go through our loop, then we open the file, prompt the user for the data for each variable and then write the whole record to file. The record is written using the PUT statement. The first argument in PUT is the file number, the second is the record number and the third is the data to be written to file. If no record number is specified for the second argument, the current file position is used, which will just append what we specify after what is already there. This works fine, so we don't need to worry about explicit record numbers.

Notice that we are now writing the whole employee variable to file. This is because we write records to file, and the whole variable contains the data for the data members (or fields).

Section 4: If the user chooses to view the existing recordset, then we first open the file, define a format string for the printout and then print the headers. Next we have a loop until the end of file is encountered.

Notice the GET statement, which is used to read from a random access file. The first argument of GET is the file number we want to read from, the second is the record number, which we are leaving blank because we can read from the current position (CP) like we did in the PUT statement, and the third is the variable in which we read the data into. This variable must be of the same data type that we wrote with or else the data types will be incompatible; you'd probably get a **TYPE MISMATCH** error if a different variable is used because the fields are not equal, so the program does not know what to assign the data to.

Well that's it for random access files. If you have understood half of what I've said, then feel good because you have a good knowledge of what QBASIC is about. Now on to some more advanced programming!

Graphics

Graphics programming in QBASIC can get fairly complex so let's start from the beginning. Your screen is made up of hundreds of pixels and the number of pixels horizontally and the vertically determines the resolution of your monitor. Right now, your monitor is set up in a video graphics mode which determines how many pixels can be displayed on screen and my resolution is set to 1024x768, which is the most common.

Graphic Modes

Your graphics mode is determined by the screen resolution in pixels, the text resolution, that is, how many lines and columns of text can fit on your screen, the number of pages of video memory, and the color palette. There are 13 screen graphics modes in QBASIC and each has its different purpose. You can look in the help index in QBASIC for a listing of the screen graphics modes and their specifications. Each aspect of a screen graphics type can be changed to create effects.

Graphic Commands

There are a number of graphics routines used in QBASIC which allow a variety of graphical effects so let's try a few:

```
SCREEN 12

LINE (0,0)-(800,600), 1

CIRCLE (320, 240), 20, 2

PSET (10,10), 14

DRAW "c15 bm100,400 l5e5f5l5"

END
```

The first line initializes the graphics mode to 12, which is 16 colors, 1 page of video memory, and 800x600 resolution.

LINE draws a line from one coordinate to another. The first optional argument after the coordinates (which are not optional) is the color. After that, a B ("box") or BF ("box fill") can be used to draw a box or a box filled with the color specified. The first coordinate can be omitted and the dash (-) left in to draw a line from the current graphics position (CP) to the relative coordinates specified, so LINE -(100,0) will draw a line from the current graphics position to 100 pixels to the right.

CIRCLE draws a circle with the center at the coordinates specified. The first argument (required) after the coordinates is the radius of the circle, then the color. After that, if you want to draw an arc, is the starting angle of the arc in radians, then the ending angle of the arc. To make an arc, first touch up on your geometry, then recall that to convert from degrees to radians is pi (3.14159265) divided by 180. The last argument is used if you want to make an ellipse, and is the ratio of the y axis to the x axis. So CIRCLE (320,240), 20, 2, 3.1415, 0, .5 would draw an elliptical green arc with the center at the middle of the screen, starting at 180 degrees (pi) and going to 0 degrees, with a compression ratio of 1 to 2 (the x axis will be twice as big as the y axis). Try it, and you'd see that this looks like a wide smiley face mouth, doesn't it?

PSET fills a pixel at the screen coordinate you specify with the color you specify. In this case, yellow.

Finally, the DRAW statement; the DRAW statement has its own commands which I strongly suggest you memorize. When we get in to scaling and rotation you will need to know your draw commands pretty well. The draw command in the above code example can be read as "color 15 (white), move without drawing to screen coordinate 100,400, draw left 5 units, draw up and right 5 units, draw down and right 5 units, and draw left 5 units" or in other words, a triangle. A unit is set by the current scale mode, which by default is 4. Now since default scale mode is 4, one unit represents 4 pixels. So our triangle is 40 pixels wide at the base.

Colors

There are 16 defined colors in QBASIC. The COLOR statement sets the current color for text output. I highly recommend memorizing the colors as well. Now try this program:

```
SCREEN 12

FOR i% = 0 TO 15

  COLOR i%

  PRINT "COLOR"; i%

NEXT i%
```

This will print out the 16 colors used in QBASIC. 0 is black and so it obviously won't show. A quick reference for colors while you're in the QBASIC IDE (integrated development environment) is to look under the OPTIONS | DISPLAY menu, the colors listed there are in the QBASIC order, starting with black and ending with bright white.

Now you know the basic graphics routines and their uses, let's make a couple of programs that demonstrate them to a greater extent. First, a program which prompts the user for a radius, calculates the area and circumference, and then draws the circle in a random color on the screen.

```
SCREEN 12
RANDOMIZE TIMER
CONST pi! = 3.1415
DO
  COLOR 15: INPUT "Radius (-1 to quit) --> ", radius!
  IF radius! = -1 THEN EXIT DO
  area! = pi! * radius! ^ 2
  circum! = pi! * 2 * radius!
  COLOR 14
    PRINT "Area =        "; area!
    PRINT "Circumference = "; circum!
  CIRCLE (320,240), radius!, INT(RND * 15 + 1)
  DO: LOOP WHILE INKEY$ = ""
  CLS
LOOP
COLOR 9: PRINT "Good Bye!"
END
```

We first set the screen graphics mode and generate a random seed number based on the system timer then we prompt for the radius in a vivid bright white, and test to see if we should end the program. We then calculate the area and circumference and then print the results in yellow. Then we draw the circle from the middle of the screen at the radius given in a random color. This random color is set by first generating a random number from 0 to 14, adding 1 to it, and converting it to an integer with the `INT` function. The next line seems weird; the `INKEY$` statement reads the keyboard and returns the string representation of the key pressed.

We are looping while INKEY$ is nothing or in other words, while the user isn't pressing anything. The loop goes on forever until the user presses any key, and at this time a value will be given to INKEY$ which you might decide to use. The screen is then cleared for the next entry. If the user breaks the loop by entering -1 for the radius, we print “Good Bye!” in bright blue letters.

Color Palette

There are a lot more colors than just 16! In fact, you can change the values of each of the 16 colors to represent some other color you specify! You can do this with the PALETTE statement (applies to screen modes 12 and 13), which has two arguments: the color you want to change and the color you specify. Specifying a color is the hard part but here is one version of the syntax of the PALETTE statement:

```
PALETTE color, blueValue * 256^2 + greenValue * 256 + redValue
```

Color is the color you are changing while the _Values are numbers from 0 to 63 which specify the intensity of that color. You must use the multipliers after the values and use the addition operator to separate them. So let’s make a program that fades the screen in and out, from black to purple (adding blue & red gives purple).

```
SCREEN 12
DO
  FOR i% = 1 TO 63
    PALETTE 0, i% * 256 ^ 2 + i%
  NEXT i%
  FOR i% = 63 TO 1 STEP -1
    PALETTE 0, i% * 256 ^ 2 + i%
```

```
  NEXT i%

LOOP WHILE INKEY$ = ""

END
```

We start by changing the value of black (0), which is the background color to purple, from one degree of blue + red to the next, then we bring it back down to black by decreasing the blue + red value. We do this over and over until the user presses a key or begins to have seizures!

Rotation

Scaling & rotation can be accomplished quite easily with the DRAW statement, although it involves some weird looking code. But first, let's define a shape that we can scale and rotate:

```
box$ = "bu5 l5 d10 r10 u10 l5 bd5"
```

Interpretation: "move up 5 spaces without drawing, draw 5 spaces left, draw 10 spaces down, draw 10 spaces right, draw 10 spaces up, draw 5 spaces left, and move 5 spaces down without drawing" This forms a box. Notice that I started at the center and not at a corner or side which would seem to be easier. Well, when you rotate something, it draws based on the starting point of the object and we want it to rotate so if we put a pen at each corner of the box, it would draw a perfect circle. Therefore we set the center of the box as the starting point of the object. This can be called the "object handle". The ta draw command stands for "turn angle", and obviously turns the object in the degrees you specify. So if we turned the box from 0 to 360 degrees, drawing the box at each step and erasing the previous image, we would get a rotating box. But we need one more function: VARPTR$. VARPTR$ stands for "variable pointer", a term you can completely ignore unless you get into C or Assembly Programming.

Now we need to somehow get the box$ shape into the draw string command we will implement in the loop, so we have to take the address of the object string and plug it into the draw string. This can be accomplished by using the X command, which tells VARPTR$ where to plug in the string's address so it can be used. With box$ defined above, here's the code for a rotating box:

```
DO

 angle% = angle% + 1

 IF angle% >= 360 THEN angle% = 1

 DRAW "c0 bm320,240 ta" + STR$(angle% - 1) + "X" +
VARPTR$(box$)

 DRAW "c1 bm320,240 ta" + STR$(angle%) + "X" + VARPTR$(box$)

LOOP WHILE INKEY$ = ""

END
```

Not that hard is it? We draw the box at the previous angle in black, and then draw the box at the current angle in blue.

Scaling

Scaling is done pretty much the same way, but instead of changing the angle and erasing the previous image, we change the scale factor and erase the previous image. Recall that the default scale factor for the DRAW statement is 4 pixels per unit. Well, if we increase this factor then we will have more pixels per unit, thus giving the image the effect of enlargement. So if we set up a FOR...NEXT loop which will increase the scale factor from 2 to, say, 200, we will get the effect of scaling, but let's start with a smaller image which is maybe 8 pixels wide from the start instead of 40 so we get a more dramatic effect.

```
SCREEN 12

box$ = "bu4 l4 d8 r8 u8 l4 bd4"

FOR s% = 2 TO 200

  DRAW "c0 bm320,240 s" + STR$(s% - 1) + "X" + VARPTR$(box$)

  DRAW "c2 bm320,240 s" + STR$(s%) + "X" + VARPTR$(box$)

NEXT s%

END
```

Notice what we're doing here, we are starting the scale factor at half of default (2) because the FOR...NEXT loop starts with s% at 2. The s draw command sets the scale factor. Also notice that we must continuously anchor the object handle at a point to keep it scaling about the handle, we do this by moving the object handle to 320,240 (center of screen) each time through the loop. Whenever we want to put a number into the draw string, we must concatenate the string format (STR$) of the number within the string. Instead of concatenating the box$ with the rest of the string, it is faster to only pass the address of the substring with the VARPTR$ function.

Scaling & Rotation

So what if you want to scale and rotate something at the same time? Simple; just set up a FOR...NEXT loop which increases the scale factor as before and then within the loop, increase the angle. But instead of subtracting a factor for the angle to erase the previous angle, you erase the previous image with the current angle, increase the angle and then draw the current image with the new current angle. This way, if we want to change the factor at which the angle increases, we will only have to change one number instead of two:

```
SCREEN 12

box$ = "bu4 l4 d8 r8 u8 l4 bd4"

FOR s% = 2 TO 250

  DRAW "c0 bm320,240 ta" + STR$(a%) + "s" + STR$(s% - 1) + "X"
+ VARPTR$(box$)

  a% = a% + 1

  IF a% >= 360 THEN a% = 1

  DRAW "c1 bm320,240 ta" + STR$(a%) + "s" + STR$(s%) + "X" +
VARPTR$(box$)

NEXT s%

END
```

Draw strings can get fairly complex, but you'll get used to them with practice and when you memorize the draw string commands.

Logical Planes

The screen coordinates for different screen modes can be fairly difficult to work with, and they do tend to be weird numbers. To make your code simpler to write, you can define a logical plane over the physical plane. An example of a physical plane is the 640x480 resolution established by the SCREEN 12 screen mode. You can define a logical or alternate user-defined plane with the `WINDOW` statement.

```
SCREEN 12

WINDOW (0,0)-(100,100)

CIRCLE (50,50),10,4

LINE (0,0)-(50,50),2

END
```

This trivial example defines a logical plane which is 100x100 with 50,50 as the center of the screen; so this draws a red circle from the center with a radius of 10. The line statement draws a green line from the lower left corner to the center of the screen. Notice that defining a logical plane sets the origin (0,0) to the bottom left of the screen, instead of the default upper left. If you want the origin to be in the upper left with a logical plane, add the SCREEN keyword after WINDOW. So to define the graphics mode 12 screen resolution, the code is:

```
SCREEN 12

WINDOW SCREEN (0,0)-(640,480)
```

Use whatever is more comfortable, but I would recommend using WINDOW SCREEN because there is less confusion when converting from logical to physical planes.

Special Effects

Finally, a little information on creating DRAW effects with the other QBASIC graphics routines. I hope you know some trigonometry for this part! Recall that in the unit circle which has a radius of 1, the coordinates of a point on the circle given an angle is defined as (COS(angle), SIN(angle)). Furthermore, if we are given a point on the circle, we can find the angle by drawing a vertical line perpendicular to the x axis from the point. If we take the arctangent of the vertical length of this line divided by the horizontal distance of this line from the origin, we will get the angle. So the angle is defined as ATN(Y/X). With this knowledge, it would be possible to create a spinning line using only the line command. If we create a loop which increments the angle from 0 to 360 then we can take the COS, SIN of the angle to get the point we should draw to. But there's only one more problem; the QBASIC functions COS and SIN think in radians, so we must first convert the angle to radians by multiplying pi/180:

```
SCREEN 12

CONST PI = 3.1415

WINDOW SCREEN (-1,1)-(1,-1)

DO

  LINE (0,0)-(COS(a% * PI / 180),SIN(a% * PI / 180)), 0

  a% = a% + 1

  IF a% >= 360 THEN a% = 1

  LINE (0,0)-(COS(a% * PI / 180),SIN(a% * PI / 180)), 14

LOOP WHILE INKEY$ = ""

END
```

We start by initializing the graphics mode, then defining PI as a constant (a variable which will never change in the program execution) then define the logical plane, and start the loop. The line starts from the center of the screen and goes to the coordinate specified by the COS, SIN of the angle. We loop until the user presses a key.

There is one more type of graphics that QBASIC has a strong point for: text! Graphical effects can be made quite easily using only text in QBASIC. There are a few functions that are quite useful when dealing with text. The first is the CHR$ function; if you pass a number to the CHR$ function, it will return the ASCII (American standard code for information interchange) text value of that number. To find a listing of the ASCII character codes, look in the help contents, and there is a listing there. For example, to print a smiley face on the screen, the code is this:

```
CLS

PRINT CHR$(1)

END
```

Since the ASCII character code for a smiley face is 1, you can use the CHR$ function to get this. Another useful function is ASC, which returns the ASCII value of a text value you pass to it, so ASC("A") will return 65 because the ASCII value of A is 65. Every printable character has an ASCII value, so these two functions make it quite easy.

Finally, the LOCATE statement is extremely useful for any text based program; LOCATE sets the text CP to the coordinates you specify. The first argument is the column, and the second is the row. So,

```
CLS

LOCATE 5,10

PRINT CHR$(219)

END
```

will print a solid white block at column 5, row 10. And that's it for graphics! You now know nearly every graphics routine in QBASIC, and have the knowledge to make a game or highly graphical program.

Graphics depend on how you arrange them, so it requires an artistic skill to some degree. If you get creative with these graphics commands, you can create nearly any effect you need!

Designing Applications

It is not practical in real world terms to set up an application in one long list of code. Many early programming languages were purely linear, meaning that they started from one point on a list of code, and ended at another point. All of the codes written in this tutorial so far has been purely of this nature; linear! However, linear programming is not practical in a team environment. If one person could write one aspect of code and another write another part of the program, things would be much more organized. QBASIC contains the capability to meet these needs and it called “modular programming”; you can break a program into different "modules" which are separate from the main program and yet can be accessed by any part of it. I highly recommend the use of separate modules in programming applications, although it is not a simple task to learn.

Procedures

These separate modules are also known as procedures in the QBASIC environment. There are two types of procedures: subs and functions.

Subs merely execute a task and return to the main program, while functions execute a task and return a value to the main program. An example of a sub might be a procedure which displays a title screen on the screen, while a function may be a procedure that returns a degree in degrees given a number in radians. Function procedures are also used in Calculus, so you Calculus people should already be familiar with functions.

Procedures can accept arguments in what is called an argument list. Each argument in the argument list has a defined data type and an object of that type must be passed to the procedure when it is called. For example, the CHR$ function accepts a numeric argument. The function itself converts this numeric argument into a string representation of the ASCII value of the number passed, and returns this one character string.

Procedures in QBASIC are given their own screen. When you enter the QBASIC IDE, you are in the main procedure which can access all the others. Other procedures are created by typing the type of procedure (SUB or FUNCTION), the procedure name and followed by the complete argument list, if any. You can view your procedures through the VIEW menu on QBASIC. Here is an example of a sub procedure which performs some operations for a program that will be using graphics, random numbers, and a logical plane:

```
SUB initProgram()
  RANDOMIZE TIMER
  SCREEN 12
  WINDOW (0,0)-(100,100)
  COLOR 15
END SUB
```

The only thing you need to type is SUB initProgram() and the screen will be switched to that procedure. The END SUB is placed there for you, so the only thing you need to type then is the code within the sub. Try typing this out on your own to see how this works. This procedure is called by simply typing initProgram in the main procedure.

An alternative method is CALL initProcedure(). Right here, the parentheses are optional, but if you were to pass arguments to the procedure, parentheses would be required with the CALL statement. Now lets try passing an argument to a procedure. We will pass two arguments to a procedure called center (a string containing the text to be centered) and the horizontal location on the screen at which you wish to center it:

```
SUB center( text$, hLoc% )

  LOCATE hLoc%, 41 - (LEN(text$) / 2)

  PRINT text$

END SUB
```

The first line after the sub declaration positions the starting point of the text at the horizontal location we passed at the second argument and vertical coordinate. The vertical coordinate is calculated by subtracting one half the screen's width in characters (41) and half the LENgth of the text we passed as the first argument. We would call center from the main procedure like this:

```
center "HTML Online", 12
```

Or like this:

```
CALL center ("Programmed by qp7@pobox.com", 12)END
```

It's quite simple actually. Functions are slightly different and involve an additional part which subs don't have, which is a return value. The return value is specified by assigning the value you want to return to the function name from within the function definition. When calling the function from within the main procedure, the name of the function is treated as a value which is evaluated at compile-time. Here is an example of a function definition:

```
FUNCTION convert.To.Radians (degree!)

  LET PI = 3.1415

  convert.To.Radians = degree! * PI / 180

END SUB

The function is implicitly called in this program

CLS

INPUT "Enter a value in degrees: ", degreeValue!

radianValue! = convert.To.Radians(degreeValue!)

PRINT "The radian equivalent is"; radianValue!; "radians"

END
```

We treat the value returned from the function as a value we that can immediately assign to another variable. The variable “radianValue!” is given the value returned from convert.To.Radians. These concepts are supported in all programming languages, so this information will be beneficial to you in the future!

A Simple “Game”

There is one final concept which has proven to be very successful in programming: a message loop. With QBASIC, you can construct a loop which runs for the length of the program, receives input from the user, and executes a message based on what the user does. We will construct a basic application which receives input from the user in the form of an arrow key, and moves a box on the screen based on the direction the user pressed. The arrow keys are different from normal inputted keys received with INKEY$. On the enhanced 101 keyboards which have arrow keys, INKEY$ returns two values: the ASCII text representation of the key pressed and the keyboard scan code of the key pressed. Since the arrow keys do not have an ASCII text representation, we must use the keyboard scan codes for them. The keyboard scan codes can be viewed in

the HELP | CONTENTS section of the QBASIC menus. For this program, we will have two procedures in addition to the main procedure. The first will initialize the program settings and position the character in its starting position while the other will move the guy in the direction which we pass to the function. The main procedure will call the sub procedures and contains the main message loop which retrieves input from the user. First of all, here is the code for the main procedure:

```
CONST UP = 1
CONST DOWN = 2
CONST LEFT = 3
CONST RIGHT = 4
TYPE objectType
  x AS INTEGER
  y AS INTEGER
END TYPE
DIM object AS objectType
initScreen
object.x = 41
object.y = 24
DO
  SELECT CASE INKEY$
    CASE CHR$(0) + CHR$(72): move UP, object
    CASE CHR$(0) + CHR$(80): move DOWN, object
    CASE CHR$(0) + CHR$(75): move LEFT, object
    CASE CHR$(0) + CHR$(77): move RIGHT, object
    CASE CHR$(32): EXIT DO
```

```
  END SELECT

LOOP

LOCATE 1,1: PRINT "Thank you for playing"

END
```

This code is fairly self explanatory with the exception of the SELECT CASE... END SELECT structure which has not yet been explained. This type of conditional testing format tests a condition and several cases for that condition are then tested. In this case, we are checking IF INKEY$ = CHR$(0) + CHR$(72), IF INKEY$ = CHR$(0) + CHR$(80), and so on. This is just a more legible format than the IF...THEN...ELSE. Note that in the QBASIC compiler, a CASE ELSE statement is required in the structure. The above code is the driver for the rest of the program. First some CONSTants are declared which remain constant for the duration of the program and in any other module. A user defined data type is declared to store the coordinates of the character. Then an endless loop is executed, calling the appropriate procedure for the arrow key pressed until the user presses the space bar (CHR$(32)). Here is the code for the initScreen procedure:

```
SUB initScreen ()

  SCREEN 12

  COLOR 9

  WIDTH 80,50

  LOCATE 24,41

  PRINT CHR$(1)

END SUB
```

The `WIDTH 80,50` statement sets the screen's text resolution to 80 columns and 50 rows then we print a smiley face in the middle of the screen in a nice bright blue color. Next we need to write the move procedure, and then we will be done with the program:

```
SUB move (way AS INTEGER, object AS objectType)
  LOCATE object.y, object.x
  PRINT CHR$(0)       ' erase previous image
  SELECT CASE way
    CASE UP: IF object.y > 1 THEN object.y = object.y - 1
    CASE DOWN: IF object.y < 49 THEN object.y = object.y + 1
    CASE LEFT: IF object.x > 1 THEN object.x = object.x - 1
    CASE RIGHT: IF object.x < 79 THEN object.x = object.x + 1
  END SELECT
  LOCATE object.y, object.x
  PRINT CHR$(1)       ' draw current image
END SUB
```

And that's the whole program, confusing as it may be! Ideas should be going through your head about what you could do with this information. Entire games can be created with this simple construct!

There are more things to consider which are beyond the scope of this tutorial, but if you were to design an application in QBASIC, you would only need the information from this section and one heck of an imagination. Programming takes knowledge of the language and a creative mind and programs are made by programmers with both! With a creative mind, you can develop any program conceivable!