Fundamentals

B	***A***	***S***	***I***	***C***
A	**L**	**Y**	**N**	**O**
S	**L**	**M**	**S**	**D**
I		**B**	**T**	**E**
C	**P**	**O**	**R**	
	U	**L**	**U**	
	R	**I**	**C**	
	P	**C**	**T**	
	O		**I**	
	S		**O**	
	E		**N**	

Loading BASIC

You will see an icon on your desktop labeled ***BASIC***. Double click on this icon to enter the BASIC programming environment.

The ***OK*** is the BASIC prompt telling the user that it is ready to receive instructions (See page 4).

BASIC MODES

1. **Direct Mode** allows the user to issue one command at a time.
   - Example : ***PRINT "HELLO WORLD"***
     The text "HELLO WORLD" will appear on the screen in the BASIC environment.
2. **Programming Mode** allows the user to issue a series of commands that are executed in numerical order as per the line number.
   - Example:

```
10 A$ = "HELLO WORLD"
20 L% = 10
30 R% = 25
40 V% = L% + R%
50 PRINT A$
60 PRINT V%
70 END
```

The result of this program is:

HELLO WORLD
35

Data Types

Just as mathematics include different types of numbers- integers, real numbers, imaginary numbers, etc., computers can deal with various types of data as well.

Numbers

1. INTEGER
   - Whole numbers
   - Can be positive, negative, or zero
   - Can range between –32,768 to 32,767
   - Designation
     1. **DEFINT**
     2. Last character of the variable name a ***%***
   - Example: In the following code, the variable ***A*** will be designated as an integer and then it will be set equal to the sum of 3 plus 4
     ```
     10  DEFINT A
     20  A = 3 + 4
     ```
   - Example: In the following code, an integer variable will be created using the ***%*** suffix and immediately assigned to the sum of 3 plus 4
     ```
     10  A% = 3 + 4
     ```
3. SINGLE-PRECISION
   - Numbers with fractional portions to the right of the decimal point
   - **If no designation is made, variables default to single precision numeric**.
   - Range is –1.7E+38 to +1.7E+78
   - When printed, 7 digits to the right of the decimal point are displayed but only 6 are accurate
   - Designation
     1. **DEFSNG**
     2. Last character of the variable name is a ***!***
   - Example: In the following code, the variable ***R*** is designated as a single precision variable and then, set equal to the quotient of 17 divided by 2.9.
     ```
     10  DEFSNG R
     20  R = 17 / 2.9
     ```
   - Example: In the following code, a single precision variable will be created and set equal to the quotient of 17 divided by 2.9.
     ```
     10  R! = 17 / 2.9
     ```
   - Example: Since the default data type is single precision numeric, the following code also creates a single precision variable and sets it equal to the quotient of 17 divided by 2.9.
     ```
     10  R = 17 / 2.9
     ```

3. DOUBLE PRECISION
   - Numbers with fractional portions to the right of the decimal point
   - Range is –1.7S+38 to +1.7D+38 (The ***D*** indicates exponential notation of double precision)
   - Accurate to the first 16 digits to the right of the decimal point

- Used mostly for fine calculations such as engineering to close tolerances etc.
- Designation
  1. DEFDBL
  2. Last character of the variable name is a ***#***
- Example: In the following code, the variable ***S*** is designated as a double precision variable and then, it is set equal to the quotient of .0005987 / 2500

```
10  DEFDBL S
20  S = .0005987 / 2500
```

- Example: In the following code, we will create a double precision variable and immediately assign it a value as before

```
10  S# = .0005987 / 2500
```

Strings

- A set of literal characters
- Numerals can also be strings but, as such, they cannot be used for calculations. They are merely numerals without numeric value.
- Can be up to 255 characters long
- Designation
  1. DEFSTR
  2. Last character of the variable name is a ***$***
- Example: We will designate the variables X, Y, and Z as strings and then assign a string literal to each

```
10  DEFSTR X,Y,Z
20  X = "I"
30  Y = "Love"
40  Z = "Computers"
```

Notice that when assigning a literal value to a string, it **must be enclosed inside quotation marks**. These are called ***string literals***.

- Example: We will create string variables and assign literals to each at creation time

```
10  X$ = "I"
20  Y$ = "Love"
30  Z$ = "Computers"
```

Rules

1. If a variable is assigned a value in a program and later in the program is assigned a value again, it assumes the value of the last assignment.
2. An error message results if a variable is assigned a value outside of its variable type. For example, A = "Hello" is incorrect because ***A*** is, by default a single precision numeric variable and "Hello" is a string literal.
3. A variable that has not been defined with a DEF--- statement or a suffix, defaults to a single precision numeric variable (see SINGLE PRECISION above).

Order of Operation

1. Powers
   - Designated with the up carrot (^)
   - Example: **5^2** is 5 squared
2. Multiplication and Division
   - Multiplication is designated with an asterisk (*)
   - Division is designated with a forward slash (/)
3. Addition and Subtraction
4. Parenthesis change the order of operation by creating terms
   - Example: 5 * 2^2 + 3 * 3

     5 * 4 + 3 * 3

     20 + 9 = 29

   Using parenthesis to create terms:

     (5 * 2)^2 + 3 * 3

     10^2 + 3 * 3

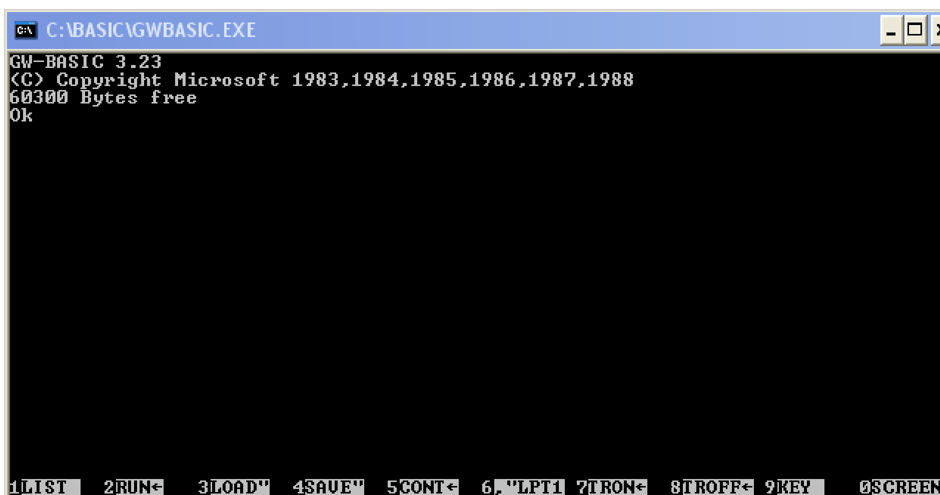
     100 + 3 * 3

     100 + 9 = 10

Hot Keys

There are four major hot keys used with BASIC. Hot key designations appear at the bottom of the BASIC programming environment screen:

1. **F1** = List the current program in memory
2. **F2** = Run the current program in memory
3. **F3** = Load a program into memory from disk
4. **F4** = Save the current program in memory



Instruction Set 1

In this section, we will explore the first fundamental set of commands that will be the building blocks for future command sets.

BEEP

This command forces the computer's internal speaker to emit a beep. It is used when the user's attention is required. For example, the user may have entered incorrect data and execution of the program will be halted until the he/she responds in some way to the problem. For example, if we want to emit a beep in line 200, we would type: **200 BEEP**

END

This is the command that signals the interpreter that the program has reached the end of its code and no more commands will be executed. It is usually placed at the end of the code but need not be at the end. It must be remembered that execution of the code will cease as soon as the ***END*** command is encountered. For example, if we want to end the program at line 500, we would type: **500 END**

GOTO

At times, it will become necessary to execute code in a different order than that of the numbered set of commands as they are written in the program. For instance, the program may need to skip a section of code depending upon the response of the user. In this case, a ***GOTO*** command will transfer execution of the code to a place other than the next numbered line. For example, here line 100 sends control to line 700: **100 GOTO 700**

LET

It is a fundamental rule of syntax that every line of code must contain a BASIC keyword (command). If the programmer wishes to assign a value to a variable, he/she can do it in two ways. First, without the ***LET*** command: **100 X = 5.379**. Here, in line 100, the ***LET*** command is **assumed**. This is one of the few times a line of code does not require a command. Alternatively, the command could be used: **100 LET X = 5.379**.

PRINT

The ***PRINT*** command is used to send output to the monitor screen. For example, if we wanted to ***print the sum of 10 and 7 in line 300***, we could combine the ***LET*** command with the ***PRINT*** command as follows:
290 LET X = 10 + 7
300 PRINT X
Alternatively, we could actually do the math in the ***PRINT*** command itself:
300 PRINT 10 + 7
In both cases, the number 17 would appear on the screen.

Following a ***PRINT*** statement, any subsequent prints will follow a **carriage return/line feed**. In other words, if lines 200 and 210 are successive ***PRINT*** statements:

200 PRINT "THE SUM IS "
210 PRINT 10 + 7
The resulting print would look like this:

```
THE SUM IS
17
```

If we want to have the "17" appear to the right of the "THE SUM IS ", we need to **place a comma** after the first print statement:

200 PRINT "THE SUM IS ";
210 PRINT 10 + 7
Now the resulting print appears like this:

```
THE SUM IS 17
```

We could combine both ***PRINT*** statements into one command as follows:

200 PRINT "THE SUM IS "; 10 + 7

Notice in both cases where a comma is used to remove the carriage return/line feed, it was necessary to **place a space between the "*S*" in the word "*IS*" and the trailing quotation mark**. This is because a space is not assumed at the end of a ***PRINT***.
Without the space, the output would look like this:

```
THE SUM IS17
```

Notice that there is no space between "IS" and "17".

IF THEN

The ***IF THEN*** construct tests for a condition. **If the condition is true, a command will be executed and execution continues at the next line of code. If it is not true, execution simply continues with the next line of code.**
For example, if we combine the ***LET, IF THEN,*** and ***PRINT*** commands, we can test the sum of two variables. If the sum is greater than 100, we can print that sum, otherwise, we end the program:

100 X = 85
110 LET Y = 25
120 IF X + Y > 100 THEN PRINT "The sum is greater than 100"
130 END

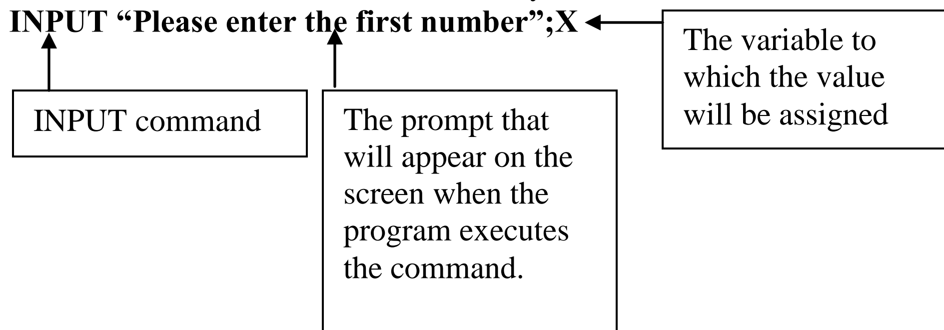
INPUT

Most computer programs require the user to furnish information to the computer in order to complete the program's assignment. For instance, if the program is to calculate the area of a square, the user will need to provide the length and width values for the calculation to take place. This is where the ***INPUT*** command comes in. Looking at the example in the IF THEN section above, let's have the user enter the values for X and Y:

100 INPUT "Please enter the first number";X
110 INPUT "Please enter the second number";Y
120 IF X + Y > 100 THEN PRINT "The sum is greater than 100"
130 END

If we dissect line 100, we can see the way an ***INPUT*** statement is constructed:

INPUT "Please enter the first number";X



Notice a few things:

1. The ***INPUT*** statement **always begins with the *INPUT* command**
2. The prompt that will appear on the screen is **always enclosed inside quotation marks**
3. A **semicolon (;) always separates the prompt from the variable**
4. The variable's data type **must match the type of data being assigned to it**
    - If you want to assign a string such as a name to a variable using an ***INPUT***, you need to either assign the variable as a string using the **DEFSTR** command or by appending the variable name with a **$**.
        - In the above example, you would use:
          INPUT "Please enter a name";X$
5. **If you do not desire any sort of prompt, simply use the *INPUT* with the variable.**
   100 INPUT X$
    - In the above case, the ***INPUT*** is usually preceded with a ***PRINT*** statement:
      90 PRINT "Please enter a name"
      100 INPUT X$

:

The colon is used to "piggyback" more than one command within the same line of code:

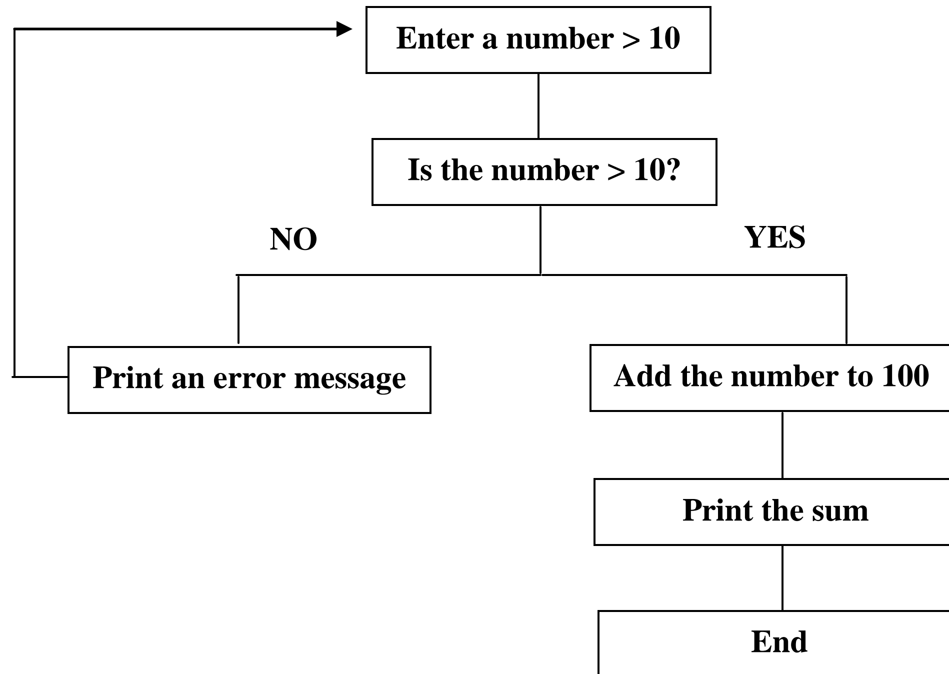
100 PRINT "Please enter a name" : INPUT X$

Normally only one command is allowed in a single line of code.

Logic Charts

The easiest way to visualize what your program will do is to construct a logic chart. For example, let's assume that we want to have the user:

1. Enter a number greater than 10
2. Check to see if the entered number is greater than ten
    - If so, add it to 100 and print the sum and end
    - If not, beep the user, print an error message, and have him/her enter another number



The program would look like this:

```
100 INPUT "Please enter a number greater than 10"; X
110 If X > 10 THEN GOTO 140
120 BEEP : PRINT "Incorrect entry!  Please try again"
130 GOTO 100
140 PRINT "The sum is ";X + 100
150 END
```

As your programs become more involved, it will become increasingly necessary to utilize logic charts.

Exercise Sheet #1
The First Eight Commands

Please create the programs to do the following activities **along with a logic chart**.

1. Have the user enter a number and print this number to the screen.

2. Have the user enter his/her first and last names and print these names in two ways:
   - The first name on one line with the last name on the second line
   - The first name followed by a space, followed by the last name all on the same line

3. Have the user enter a number less than 50. Test the number for correct range and if it is OK, print a message saying “The numbered entered was “<*the number entered*>. If it is not within range, print a message saying “Number out of range. Try again.” and have the user enter another number.

4. Have the user enter a number within the range of 10 through 20. Check to be sure it is within range and if so, print the sum of the number plus 100. If not, beep him/her, print a message and have the user enter another number.

Exercise Set II
The First Eight Commands

Please create the programs to do the following activities **along with a logic chart**.

1. Remember that the upper carat (^) signifies exponents. Have the user enter a number as well as his/her first name. Create the following output to the screen:
   Hello <*entered name*>**. Your number squared is** <*entered number raised to the second power*>**.**
   Note: Don't forget to include periods for your two sentences. Utilize the ***PRINT*** command to accomplish this.

2. Create an application that will:
   - Allow the user to enter the width and height of a square
   - Print the area of that square

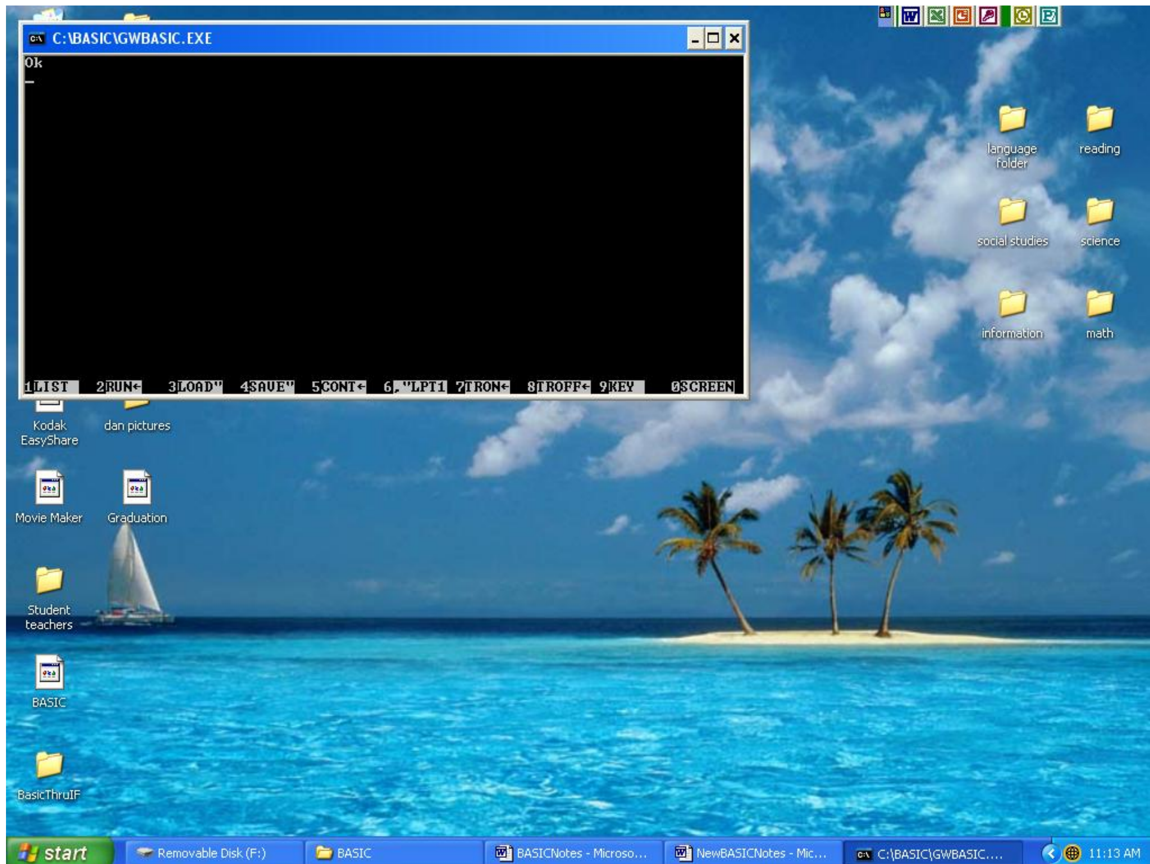
3. In mathematics, factorial means multiplying a number by the first number below it and then multiplying this product by the next number below it until you reach 1. For instance, 5 factorial is 5 * 4 * 3 * 2. Create an application that has the user enter a number greater than 1 and less than 15, and print its factorial representation along with a statement explaining what you are printing.

Instruction Set 2

Now that we have "gotten our feet wet" with the way programming works, we will depart a bit from numeric computations into the world of graphics. BASIC offers an easy-to-use graphic set that can be utilized in either graphic or text mode.

SCREEN

This command will change the appearance and modality of the screen. For the following examples, we will use screen setting 2. When loading BASIC in Windows XP, your screen will probably look like this:



By typing the following commands, we can change the screen so that it covers the entire monitor area:
SCREEN 2 <ENTER>

From this vantage point, we can then begin to utilize the entire screen for our creations.

PSET or PRESET

PSET turns a given pixel on or off depending upon the coordinates referenced, and **PRESET** turns a pixel off. The screen can be viewed as a grid of dots called **pixels**. The state of each pixel, either on or off, will determine what is viewed on the screen.

If we consider the screen a grid of 200 pixels high (**0 – 199**) and 600 pixels wide (**0 – 599**), we can then begin to use it as a canvas upon which to create our masterpieces. Below is an example of how we could create a diagonal line on the screen beginning at the pixel 50 places from the top and 100 pixels from the left to the pixel 100 pixels from the top and 150 pixels from the left.

```
10 CLS
20 X = 49 : Y = 99
30 PSET (X,Y)
40 X = X + 1 : Y = Y + 1
50 IF X = 99 THEN END
60 GOTO 30
```

This program will yield the desired line on the screen. It will appear somewhat like the example below. Note that we start at the (49,99) pixel and move through the (99,149) pixel to create the diagonal line.

Turning Pixels Off

Any pixel that has been turned on with the PSET command can be turned off again by using PRESET or by setting the PSET color parameter to zero (0). So, if we look at the example above, we could turn off 10 of the middle pixels by changing the code as below:

```
5 SCREEN 2
10 CLS
20 X = 49 : Y = 99
30 PSET (X,Y)
40 X = X + 1 : Y = Y + 1
50 IF X = 99 THEN GOTO 70
60 GOTO 30
70 X = 79 : Y = 129
80 PSET (X,Y),0
90 X = X + 1 : Y = Y + 1
100 IF X = 89 THEN END
110 GOTO 80
```

Note in line 80, that we have included the color attribute of zero. This changes the pixel color to black, thus making it invisible. Likewise, if we change line 80 to
80 PRESET (X,Y)
we could accomplish the same thing without the use of a color parameter.

PSET STEP or PRESET STEP

These commands turn pixels on or off respectively based upon a relative position from the current position of the screen. For instance, if we wanted to draw a 100 pixel diagonal line starting from an X,Y location of 320,100 going down and to the right 1 pixel at a time, we could use the following code:

```
10 SCREEN 9
20 PSET (320,100)
30 A = A + 1
40 PSET STEP ( X + 1, Y + 1)
50 IF A < 100 THEN GOTO 30
60 END
```

Note that with the "STEP" version of the command, you are referencing a pixel relative to the current pixel being worked with.

FOR … NEXT

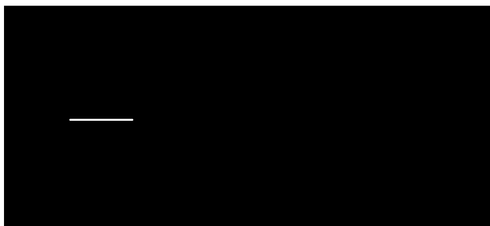
We now see that we can change the value of a variable by placing it on both sides of the equal sign and operating on it. Note in the program on the previous page, we coded line number 40 to say " *X = X + 1 : Y = Y + 1*". This is the same as saying "Let the variable X be equal to its current value plus 1 and let the variable Y be equal to its current value plus 1." After performing each set of additions, in line 50, we check to see if the variable X is equal to 99 and if so, we end the program (IF X = 99 THEN END).

While this is a very handy technique to use it can become a bit cumbersome. After all, most of the program is dedicated to establishing a beginning value for the variables, and manipulating them until they reach a set limit. There is an easier way!

For simplicity's sake, let's change the program so that we do not change the Y value and only change the X value. This will result in a horizontal line located at the established Y location:

```
10 CLS
20 X = 49 : Y = 99
30 PSET (X,Y)
40 X = X + 1
50 IF X = 99 THEN END
60 GOTO 30
```

This program now results in a horizontal line at Y=99 and X varying from values of 49 through 99.



Here is an example of how this new program will appear.

We will now use a FOR… NEXT loop to do the same thing:

```
10 SCREEN 2
15 CLS
20  Y = 99
25 FOR X = 49 TO 99
30 PSET (X,Y)
40 NEXT X
50 END
```

Note the following:

- The "FOR" statement implements the variable name, its beginning value, and its limit. It also checks if the current value has reached the limit.
- The "NEXT" statement sends control back to the "FOR" statement to begin the next iteration of the loop.
- Upon reaching the limit, program control is transferred to the **next line of code following the NEXT statement**.

Would you like to make the line a dotted line? Easy! Let's change the program to only set the pixels for every fifth one.

```
10 SCREEN 2
15 CLS
20  Y = 99
25 FOR X = 49 TO 99 STEP 5
30 PSET (X,Y)
40 NEXT X
50 END
```

Note the **STEP CLAUSE** at the end of line 25. In effect, what this is saying is to use every 5th number within the range 49 through 99 and set it to on.

Here is the resulting program run.

Using a FOR…NEXT loop in the PSET STEP example above, we could draw the same diagonal line as follows:

```
10 SCREEN 9
20 PSET (320,100)
30 FOR A = 1 to 100
40      PSET STEP ( X + 1, Y + 1)
50 NEXT A
60 END
```

Here, the FOR…NEXT loop is used to count 100 pixel additions to the diagonal line.

WHILE … WEND

The WHILE…WEND loop works exactly like the FOR…NEXT loop except that all of the parameters and sentinel must be manipulated manually in the code. In other words, while the FOR statement includes the counter or ***SENTINEL*** variable, its beginning value and its limit along with the optional step clause for incrementing the sentinel, these must all be created and changed individually in the WHILE…WEND loop. Below is the dotted line program from above but using a WHILE…WEND loop:

```
10 SCREEN 2
15 CLS
20 Y = 99
21 X = 49
25 WHILE X < 100
30      PSET (X,Y)
40      X = X + 5
50 WEND
60 END
```

- We create the sentinel variable "X" in line 21 and set its beginning value to 49
- Line 25 checks to see if the sentinel reached its limit
- Line 30 does the actual work
- Line 40 increments the sentinel by 5
- Notice the **indentation** in lines 30 and 40. This makes it easy to know what lines of code are included inside the loop. It is a good programming practice and can be used between the FOR and NEXT statements as well. BASIC does not care how many spaces separate the line number from its associated code.
- While…Wend loops are used when changes inside the loop such as user input or calculations make it impossible to know what the limit of the loop will be.

Exercise Sheet 1
Loops

1. Create **two programs** to print the numbers 20 through 30. Use a different type of loop to do this in each of the two programs.

2. Change the two programs you created in number 1 to print the numbers **30 through 20**.

3. Change the two programs you created in number 1 to print the **even numbers** 20 through 30.

4. Using the correct type of loop, print the product (multiplication) of the numbers 2 through 10 multiplied by 10, quitting when the product exceeds 50.

5. Have the user enter a number within the range 10-20. Using a loop, multiply this number by the **even numbers** 2 through 10. Print each of the results.

6. You have a dump truck that holds 20 tons of sand. When you dump the sand, it exits the truck at a rate of 1000 pounds per minute. Write a program, **using a loop**, to **print the weight** of the sand remaining in the truck for each minute it is being dumped. In addition, **print the number of minutes** it will take to empty the truck.

Exercise Sheet 2
Loops

1. Using a loop, create a program to draw a vertical line from the coordinates X=100 Y=100 to the coordinates X=300 Y=100.

2. Change exercise 1 so that the line you draw is a dotted line with every 10th pixel used.

3. Using the program from exercise 2, create the illusion of movement by turning the pixels on, off, moving 5 pixels to the right of the previous starting point, and recreating the line. Do this 10 times with the program ending after the tenth line draw, before the pixels are turned off.
   Special Note: In this exercise, you will need to slow down execution of the program so that there is a time lag between turning the pixels on and off. Use a simple FOR…NEXT loop from 1 to 1,000,000 and no code between the FOR and the NEXT statements. This, in effect, acts as a delay timer during program execution.

Instruction Set 3

REM

Documenting programs is an extremely important part of the programmer's duties. Think of how hard it is to figure out what other people are thinking. Now think about trying to decipher another programmer's ideas based upon the code he or she has written. Indeed, the programmer who revisits his or her own code after a couple of months or even years, will find it difficult to remember how the logic flowed.

The REM command renders a line of code non-executable and serves to make the line nothing more than a note to the programmer who examines the program. By way of an example, the following code could be the first several lines of a program written by John Smith to examine the grades of his students:

```
10 REM  ***** Student Record Examination  *****
20 REM  ***** Written by John Smith       *****
30 REM  ***** Sept. 5, 2005               *****
```

The additional asterisks are a mere "window dressing" designed to catch the eye of the reader and are not necessary. They are merely a matter of personal choice. The main thing to remember is that **any code written following the REM statement in any one line is not executed**.

SCREEN (Statement)

The SCREEN command sets the resolution for the screen display. Viable options are:
SCREEN 0 = Text mode only
SCREEN 1 = 320 x 200 pixel resolution, 80 x 25 text format
SCREEN 2 = 640 x 200 pixel high resolution graphics, 40 x 25 text format
SCREEN 9 = 640 x 350 graphics, 80 x 25 text format

So, code that places text in a certain place (this information will follow later) or draws graphic images, must include the screen setting first, in order for the output to appear as desired.

```
10 SCREEN 9
20 Y = 175
30 FOR X = 0 to 640
40      PSET (X,Y)
50 NEXT X
60 END
```

This code draws a horizontal line dividing the screen in half in SCREEN mode 9.

COLOR

Now that we have learned some basics about setting screen pixels to an off or on state and have developed a rudimentary knowledge of animation, it's time to put some color into the mix. It needs to be stressed that, in order for color settings to take place, it is necessary to put the screen in the correct graphic mode. Examples of these would be :
SCREEN 0
(This setting includes an option for a border color.)
Or
SCREEN 9

The syntax for the ***COLOR*** command appears below:
COLOR <Foreground color>, <Background Color>
In this syntax, **Foreground** is the **text color** and **Background** is, obviously, the **background color**.

In SCREEN 0, the syntax would be:
COLOR <Foreground color>, <Background Color>, <Border Color>

Below, you will find a chart of color combinations that you can use.

0 = Black	8 = Gray	16 = Blinking Black	24 = Blinking Gray
1 = Blue	9 = Light Blue	17 = Blinking Blue	25 = Blinking Light Blue
2 = Green	10 = Light Green	18 = Blinking Green	26 = Blinking Light Green
3 = Cyan	11 = Light Cyan	19 = Blinking Cyan	27 = Blinking Light Cyan
4 = Red	12 = Light Red	20 = Blinking Red	28 = Blinking Light Red
5 = Magenta	13 = Light Magenta	21 = Blinking Magenta	29 = Blinking Light Magenta
6 = Brown	14 = Yellow	22 = Blinking Brown	30 = Blinking Yellow
7 = White	15 = High Intensity White	23 = Blinking White	31 = Blinking High Intensity White

Foreground and Background Colors (column 1)

Foreground Colors Only (columns 2–4)

For example, the command ***10 COLOR 2,0*** creates a text (foreground) color of **green** and a background color of **black**.

The COLOR setting, once set, **remains in effect until another COLOR statement is executed**.

LINE

The line statement will draw a line from a beginning point designated by its X and Y coordinates to an end point designated in the same way. It also includes a color setting.

Below is an example of code that will draw a cyan diagonal line from the top-left to the bottom right in SCREEN mode 9 on a red background.

```
10 SCREEN 9
20 COLOR 3,4
30 LINE (0,0) - (640,350)
40 END
```

Note that, this occurs because, in SCREEN 9 mode, the graphic set is 640 pixels wide and 350 pixels high. The first number in each set of parenthesis indicates the location in the “X” axis or width and the second number in each parenthesis sets that point’s location on the “Y” axis or height. So, this particular line starts at the left-most, top-most location (0,0) and ends at the right-most, bottom-most location (640,350).

The line color can also be expressed within the LINE statement itself. For instance, let’s say we want to make the line color cyan from the top-left corner of the screen to the middle of the screen and then change it to magenta from the middle of the screen to the bottom-right corner of the screen. In this case, the code would be:

```
10 SCREEN 9
20 COLOR 3,4
30 LINE (0,0) - (320,175)
40 LINE (321,176) - (640,350),5
50 END
```

The “,5” following the line designation in step 40 changes the line color to magenta (number 5 in the foreground color chart).

An even simpler way of doing this is to use the LINE statement’s ability to use the last line referenced as its beginning point and designate only the point to which it needs to go next.

```
10 SCREEN 9
20 COLOR 3,4
30 LINE (0,0)-(320,175)
40 LINE -(640,350),5
50 END
```

Note in line 30 that the last point referenced was 320,175. Line 40 picks up there with a **dash (-)** and moves to the **next referenced point** (640,350).

Drawing Rectangles

You can draw rectangles and can even fill them in using the LINE statement's **B** or **BF** options. For instance, let's say you want to create a cyan rectangle with its upper left corner at the upper left of the screen and would like it to be 50 pixels high and 100 pixels wide. You would then like to draw a brown filled rectangle with the same dimensions beginning at the lower right corner of the previous box.

```
10 SCREEN 9
20 LINE (0,0)-(100,50),3,B
30 LINE -(200,100),6,BF
40 END
```

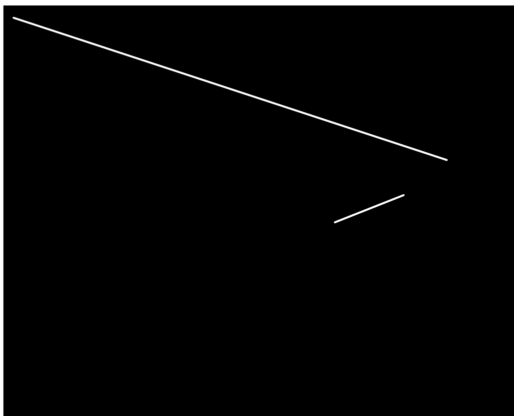
The "**,3,B**" portion of line 20 designates the LINE command creates an unfilled cyan colored box beginning at point 0.0 (the upper left corner of the screen) and ends with the bottom-right corner at point 100,50 (creating a box 100 pixels wide and 50 pixels wide (0 + 100 by 0 + 50). It designates color 3 (cyan) and an unfilled box (B).

Line 30 then says "Starting at the last point referenced (100,50), draw a brown (,6) filled box (,BF) with its bottom-right corner at pixel (200,100).

LINE STEP

Just like the PSET STEP command, the LINE STEP command allows the user to draw a line from the current pixel location to another. Check out this example:

```
10 SCREEN 9
20 LINE (350,200) – (500,150)
30 LINE STEP (50,-50) -  (10,10)
40 END
```



Here, line 20 draws a line from X,Y position 350,200 to position 500,150. Then, in line 30, the first set of coordinates indicates the number of pixels to move from the last location referenced, to an absolute location coordinate of (10,10). The last location referenced was (500,150) and the **relative** movement (50,-50) means, move the beginning of the next line 50 pixels right of (500,150) and 50 pixels up from (500,150). The (10,10) in line 30 is the **absolute** end point of the second line.

BASIC Programming

Exercise Sheet 1
Command Set 3

Note: For all of these exercises, use a remark to place your name and the name of your program at the beginning of the program.

1. Create three (3) separate programs that draw a horizontal line and a vertical line that intersect in the middle of the screen. Each program should use a different **SCREEN** setting.

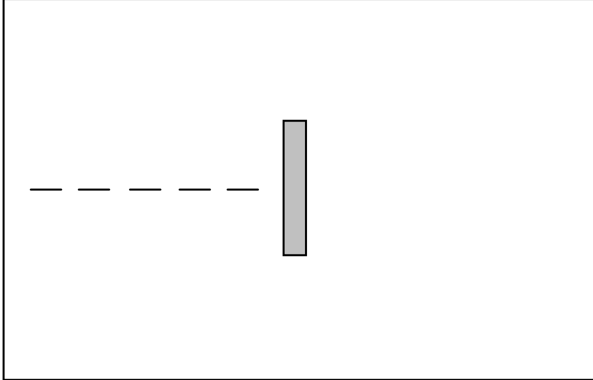
2. Add color combinations to your lines in all three programs in number 1. What do you notice when using SCREEN settings 1 and 2?

3. Create a program in SCREEN 9 that creates an arrow pointing up with the point of the arrow directly in the center of the screen.

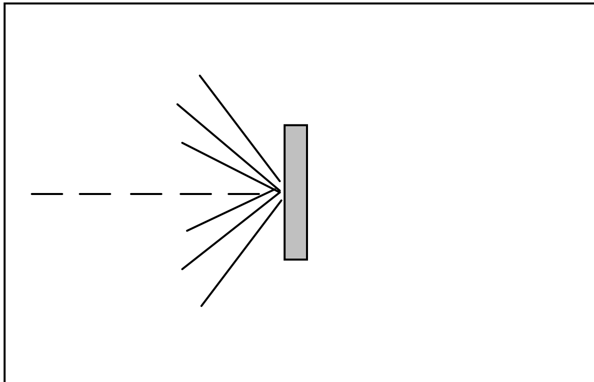
4. Create a program that draws an arrow pointing right and creates the illusion of the arrow moving across the screen until it reaches the horizontal middle.

Exercise Sheet 2
Command Set 3

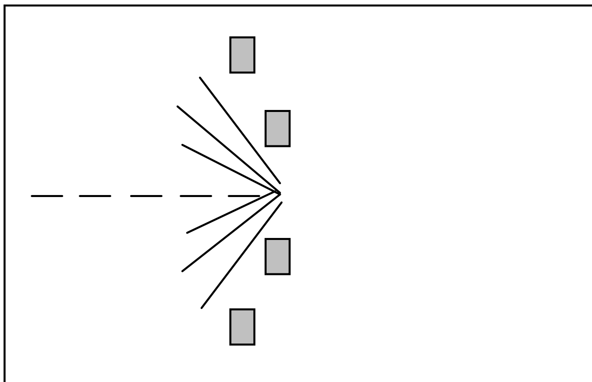
1. Write a program that shows a 10-pixel line moving across the screen, colliding with a colored rectangle at the center of the screen. Use remarks to document your code.



2. Add some explosion effects to number 1 using varying colors. Use remarks to document your code.



3. To number 2, add the effect of the box in the middle of the screen exploding with pieces moving out from the point of impact. Use remarks to document your code.



Command Set 4

GOSUB

You probably already noticed that there is quite a bit or replication involved with writing code, particularly when dealing with graphics. Wouldn't it be nice if you could create a small program that would accomplish a task (such as setting up a graphic on the screen) and simply repeat it as many times as necessary? Well, you can. It can be done by creating a ***subroutine***.

A subroutine is nothing more than a small program that runs inside a larger program. Here is an example of how a subroutine can be used to create three 100-pixel horizontal lines, 50 pixels apart starting at the left edge of the screen in row 100. Each line will be a different color:

```
10 SCREEN 9
20 CLS
30 FOR X = 100 TO 200 STEP 50
40      IF X = 100 THEN C = 4
50      IF X = 150 THEN C = 6
60      IF X = 200 THEN C = 3
70      GOSUB 100
80 NEXT X
90 END
99 REM ********** SUBROUTINE TO DRA
100 LINE (0,X) - (100,X),C
110 RETURN
```

- When a **GOSUB** is encountered, execution of the code moves to the line of code in the GOSUB command.
- Execution of the code continues until the **RETURN** statement is encountered. At that point, execution returns to the **line of code after the GOSUB** (In this case, line 80).

In this example, we had to code the creation of the line one time (in the sub-routine) and simply changed the row number and color prior to calling the sub-routine with the GOSUB command.

Notice the documentation of the subroutine with a remark. This is a good habit to get into as it tells the reader of the program what is being done in that particular sub-routine.

Note the placement of the **END** statement. You will find that it is imperative to place some sort of buffer between your main code and your subroutines in order to minimize the chance of accidentally blundering into a sub-routine without being sent there by a GOSUB command. This will crash your program. Most times, this is accomplished with either a GOTO or an END command.

Sub-routines can be placed anywhere in the program. Most BASIC programmers place them at the end of the program in their own section, preceded with a buffer (see above) and a remark stating the beginning of the sub-routines section.

CIRCLE

You can draw a circle on the screen using the CIRCLE command. This command will allow you to:

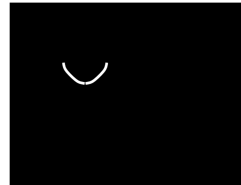
- Designate the X (horizontal) and Y (Vertical) center of the circle
- Designate the radius of the circle in pixels
- Designate the color of the circle
- Designate a start point in **radians** (0 = right, .5π = top, π = left, and 1.5π = down)
- Designate a stop point **counter clockwise from the start point** in radians
- Designate an **aspect** to create an oval (An aspect < 1 results in a “flat and fat” oval and an aspect > 1 results in a “tall and thin” oval.)

For example, the following program creates a circle located at X = 200 and Y = 100 with a radius of 50 pixels colored green.

```
10  CIRCLE (200,100),50,2
```

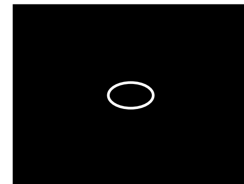
The following example results in a smile-shaped arc.

```
10 CIRCLE (200,100)50,2,3.14,0
```



The following example creates a flat oval centered at (320,175) with a radius of 50 pixels and a yellow color. Notice that, since we want to designate an aspect ratio, we need the aspect property but want to skip the start and stop properties. So, we **place commas to designate a non-setting**.

```
10 CIRCLE (320,175),50,14, , , .5
```



ON…GOTO

Many times it is desirable to give the user options expressed in a menu. For instance, if you wanted to have the user decide what arithmetic operation to perform on two numbers, you might create a menu that looks like this:

1. **Add the Numbers**
2. **Multiply the Numbers**
3. **Quit**

Then, all the user has to do is choose an option (1, 2, or 3). In this situation, it would be easy to simply direct execution of the code to a particular part of the program based upon the value entered by the user. This is where the ON…GOTO command comes in. This command requires a variable to have an integer value starting at 1 and ranging up to the number of options desired. In the above case, it would be 1 through 3. To use ON…GOTO to accomplish the task, the program would look something like that below:

```
10 INPUT “PLEASE ENTER A NUMBER”;N1
20 INPUT “PLEASE ENTER ANOTHER NUMBER”;N2
30 CLS
40 PRINT “1. Add the Numbers”
50 PRINT “2. Multiply the Numbers”
60 PRINT “3. Quit”
70 INPUT “PLEASE ENTER YOUR CHOICE”;C
80 IF C < 1 THEN BEEP : GOTO 70
90 IF C > 3  THEN BEEP : GOTO 70
100 ON C GOTO 150,200,250
150 PRINT “THE SUM IS”; N1 + N2
160 PRINT “ENTER ANY KEY TO CONTINUE” : INPUT X$
170 GOTO 30
200 PRINT “THE PRODUCT IS “; N1 * N2
210 PRINT “ENTER ANY KEY TO CONTINUE” : INPUT X$
220 GOTO 30
250 END
```

Beginning at line 70, we input a value into the variable C. Then, we check to be sure that C is within the proper range in lines 80 and 90. In line 100, the **ON C GOTO 150,200,250** says that if C = 1 then go to line 150. If C = 2 then go to line 200. Finally, if C = 3 then go to line 250.

You may have as many options as you need but they must be integers starting at 1 and moving up from there.

ON…GOSUB

You'll notice in the code above that, after completing the desired calculation, we are required to enter any key (This is done so that we don't immediately go to line 30 and clear the screen before our output can be read.) and then go back to line 30 to clear the screen and display the menu again. Notice that we needed to do this twice – once after the addition and once after the multiplication. We can eliminate this double work by utilizing subroutines to do the work, keeping in mind that the RETURN command will take us back to the line of code following the GOSUB.

Using the **ON…GOSUB** command, works quite nicely here. Take a look at the code on the next page. It is doing the same thing except that it uses subroutines instead of merely separate sections of code.

```
10 INPUT "PLEASE ENTER A NUMBER";N1
20 INPUT "PLEASE ENTER ANOTHER NUMBER";N2
30 CLS
40 PRINT "1. Add the Numbers"
50 PRINT "2. Multiply the Numbers"
60 PRINT "3. Quit"
70 INPUT "PLEASE ENTER YOUR CHOICE";C
80 IF C < 1 THEN BEEP : GOTO 70
90 IF C > 3  THEN BEEP : GOTO 70
100 ON C GOSUB 150,200,250
110 PRINT "ENTER ANY KEY TO CONTINUE" : INPUT X$
120 GOTO 30
149 REM ***** SUBROUTINES *********
150 PRINT "THE SUM IS"; N1 + N2
160 RETURN
200 PRINT "THE PRODUCT IS "; N1 * N2
210 RETURN
250 END
```

Again, we input a value into the variable C and check to see that it is within range. Then, in line 100, instead of simply using a GOTO to move to a different portion of the code, we access subroutines located at lines 150, 200, and 250. The RETURN at the end of each subroutine sends us back to the line of code **after the ON…GOSUB**, in this case, line 110 where we only need to code the instructions once.

Exercise Sheet 1
Command Set 4

1. Have the user enter two numbers within the range 10 through 20. Use a subroutine to check if the numbers are within range and, if not, re-enter the number. Then, print their sum.

2. Change number 1 so that the second number must be within the range 1 through 7. Use the same subroutine to check both numbers out.

3. You are calculating a series of areas for triangles with a constant base of 5 inches and a height that varies from 1 through 4 inches in increments of ¼ inch. The formula for the area of a triangle is **½ * Base * Height**. Use a sub-routine to print a header as :

HEIGHT	**BASE**	**AREA**
------------	**-------**	**--------**

   Then, place the data under the appropriate heading for all of your triangles.

4. Have the user enter two numbers within the range 10 through 50. Use a **sub-routine** to provide the menu to 1) Add the numbers 2) Multiply the numbers 3) End the program. Use **ON GOSUB** to do the desired option. Keep in mind that you can only end the program with the menu and the **menu must be created with a sub-routine**.

Exercise Sheet 2
Command Set 4

1. Write a program to create a red circle that starts at the X,Y coordinates 100,250 and moves upward and to the right 100 moves, increasing the X value by 1 and the Y value by 2 each time it moves.

2. Change the movement to the circle in exercise 1, starting at an X location of 100 and a Y location of 100, moving down, continuing to move it to the right using the same ratio of horizontal to vertical movement. When it hits the bottom, have it bounce back up. **Use a single sub-routine to handle all of the delay loops.**

3. Create a program where the user enters the test scores for 3 students. Enter the data for each student in a loop. Use **On…GoSub** to create a simple bar graph of each student's score at the bottom of the screen, changing location for each bar in each individual sub-routine.

Command Set 5

DATA … READ

Up to this point, the only way for a user to place data into a program is with an INPUT command. In this section, you will be introduced to two other methods. The first one is a DATA … READ structure. Simply put, you can pre-determine what values will be read into the program by placing each value in a DATA statement. Then, the READ command will, in turn from left to right and top to bottom, place that particular value into the program. Here's an example – Suppose you want to merely show the graph of the scores from the last exercise in the previous set and do not want the user to enter them. Below is how it would look:

```
10 DATA 85, 72, 97
20 FOR S = 1 TO 3
30      READ SCORE
40      ON S GOSUB 200,300,400
50 NEXT S
60 END
|
|
|
```

- The DATA statement usually comes at the beginning of the program
- Each data element is separated by a comma **except the last one**
- There can be as many DATA statements in a program as you need.
  10 DATA 85, 72
  20 DATA 97

  The number of elements in each line can exceed the length of a line. It will merely wrap into the next line:

The READ command gets each value in order starting from the first one on line 10 through the last one.

You can read more than one element at a time too. For instance, if you wanted to read a student name and his/her score, you could do it like this:

```
10 DATA BOB, 85, JANE, 72, SUE, 97
20 FOR S = 1 TO 3
30      READ STUDENT$, SCORE
Etc.
```

RESTORE

Using the RESTORE command resets the data pointer to the beginning of the first DATA statement. Including the line number of a DATA statement in the command resets the data pointer to the first data element in that line.

10 RESTORE 20

This command will reset the data pointer to the first data element in line 20.

INKEY$

We are now able to add information to a program via INPUT and DATA…READ. There is still another way to do it. You have probably used programs that used the structure where you are to press any key to continue. Simply put, you are entering data via a single keystroke without having to press the ENTER key. In BASIC, this is accomplished via the ***INKEY$*** command.

To start the explanation of this handy little command, let's take a look at the programming structure used to "Press Any Key To Continue". In our example, we will place this structure inside a loop that simply prints the numbers 1 through 5, making the user press a key before each iteration of the loop can execute:

```
10 CLS
20 FOR X = 1 TO 5
30      PRINT X
40      PRINT : PRINT
50      PRINT "PRESS ANY KEY TO CONTINUE"
60              A$ = INKEY$
70              IF A$ = "" THEN GOTO 60
80      CLS
90 NEXT X
100 END
```

Print the number and skip two lines

Print the user's prompt and set the **string variable** A$ equal to the next key pressed

A set of quotation marks with nothing between them (not even a space) signifies that no key has been pressed. Therefore this line of code is saying "If no key has been pressed, go to line 60. When any key is pressed, A$ is no longer equal to an empty string. Therefore, we do not go to line 60. Instead, we go to the next line of code which clears the screen and increments the loop for the next number.

The INKEY$ structure also works for analyzing menu options. In the next example, we input 2 numbers and offer the user the option to 1) Add them or 2) Multiply them.

```
10 INPUT "PLEASE ENTER A NUMBER";X
20 INPUT "PLEASE ENTER ANOTHER NUMBER";Y
30 CLS : PRINT "PLEASE CHOOSE AN OPTION"
40 PRINT : PRINT "1) ADD" : PRINT "2) MULTIPLY" : PRINT "3)QUIT"
50 T$ = INKEY$
60      IF T$ = "1" THEN PRINT X + Y :END
70      IF T$ = "2" THEN PRINT X * Y : END
80      IF T$ = "" THEN GOTO 50
90      BEEP : GOTO 50
```

T$ = The next key pressed

If the user presses a "1" then print the sum and end. If the user presses a "2" then print the product and end. If the user has not pressed a key, go back to 50 and check for a key to be pressed. If the user presses any key other than "1" or "2" then emit a beep and go back to line 50 to scan for a key again.

You need to remember a couple of things about INKEY$:

- **It will only work with string variables**. That's why, even though the menu options in the example were numbers, the chosen option is stored in a string variable (T$) and when they are referenced in the IF…THEN portions of the code, they are enclosed in quotation marks.
- **Always follow the allowed keystroke responses with code to bounce back up to the INKEY$ statement if no key is pressed.** This is because the code does not wait for the user to press a key as does INPUT. The code continues on scanning for either a pressed key or no pressed key. This should **always go to the INKEY$ statement**.
- **Always follow the scan for no key pressed with an error trap (BEEP : GOTO 50**). Think about it. If an acceptable key is pressed, execution will continue with whatever is to the right of the THEN in that particular IF…THEN statement. If no key is pressed, execution continues with whatever is to the right of the THEN in the IF T$ = "" statement (In our example, we go to line 50). So if a key is pressed and it is not an acceptable option, it has to be wrong and we need to **always go back to the INKEY$ statement**.

Exercise Sheet 1
Command Set 5

1. Write a program that prints the numbers 1 through 100 in sets of 10, clearing the screen prior to printing each set. Have the user **Press Any key To Continue** between each print.

2. Write a program that allows the user to decide if he/she would like to 1) Calculate the area of a circle ($A = \pi r^2$); 2)Calculate the area of a rectangle; 3)Calculate the area of a rectangle; or 4)Quit. The user should only need to **press the key** of his/her response. Documented subroutines should be provided to accomplish each task.

3. Write a program that reads 10 numbers and calculates their average.

4. Write a program that reads 5 student names and a corresponding test score for each printing each student name with his/her score on 5 separate lines.

5. Write a program that reads 5 student names and a corresponding test score for each and provides a menu that allows the user to choose to see 1) The highest score; 2) The lowest score; or 3) The average score. The user should only need to **press the key** of choice and subroutines should be provided to print the result.

Exercise Sheet 2
Command Set 5

1. Create a program that reads radius data, forming concentric circles.

2. Change number 1 so that a color is also read into the program, changing both the radius of the circle and its color.

3. Change number 2 as follows: Prior to printing the concentric circles, provide a menu that allows the user to choose the distance between circles. Make the options 1) 1 pixel; 2) 5 pixels; 3) 10 pixels. The user should only have to **press the key of the desired menu option** to start the circles. Start the circles with a radius of 10 pixels and read the desired colors into the program.

4. Keep the option menu as per exercise 3. This time, create the multi-colored concentric circles in such a manner as to create the appearance similar to a ripple effect in a pond where the circles radiate out ten times, each one disappearing before the next one appears. This rippling effect should happen 10 times and the program should use the same DATA statement for colors each time. After each set of 10 ripple sets, include the question “Do Again? Y/N” where the user simply **presses the “Y” key** to do it again or the “N” key to quit.

Command Set 6

BASIC comes with a fairly complete set of numeric and string functions that can be used to work with these two data types.

Numeric Functions

There are four numeric functions that will be discussed. Three involve rounding numeric values. These are as follows:

FIX

The FIX function truncates or **cuts off any numbers to the right of the decimal point**.

INT

The INT function **rounds all numbers down to the next whole number.**

CINT

The CINT function **rounds to the nearest whole number**.

The table below shows the differences between the three functions:

Number	FIX Function	INT Function	CINT Function
7.89	7	7	8
-7.89	-7	-8	-8
5.21	5	5	5
-5.21	-5	-6	-5

Notice that, for positive numbers, FIX and INT return the same result. The difference occurs with negative numbers. Here, rounding down actually returns a negative number that is lower in value than the FIX result.

For example:

```
10 A = 7.89
20 PRINT FIX(A) : PRINT INT(A) : PRINT CINT(A)
```

Since these commands are functions, they must be followed by a set of parenthesis **( )**. The **parameter** upon which the function is to work is then placed inside the parenthesis.

SQR

The SQR function returns the square root of a number. For example:

```
10 PRINT SQR(25)
```

For this example, the program would return a 5.

VAL

This function returns the numeric value for a string numeral.

10 X$ = “5” : PRINT VAL(X$) = 100 This command returns 105

String functions

LEN

The LEN function returns the length **in characters** of a string. For example:

```
10 X$ = "COMPUTER"
20 N = LEN(X$)
30 PRINT N
40 END
```

The above program would result in a printed output of **8** because **there are 8 characters in the word "COMPUTER"**. Remember that **spaces are included as characters**. In line 10, if we changed the string to "COM PUTER", the resulting print would be 9.

LEFT$

The LEFT$ function returns a given number of characters reading left to right.

```
10 X$ = "COMPUTER"
20 PRINT LEFT$(X$,4)
30 END
```

The above code would result in a printed output of "**COMP**" because these are the **four leftmost characters of the string "COMPUTER"**.

RIGHT$

The RIGHT$ function returns a given number of characters **beginning at the rightmost character and moving to the left**. The result of the function will, however, be displayed **beginning at the leftmost character of the substring**.

```
10 PRINT RIGHT$ ("COMPUTER",3)
```

The resulting printed output from the program line above would result in "**TER**" being displayed on the screen because these are the **three rightmost characters in the string** read left to right. Note that **you can use a string variable or a string literal in the function**.

MID$

The MID$ function returns a substring pulled from inside a string. Its structure is:
MID$ (<String>,<Starting Point>,<Number of Characters to Include>)

```
10 X$ = "COMPUTER"
20 PRINT MID$(X$,4,3)
```

The above program would result in displaying "**PUT**" because these are the characters in the string "COMPUTER" **beginning at the 4th character** and **including 3 characters**.

INSTR

The INSTR function returns the **numeric** point at which a substring appears in a string. It structure is: INSTR (<STARTING POINT>,<STRING>,<SUBSTRING>). The **starting point can be omitted which will start the search at the first character**. If the **substring does not appear in the string**, INSTR will return a **zero**.

```
10 X$ = "COMPUTER"
20 PRINT INSTR(2,X$,"MP")
```

This program returns a **3** because MP occurs at the 3rd character of COMPUTER.

Boolean Operators

AND

The AND operator means that **both** conditions on either side of it **must be true in order for the entire statement to be true**.

```
10 X = 10 : Y = 20
20 IF X > 5 AND Y > 30 THEN PRINT "OK" : END
30 PRINT "NOT OK" : END
```

The condition in line 20 is false because **one of the conditions (Y > 30) is false** because Y is equal to 20. Therefore, the PRINT to the right of the "THEN" will not execute, instead, "NOT OK" will appear.

OR

The OR operator means that **only one of the conditions** needs to be true in order for the entire statement to be true.

```
10 X = 10 : Y = 20
20 IF X > 5 OR Y > 30 THEN PRINT "OK" : END
30 PRINT "NOT OK" : END
```

Now, we would see "OK" on the screen because, even though Y is not greater than 30, X is greater than 5. Thus the entire statement is true.

NOT

The NOT operator **negates** a statement.

```
10 X = 10 : Y = 20
20 IF X > 5 AND NOT (Y > 30) THEN PRINT "OK" : END
30 PRINT "NOT OK" : END
```

Now, we would again see "OK" because X is indeed greater than 5 and Y is **not greater than 30**.

Exercise Sheet 1
Command Set 6

1. Write a program to print the first through the third letters of your name.

2. Write a program to print the third-last to the last letters of your name.

3. Write a program to print the second and third letters of your name.

4. Write a program to print all of the letters of your name **individually** with each letter being printed on a separate line.

5. Write a program where the user enters a string of **at least 10 characters** and enters a substring of **no more than 2 characters**. Have the program print the location of the substring in the string if the substring **occurs after the 3rd character but not after the 7th character.**

6. You are a programmer working for the police department. You have been told that a robbery was committed by a person driving a car with a license number containing a “G3” in it. Write a program to read at least 10 license numbers and the owner names and checks each license number for the existence of this substring. Print out any licenses and the owners for each of these vehicles.

Exercise Sheet 2
Command Set 6

1. Place five (5) **non-integer** numbers in a data statement. Read each of the numbers and print each rounded down, rounded to the closest integer, and with everything to the right of the decimal point truncated.

2. Write a program to calculate **all of the integer factors** for any number entered by the user.

3. Write a program that calculates all of the numbers from 1 through 150 that are divisible by 3 and 5.

4. A perfect square is a number which can be derived by squaring another **integer**. Write a program to print all of the integers from 1 to 200 that are perfect squares.

Command Set 7

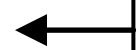
Arrays

In this section, we will discuss a very powerful data type called an **array.** We know that a variable can hold a value or a string but that it can only hold one of these at a time. An array gives us the chance to store more than one value or string in a variable at one time.

Think of a variable as a file cabinet that holds a value (in the illustration below, it holds a value of 10). Imagine that this cabinet contains five drawers, each one of which can hold a different value. Now we are no longer talking about a variable. We are talking about an **array**. In the illustration below, the array contains values of 5, 7, 10, 9, and 17. Each one of these values is called an **element** of the array. Each element is referenced by its **index**. So, in this case, the element in index 1 is a value of 5. The element in index 2 is a value of 7 and so on.

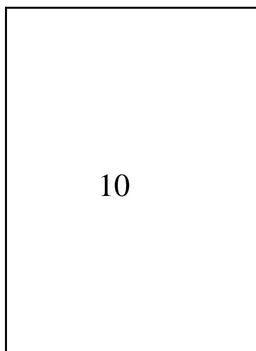
Elements of arrays are referenced in programs with **parenthesis**. So, if the array below was named "X" we could add its first two elements an print this sum using the following code:

```
10 X(1) = 5 : X(2) = 7
20 PRINT X(1) + X(2)
```

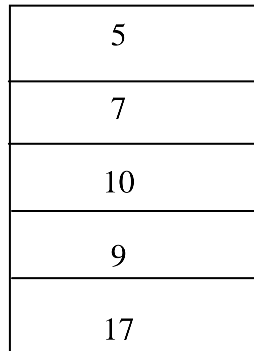


Arrays are created by simply adding parenthesis to the variable name. Data type rules hold for arrays as well. An array that holds strings would need a "$" after its name before the opening parenthesis. For example S$(1).

VARIABLE



ARRAY



DIM

Storage for array element values must be provided in RAM memory. BASIC automatically provides sufficient storage for 10 elements and these small arrays can be created as above. Larger arrays must have storage reserved for them by dimensioning the array.

```
10 DIM X(20) : DIM Y$(15)
```

In the above example, we have reserved memory for 20 single precision numeric elements in X and 15 string entries in Y$.

ERASE

Prior to the advent of the powerful memory modules in use today, RAM memory was at a premium. Arrays can consume a great deal of memory, so it was considered good policy to free up RAM memory after an array is finished being used. The **ERASE** command removes the array from memory, rendering it no longer useable but also freeing up the memory it had used.

```
100 ERASE X, Y$
```

In the above example, the single precision numeric array X and the string array Y$ are no longer useable and the memory they used is now available.

OPTION BASE

Computers have an annoying habit of starting all of their counting functions at zero rather than 1. Normally, this is not a big issue except when the program requires arrays of 10 elements (We would probably number them 1 through 10). This is fine except when we reach element 10. Recall that the computer has reserved memory for 10 elements (It will number them 0 through 9). Therefore, when we hit element #10, we are actually using the 11th element as far as the computer is concerned.

We can eliminate this problem by setting the **OPTION BASE** to 1. The only two legal values for this command are 0 and 1. So we would probably start our program with something like this:

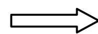
```
10 OPTION BASE 1
```

Now, all arrays will start with a first element number of 1.

Multi-Dimensional Arrays

Recall our earlier example of an array being like a cabinet with drawers that can hold separate values. Now imagine that each drawer has compartments, each of which can hold separate values. We will start by numbering each drawer and then, pull out the first drawer and number its four compartments:

1
2
3
4
5



Drawer 1 Comp.1 (1,1)
Drawer 1 Comp.2 (1,2)
Drawer 1 Comp.3 (1,3)
Drawer 1 Comp.4 (1,4)

Since each compartment is contained in drawer 1, the first element dimension is 1 and each of the compartments (1 through 4) comprises the second element dimension: (1,1), (1.2), (1,3), and (1,4)

Similarly, each of the 4 compartments in drawer 2 would be dimensioned as (2,1), (2,2), (2,3), and (2,4) respectively and so forth for the other 3 drawers.

A multi-dimensioned array **must always be dimensioned** before it can be used. Let's assume our 2-dimension array will hold the four test scores for 5 students. We could write a program such as the following:

```
10 DIM SCORE(5,4)
20 FOR STUDENT = 1 TO 5
30      FOR GRADE = 1 TO 4
40            INPUT "PLEASE ENTER A GRADE";SCORE(STUDENT,GRADE)
50      NEXT GRADE
60 NEXT STUDENT
70 END
```

After running this program, all five students will have four grades entered for each one of them.

Note the use of a **nested loop** in loading the grades into the array. This technique will often come in very handy when working with multi-dimension arrays.

The contents of arrays can be accessed easily to retrieve data. Let's say we want to choose a student and choose one of the four test scores to view. We could augment the program above to look like this:

```
10 DIM SCORE(5,4)
20 FOR STUDENT = 1 TO 5
30      FOR GRADE = 1 TO 4
40            INPUT "PLEASE ENTER A GRADE";SCORE(STUDENT,GRADE)
50      NEXT GRADE
60 NEXT STUDENT
70 INPUT " PLEASE CHOOSE A STUDENT";S
80 INPUT "PLEASE CHOOSE THE TEST SCORE (1-4)";SC
90 PRINT "THE SCORE IS "; SCORE(S,SC)
100 END
```

Note that, by simply using the index number(s) we want, we can retrieve the desired data from the array. If it were a one-dimension array, we would only need one index number.

It is important to know that **once the program is terminated or the power is cut to the computer, all array data is lost because it is stored in RAM memory which is volatile** (clears easily).

Exercise Sheet 1
Command Set 7

1. Create a program that asks the user to enter 3 names. Store these names in an array. After all of the names have been entered, print all three of them.

2. Add the following to the program in number 1: After all of the names have been printed in the order in which they were entered, print them in reverse order.

3. Add the following to the program in numbers 1 and 2: After printing the names in order and in reverse order, have the user choose either the 1st, 2nd, or 3rd name and print only that name.

4. Read 15 random numbers into an array and, **after they have been read into the array**, print the largest and the smallest.

5. Add the following to number 4: After printing the largest and the smallest, print the numbers in order from smallest to largest.

Exercise Sheet 2
Command Set 7

1. You are a grocery store manager. Create a program that stores the contents of two vegetable displays with 3 different vegetables in each display. Use DATA…READ to load the data into the program. Then, display the contents of both displays, compartment by compartment. For example:

Display	Compartment	Contents
1	1	Beans
1	2	Peas
1	3	Tomatoes
2	1	Squash
2	2	Lettuce
2	3	Broccoli

2. To number 1, add the ability to have the user of the program choose the display and a compartment and change the vegetable in it. Then, print the new contents as above.

3. Given the Bingo card below, use a DATA …READ structure to load the numbers into a 2 dimensional array. Then, using the array, print the numbers out in the proper rows and columns to mimic the card.

B	**I**	**N**	**G**	**O**
7	**15**	**40**	**46**	**62**
3	**20**	**37**	**52**	**57**
10	**17**	**FREE**	**49**	**60**
4	**22**	**32**	**50**	**67**
5	**18**	**35**	**55**	**59**

Command Set 8

To this point, we have pretty much been limited in placing text to the upper left corner of the screen unless we used print statements or spaces to move it. Now, we will explore how we can locate text all over the screen and find out what is at any given character point on the screen.

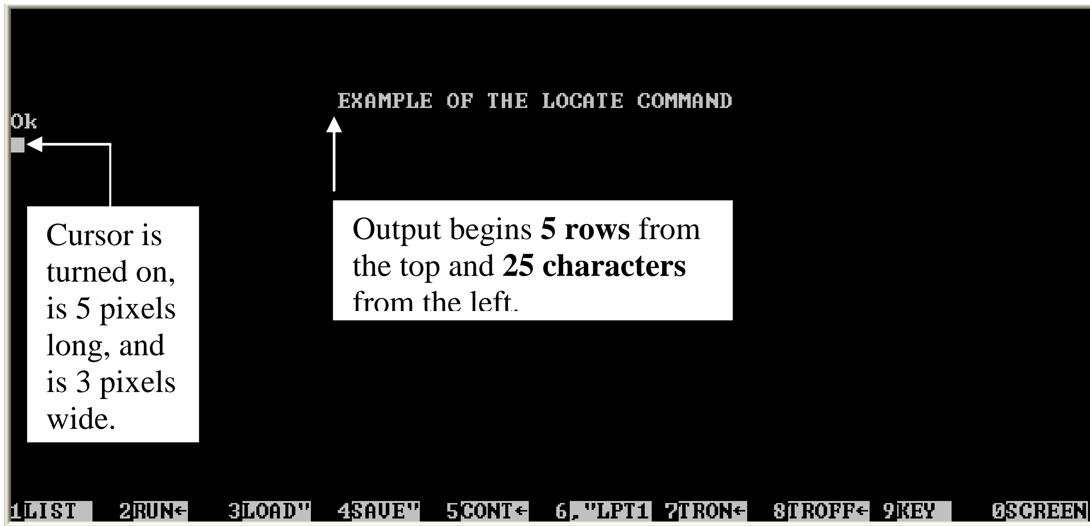
LOCATE

The LOCATE command moves the cursor to a given location on the screen and sets parameters for the cursor. The parameters for the command are as follows:

Parameter	Row	Column	Cursor On/Off	Cursor Length	Cursor Width
Limits	1 - 25	1 - 80	0 or 1	1 - 7	1 - 7

For example, the code below will result in the following output:

```
10 CLS : LOCATE 5,25,1,5,3
20 PRINT "EXAMPLE OF THE LOCATE COMMAND"
30 END
```



CSRLIN

Returns the **current row in which the cursor resides**. It is a **numeric value** and can, therefore be used to add or subtract values, thus changing the cursor location mathematically.

POS (0)

Returns the **current column in which the cursor resides**. It also is a **numeric value** and can be used to add or subtract values, thus changing the cursor location mathematically. As a function, it **must include parenthesis** and **a dummy value** (usually zero) **must be inside of them**.

An example of the use of CSRLN and POS(0) appears on the next page:

This code:

```
10 CLS : LOCATE 5,25,1,5,3
20 PRINT "EXAMPLE";
30 FOR X = 1 TO 5
40       LOCATE CSRLIN + 2,POS(0) - 3
50       PRINT "EXAMPLE";
60 NEXT X
70 END
```

The semicolon (;) following each print eliminates the carriage return/line-feed following the print, so the cursor remains at the **first column following the last character printed**.

Results in this run:

```
EXAMPLE
  EXAMPLE
    EXAMPLE
      EXAMPLE
        EXAMPLE
          EXAMPLE
```

In the loop, adding 2 to the CSRLN value skips one row. Subtracting 3 from the POS(0) value moves the cursor 3 spaces to the left of its current position. Basically, **down 2 and 3 to the left**.

Output begins **5 rows** from the top and **25 characters** from the left.

TAB( )

The TAB() function moves the cursor to a specific column number. This function differs from the POS( ) function in that, the POS( ) function is **relative to the cursor's current column location**. The TAB( ) function, on the other hand, is an **absolute column location**.

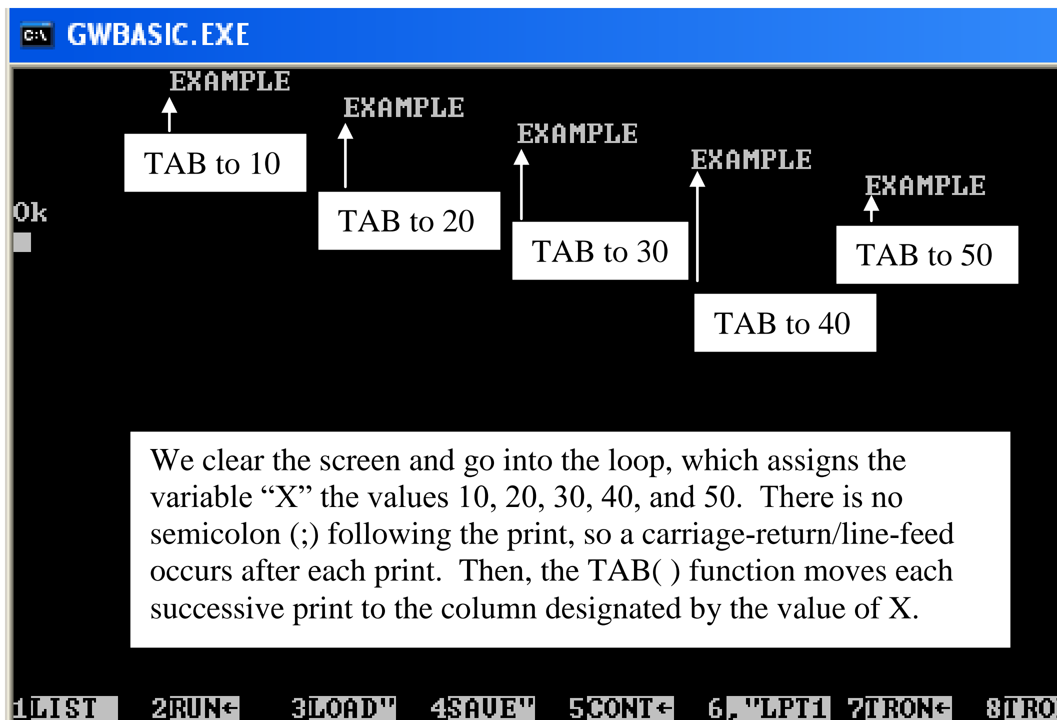
The TAB( ) function **must be used with a PRINT command**.

An example of this function appears on the next page.

This code:

```
10 CLS
20 FOR X = 10 TO 50 STEP 10
30       PRINT TAB(X);"EXAMPLE"
40 NEXT X
50 END
```

Produces this output:



Exercise Sheet 1
Command Set 8

1. Create a program that has the user enter 2 numbers and then, clear the screen and offer the menu to 1) Add them, 2) Multiply them, or 3) Quit. **Center the menu near the middle of the screen** (Screen 9).

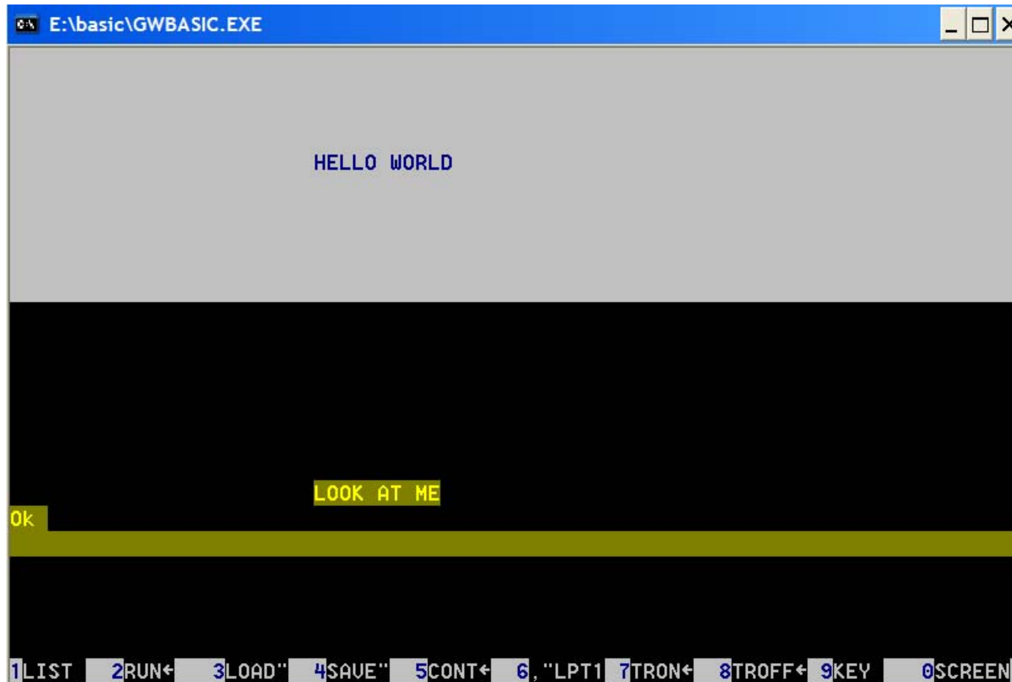
2. Write a program that has the user enter the following: First Name, Last Name, Address, City, State, and Zip Code. Design the program so that each of the categories is printed on the screen and the cursor is located to the right of each of the categories for entry of the information. **Following the pressing of the <ENTER> key, the cursor should move to the right of the next category prompt on the screen**.

3. Create a customized entry form for all of the items listed in exercise 2 but include a graphic of some sort and **locate the entry of the data around the graphic**.

Command Set 9

VIEW PRINT

The VIEW PRINT command allow you to split the **text** screen vertically so that different rows could have separate foreground and background colors and you can control what happens in each part of the screen without changing other parts of the screen.



The Code:

```
10 SCREEN 9 : SCREEN 0
20 VIEW PRINT 1 TO 10
30 COLOR 1,8
40 CLS
50 LOCATE 5,25
60 PRINT “HELLO World”
70 VIEW PRINT 11 to 20
80 COLOR 14,6
90 LOCATE 19,25
100 PRINT “LOOK AT ME”
110 END
```

10 View Print only works in **text mode** so, we had to enlarge the screen with a **SCREEN 9** command and follow with **SCREEN 0** to get back to text mode.
20 Set the area of the screen to work with to lines 1 through 10 (You are limited to a total of 25 lines.)
30 We set the foreground and background colors for these lines to blue and gray respectively.
40 The CLS will set **only the print screen rows to those colors**.
50 We locate to row 5, column 25 (**You are limited to 80 columns on the screen**.)
60 We print “HELLO WORLD” at that location
70 We change to a view print of rows 11 to 20.
80 We change the colors for these rows
90 We locate to the desired row and column in this view print.
100 We print “LOOK AT ME” **without first clearing the screen in those rows**. The color settings **apply only to the text printed**.

CLS0

We have seen that the CLS command, when used while a view print is in effect, will only clear the screen within that view print. The **CLS0** (the last character is a zero) will clear the entire screen no matter what view print is in effect.

ASC ( )

The ASC ( ) function will return the ASCII code number for any character. Below is a table of the standard ASCII code:

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.asciitable.com

The decimal (base 10) numeric code – **under the Dec column** – is matched to its character – **under the Chr column** – by the ASC function. So, if the character was a **capital A**, the ASC ( ) function would return a **65**. The coded function looks like this:

```
10 L$ = “A”
20 PRINT ASC(X$)
```

You could also substitute the string literal in the function – ASC (“A”).

Eventually, people needed more characters to be used by their computers. So, the ***Extended ASCII Code*** was developed. It appears below:

128	Ç	144	É	161	í	177	▒	193	┴	209	╤	225	ß	241	±
129	ü	145	æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
130	é	146	Æ	163	ú	179	│	195	├	211	╙	227	π	243	≤
131	â	147	ô	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
132	ä	148	ö	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
133	à	149	ò	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
134	å	150	û	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
135	ç	151	ù	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
136	ê	152	_	169	_	185	╣	201	╔	217	┘	233	Θ	249	·
137	ë	153	Ö	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
138	è	154	Ü	171	½	187	╗	203	╦	219	█	235	δ	251	√
139	ï	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	_
140	î	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	²
141	ì	158	_	174	«	190	╛	206	╬	222	▐	238	ε	254	■
142	Ä	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	
143	Å	160	á	176	░	192	└	208	╨	224	α	240	≡		

www.AsciiTable.com

Source: www.asciitable.com

CHR$( )

The CHR$ ( ) function is the reverse of the ASC ( ) function. Instead of returning the ASCII number for a character, **this function returns the actual character corresponding to the ASCII code number**. For instance:

```
10 PRINT CHR$(251); 25
```

This code would print √ **25**.

SCREEN

The screen command will **return the ASCII code number for the character located at a given position**. So if a capital A entered at row 10 column 25 on the screen was a correct response to a multiple choice question, we could check the answer like this:

```
100 IF SCREEN (10,25) = 65 THEN PRINT "Correct Answer"
```

Command Set 9
Exercise Sheet 1

1. Write a program to print the ASCII code for the letters “A”,”L”, “i”, and “g”. (Note that the case of the letters is important.)

2. Write a program to print the characters ♣, Π, ∑, and Δ.

3. Have the user enter a string of **at least 7 characters** but **no more than 15 characters**. Check to be sure that the string entered is within the desired parameters. Print the **ASCII values** for the **first character**, the **5th, 6th, and 7th** characters, and the **last character**.

4. Have the user enter a string of **15 characters**. Then, create a heading as follows:
   1 THROUGH 5 <10 Spaces>**6 THROUGH 10**<10 Spaces>**11 THROUGH 15**
   ♦♦♦♦♦♦♦♦♦ ♦♦♦♦♦♦♦♦♦♦ ♦♦♦♦♦♦♦♦♦♦

   Then, print the characters from the string under their heading with each print starting underneath the “H” in the word “THROUGH”. Use the **SPACE$** function.

5. Print the numbers 1 through 50 in rows of 10 with 5 spaces between each number using the **SPACE$** function.

Command Set 9
Exercise Sheet 1

1. Have the user enter his/her first and last names. Then print the following heading "**FIRST NAME <10 Spaces> LAST NAME**". Print the first and last names so that the first letter of each name is aligned under the first letter of its heading **no matter what names are entered** or **how many characters are in the names**.

1. Create a program to have the user enter 3 numbers. Then, print the heading:
   ENTERED NUMBER <10 Spaces> **CURRENT SUM**
   ******************   ***************

   Print each of the numbers **left justified** underneath the "E" in "CURRENT", list the **running sum** of the numbers printed underneath the "N" in "CURRENT".

2. Create a program to do the following: Have the user enter a string of a length 10 – 20 characters. Check this length. Then, have the user enter a second string of 3 characters. Check this length. Search for the existence of the second string within the first string and print the number(s)of the character(s) in the first string where the second string can be found. Use a header as follows:
   ****
   ** FOUND AT   CHARACTER **
   ****
   Print the number of the character(s) where the match was found under the first "R" in "CHARACTER".

3. Create a program to have the user enter his/her favorite band as well as his/her favorite food. Create a header: *****BAND <10 Spaces> FOOD *****. Then, print the entered data under the correct heading. Don't forget **the alignment must work for any band or food**.

Command Set 9
Exercise Sheet 3

1. Print the characters associated with the ASCII code numbers 33 through 57.

2. Create a header as follows: **Character** <15 Spaces> **ASCII Code**
   ********* ***********

   Align the data underneath the first character in each heading column for the ASCII code for the characters "?" and "+". Be sure to use the **SPACE$** function and the **STRING$** function to do this.

3. Input 2 numbers. Create a menu to: 1) Add the numbers or 2) Multiply the numbers. Use **INKEY$** to choose an option and use **ON GOTO** to branch to the appropriate code.

4. Have the user input a string of at least 10 characters. Then, provide a menu to 1) Find a "s" or a "S" in the string, 2) Find a "a" or a "A" in the string, or 3) End the program. Create a header: ***** LETTER** <5 Spaces> **FOUND AT *****
   Use **ON GOTO** to choose the option with the user **pressing the key of the desired option**. Place the upper case version of the letter chosen under the **second "T"** in the header word "**LETTER**" and the point at which the letter is found under the "**N**" in the header word "**FOUND**".

Command Set 10

We now commence the discussion of storing data permanently on disk in order to access it later. It is with these commands that the programmer can create and maintain databases. The first order of business is to decide upon how the database is to be constructed. Then, the programmer must create that structure to support the database. Finally, after the structure has been created, the actual data can be entered into the various database files as required.

For our purposes here, we will create a database for a small produce farm (Happy Valley Farms). Here are the parameters:

- The farm raises the following fruits with the following pricing structure:
  - Apples = $3.50/bushel
  - Peaches = $5.00/bushel
  - Strawberries = $1.25/quart
- The farm also raises the following vegetables:
  - Green Beans = $2.25/pound
  - Sweet Corn = $3.00/dozen
  - Squash = $.50/pound
  - Peas = $1.30/pound
- The farmer would also like to keep track of his customers and what they have ordered as well as how much has been shipped. In addition, he needs to know how much money each of his customers has spent and on what type of produce.

Turning our attention to the actual products, we can start by deciding what it is about each product that we need to know. It seems logical that we would like to know these things:

- Product name
- Price
- Unit of measure

If we look at this scenario, we can begin to see the structure of the **product** database. This database file should contain an entry for each of these three bits of information. These entries are called ***fields***. Every set of fields is called a ***record***.

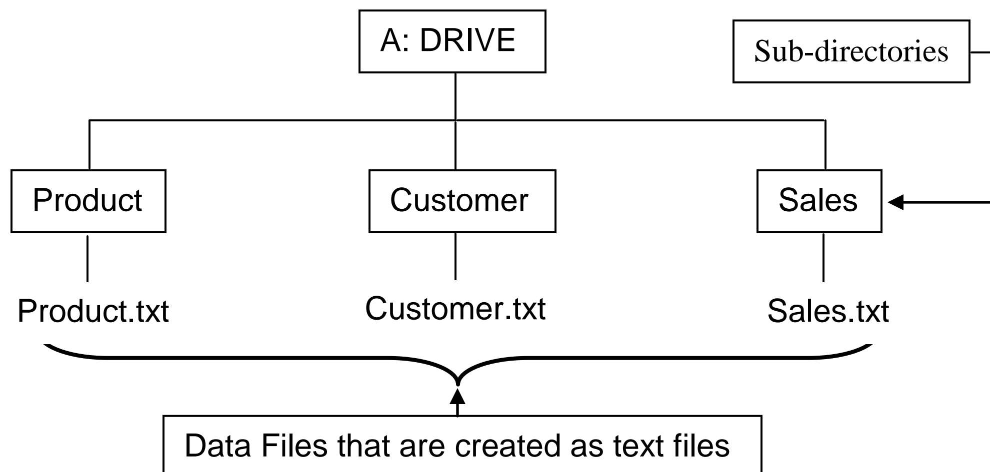
Looking at the **customer** database, we could probably use the following fields:

- Customer first name
- Customer last name
- Customer mailing address
- Customer state
- Customer zip code
- Customer phone number
- Customer email address
- Customer number (Here, we need to construct a system by which each customer has a **unique code number** identifying him/her)

We probably need a **sales** database to keep track of what was sold, to whom etc. Let's include these fields in this database file:

- Product
- Customer number (Customer who ordered the product)
- Requested ship date
- Amount shipped
- Date shipped
- Price for product based upon amount ordered
- Amount paid
- Date paid

Now that we know what we want to keep track of, we can create the ***structure*** of the entire database. For our purposes, we will keep our data on our diskette (A: drive). The entire structure could look like this:



We can create this database structure via DOS commands at the **command prompt** but we can also use BASIC programming commands as well:

MKDIR

This command creates a directory on a given drive. It is short for ***make directory***. It follows the DOS convention of separating each level of the database with a **backslash** (\). So, in order to create the three sub-directories, we can create this program:

```
10 MKDIR "A:\Product"
20 MKDIR "A:\Customer"
30 MKDIR "A:\Sales"
40 END
```

Note that **there can be no other directory at this level in the database with the same name as the directory being created.**

RMDIR

If we decide to remove a given sub-directory, we can use this command. For instance, if we wanted to consolidate our sales and product information into one file in the **Product** directory, we could remove the **Sales** directory by typing this command:

RMDIR "A:\Sales"

Note that **the directory must be empty** (that is, it can contain no other directories or files) **before it can be removed**.

CHDIR

By default, the ***default directory***, which is the **place to which the program looks for its data,** is the place from which the BASIC environment was launched. So, if you load BASIC from your hard drive (Let's say the BASIC folder at the root of the C: drive), your default directory is ***"C:\BASIC"***. That is why, when saving or opening existing program files from your diskette, you will need to include the ***path*** to the file as well as the file name. For example: ***A:\Myprog***

You can eliminate this by changing the default directory using the **CHDIR** command. For instance, to change the default directory to the A: drive we would enter the command: ***CHDIR "A:"***

FILES

Much like the *DIR* command at the DOS prompt, the ***FILES*** command returns the list of files residing in the directory included in the command. For example, if we wanted to see the files that reside in the *SALES* director on the A: drive, we would enter the following command: **FILES "A:\SALES"**

OPEN

Now that we have created the structure for our database, we can begin to create the data files that will hold our data. Files are created by opening them for ***output***. There are two kinds of data files, ***sequential*** and ***random***. Sequential files are accessed by starting at its first record (Remember that a record is a complete set of fields.) and continuing through the file until we find the record we need or until we reach the last record in the file. When a sequential file is opened, it is opened in one mode only. That is, it can input data or write data but not both.

Random files, on the other hand, can have any of their records accessed and do not need to be indexed through in order. They can read or write data.

As an example, let's say we want to create the ***product.txt*** data file with three records. It may look something like this:

Record Names

Record #	Product Name	Price	Unit of Measure
1	Apple	3.50	bushel
2	Peach	5.00	bushel
3	Strawberry	1.25	quart

Files can be opened in several ways. Below is a synopsis of how to open various files:

Sequential File for Output

```
10 FILE$="A:\PRODUCT\PRODUCT.TXT"
20 OPEN FILE$ FOR OUTPUT AS #1
```

As is the case with most commands involving strings, we could substitute the string literal in the OPEN command:

```
10 OPEN "A:\PRODUCT\PRODUCT.TXT" FOR OUTPUT AS #1
```

At this time, the file is created. If it already existed, it is deleted and recreated as an new empty file.

Sequential File for Append

```
10 FILE$ = "A:\PRODUCT\PRODUCT.TXT"
20 OPEN FILE$ FOR APPEND AS #1
```

OR

```
10 OPEN "A:\PRODUCT\PRODUCT.TXT FOR APPEND AS #1
```

At this point, the file is opened with the file pointer at the end of the last record so that additional records can be added to the file.

Sequential File for Input

```
10 FILE$ = "A:\PRODUCT\PRODUCT.TXT
20 OPEN FILE$ FOR INPUT AS #1
```

OR

```
10 OPEN "A:\PRODUCT\PRODUCT.TXT" FOR INPUT AS #1
```

At this point, the file is opened so that records can be read into the program.

Random File for Output or Input

Random files are a bit different in that, not only are they opened in read/write mode, that is, able to read data into the program or write data to the file, but they also require additional information. They need the following as well:

- The length of each record calculated by adding the number of characters in all of the fields contained in the record. For instance, if we decided to allow 10 characters for the product name field, 6 characters for the price field, and 10 characters for the unit field, the length of each record is 26.
- Following the OPEN statement, we then need to designate these field lengths with a FIELD statement. In the case as explained above, we would use this FIELD statement:

  ```
  FIELD, #1, 10 as P$, 6 as D$, 10 as U$
  ```

Putting it all together, we would open this file for random access as follows:

```
10 FILE$ = "A:\PRODUCT\PRODUCT.TXT"
20 OPEN FILE$ FOR RANDOM AS #1 LEN=20
30 FIELD #1, 10 AS P$, 6 AS D$, 10 AS U$
```

File Number

It can be seen that each time a file is opened, it is given a number. In all of our examples, the files were given a ***#1*** label. By default, you are allowed to open two files at the same time in any mode you desire. You **cannot**; however, open a sequential file for both output or append and input at the same time. In other words, the only type of file that accommodates both data writing and reading at the same time is the RANDOM type of file.

Below is an example of how to open two sequential files at the same time:

```
10 OPEN "File1.txt" FOR OUTPUT AS #1
20 OPEN "File2.txt" FOR INPUT AS #2
```

In the above example, ***File1.txt*** was opened with the record pointer positioned at the beginning of the file. In addition, if this file already existed, it was deleted and recreated with all of its old data destroyed. This file can store data gathered by the program. ***File2.txt*** was opened for input so that data can be input from it and used by the program.

Writing Data to Sequential Files

After opening a file for output or append, we need a way to add records to it. This is done using the ***WRITE#*** command where the number sign (#) designates which file is to have the data written to it. Below is an example of a program to create the data file "PRODUCT.TXT" as it appears opened in Notepad (To the right):

```
10 OPEN "PRODUCT.TXT" FOR OUTPUT AS #1
20 INPUT "PLEASE ENTER THE PRODUCT";P$
30 IF P$ = "NONE" THEN GOTO 80
40 INPUT "PLEASE ENTER THE PRICE";P
50 INPUT "PLEASE ENTER THE UNIT";U$
60 WRITE #1,P$,P,U$
70 CLS : GOTO 20
80 CLOSE #1 : END
```

```
"APPLE",3.50,"BUSHEL"
"PEACH",5,"BUSHEL"
"STRAWBERRY",1.25,"QUART"
```

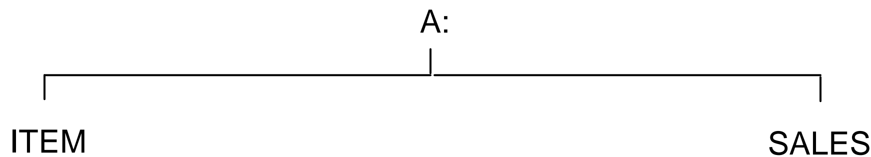
Reading Data Into a Program

After the file is open for input, the data can be read as per the following example:

```
10 OPEN "DATA\PROD.TXT" FOR INPUT AS #1
20 WHILE NOT EOF(1)
30      INPUT #1,N$,P,U$
40      PRINT : PRINT N$;" cost $";P;"per ";U$
50 WEND
60 CLOSE : END
RUN

APPLES cost $ 3.5 per BUSHEL

PEACHES cost $ 5 per BUSHEL

STRAWBERRIES cost $ 1.25 per QUART
Ok
```

Command Set 10
Exercise Set 1

1. Format your diskette and create the following file system on it:



2. Write a program to create the file "ITMDATA.TXT" inside the ITEM directory. This data file will track the following information about the items produced: **Item Name, a 5-digit item number,** and **Price.** Here is the data you need to store in the ITMDATA.TXT file:

Name	Item Number	Price
Table	TB357	$275.95
Chair	CH592	$57.53
Stool	STO79	$23.47

3. Create a program to input the data from the file you just created and print out all of the contents as follows:

ITEM NAME	ITEM #	PRICE
Table	**TB357**	**$275.95**
Chair	**CH592**	**$57.53**
Stool	**ST079**	**$23.47**

Be sure to include the following:

- When inputting data from the file, do not allow the program to go beyond the end of the file.
- Line up your data underneath the appropriate header.
- Format the price with a dollar sign and 2 digits right of the decimal point.
- **Do not erase your data file when finished**.

Move on to Exercise 2

Exercise 2

You are writing a program for the sales department to use when taking orders over the phone or via the Internet.

1. Using ***VIEW PRINTS***, create a form that appears as below”

First Name:	Last Name:
Item #:	Quantity Sold:
Item Name:	Price / Piece:
Total Price For This Sale:	
Choose an Option: **1) Add Another Sale** **2) Quit**	

- Each view port on the screen should be a different background color with a foreground color easily read.
- The middle and bottom view ports **should be blank to begin**.
- The cursor should **automatically move** from a position right of the colon for the names, item #, and quantity sold. **After this information has been entered**, the Item Name and Price / Piece data should automatically be placed to the right of the colon for each of these items.
  - At this point, the “Total Price” label in the middle port should appear along with the calculated price for the sale. The bottom option port should also display its options
  - A customer number should be generated (1st 3 letters of first and last names and a number starting with 1 and incrementing **if different customers have similar first 3 letters of both names**.
- When the “Add Another Sale” option is chosen (press the key):
  - The following data should be written to the “SALDATA.TXT” file in the SALES directory:

First Name	Last Name	Item #	Pcs. Sold	Total Price	Cust. #

  - The field information in the top port should return to its original state with no entered data and the middle and bottom ports should become blank again.
- When the “Quit” option is chosen, the data as above should be written to the file and the file should then be closed and the program ended.

KILL and NAME

Let's say that we would like to change the data in a data file. Perhaps the prices have changed or maybe we made a mistake in entering the data. As was mentioned before, since files are created as text files, editing can be done from any simple text editor. But maybe we would like to streamline the process for the user by creating a program to do this.

Let's say that we want to change the price of apples from $3.50 to $3.75. We need to complete these tasks:

1. Write a program to input data from the original file and write it to a temporary file, allowing the user to enter new data when the item is found in the existing file.
2. Then, we need close all files
3. Then, we need to ***KILL*** the original data file and ***NAME*** the temporary file as the original data file

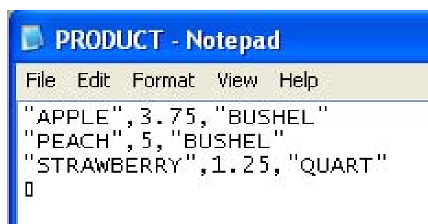
The program appears below:

```
LIST
10 REM ***** EDITING DATA *****
20 OPEN "PRODUCT.TXT" FOR INPUT AS #1
30 OPEN "TEMP.TXT" FOR OUTPUT AS #2
40 INPUT "ENTER THE NAME OF THE ITEM";NM$
50 WHILE NOT EOF(1)
60       INPUT #1,N$,D,U$
70       IF NM$ = N$ THEN GOTO 80 ELSE WRITE #2,N$,D,U$ : GOTO 120
80       INPUT "PLEASE ENTER THE NEW ITEM NAME";NN$
90       INPUT "PLEASE ENTER THE NEW PRICE";NP
100       INPUT "PLEASE ENTER THE NEW UNIT OF MEASURE";NU$
110       WRITE #2, NN$,NP,NU$
120 WEND
130 CLOSE  : KILL "PRODUCT.TXT"
140 NAME "TEMP.TXT" AS "PRODUCT.TXT"
150 END
Ok
RUN
ENTER THE NAME OF THE ITEM? APPLE
PLEASE ENTER THE NEW ITEM NAME? APPLE
PLEASE ENTER THE NEW PRICE? 3.75
PLEASE ENTER THE NEW UNIT OF MEASURE? BUSHEL
Ok
```

Enter the name of the item to change

If the item is found, allow the user to change it's information before writing, otherwise, write the existing information

Close the files, KILL the original file, and NAME the TEMP.TXT file as the PRODUCT.TXT file



The PRODUCT.TXT file now reflects the new value entered for the price of apples.

Exercise 1

Using the data files created in the previous exercises, write the following programs:

1. Read all of the data from the ITMDATA.TXT files and print all of them underneath a heading:

ITEM	ITEM #	PRICE
====	======	=====

   This heading created using a sub-routine.

2. Change exercise 1 so that the user can enter either the item name or its number and print all of the data for this item only underneath the header.

   Include a menu: **1) Search by Item Name**
   2) Search by Item Number

   The user need only press the key of the desired option for the appropriate input to appear, starting the search.

3. Create a program to change data for any given record in the ITMDATA.TXT file.