Introduction: Visual BASIC 6.0

Gary Haggard; Wade Hutchison; Christy Shibata

Microsoft®
Visual Basic®

Gary Haggard, Wade Hutchison & Christy Shibata

Introduction: Visual BASIC 6.0

Introduction: Visual BASIC 6.0
1st edition

ISBN 978-87-403-0341-4

Contents

Introduction

BASIC has come a long way from the teletype interface[1] most current computer users might see in a museum. The language has evolved into an object oriented programming language used in sophisticated applications for PCs. Fortunately, BASIC can still be used to help the nonprogrammer understand what capabilities a programming language has and how all these features are used to solve real problems. This text is intended to help the student who expects their computer usage to consist primarily of using word processors, spreadsheets, presentation packages, and other software for specialized applications to understand what the commands these applications provide with single words and single mouse clicks are actually doing as encapsulated programs.

The text is organized to introduce problem solving with BASIC in a variety of contexts. Chapter 1 gives the needed instructions about how to execute a BASIC program with a minimal discussion of technical systems level ideas. The book uses a template that can be stored as a word processing file and pasted on the code form to provide all the support needed to write and print out both programs and program results. Output of computing results is controlled by a simple double click anyplace on the output form. Using the convenience of storing BASIC programs as word processing files avoids the complications of saving projects and forms that would never be very meaningful to this audience. Using simple highlight-copy-paste procedures is all that is needed to start programming. The same process starting with the code form provides an easy way to save programs for future use. All the procedures used to execute a program are shown with screen captures that guide the user through the systems set up for programming.

Chapters 2–5 take a step by step approach to introduce the output form and how the user controls the placement of output. Then variables and assignment statements are introduced to introduce how the programmer interacts with the computer's memory. Arithmetic operations are next with a discussion of the priority of operations as a guide to how the computer operates. Finally, the logical power of the programming language to ask questions about data and redirect the execution of the program depending on the current state of a computation introduce the fundamental tools of computing.

Chapters 6–7 provide several ways to process multiple pieces of data using the same code over and over. Sentinels and data counts are seen as ways to process unknown numbers of data items with conditional transfers providing a method of ending when the data ends. Once the problem is understood and students practice with the implementation, the power and simplicity of for loops is understood for what it is, an encapsulation of code for controlling repetition.

Chapter 8 introduces random numbers. Simulations of simple dice throwing and slot machine plays lay the groundwork for simple simulations. Scaling of random numbers is also introduced as an important technique for simplifying parts of simulations.

Chapter 9 introduces computer graphics. The drawing of a line segment is developed from the two point formula for a line so readers will see how other functions can be used to draw parts of more complex pictures. The encapsulated code for Line and Circle are explored in detail as well as how to use color and drawing styles inside closed figures. The standard spreadsheet charts for pie charts and histograms are drawn using BASIC and its graphic features.

Chapter 10 introduces arrays and the important idea of file input in BASIC. File input is used so that applications that reuse data in a computation that depends on an earlier computation using the same data can be introduced. The obvious examples are finding the mean and standard deviation of a set of numbers and then asking how many are in a certain range that depends on these two values. Arrays allow a simplification and generalization of code for drawing pie charts and histograms.

The text provides an introduction to programming (without multiple communicating programs) that do give an insight into how functions in spreadsheets, for example, are working in back of the interface consisting of just the name of the function and a list of its arguments.

Each chapter has several features aimed at helping the learning process. The **On Your Own s**ections are short answer questions to help ensure the material presented is understood. The **SYNTAX** boxes are a way to highlight the syntactical aspects that need to be adhered to in any program. **Putting It All Together** is a collection of problems that uses the features introduced in the chapter as well as ideas from previous chapters.

Special thanks to Julia Buffinton for her help with the development and editing of the manuscript.

The authors are interested in helping make this text an effective learning resource. Any errors you find that you relate to us would be greatly appreciated. Any suggestions for making the text better for your purposes would also be appreciated and seriously considered. Just send your ideas or questions via email to vbprogramming@bucknell.edu. All suggestions will be acknowledged and noted at https://www.linux.bucknell.edu/~vbprogramming.

Gary Haggard
Wade Hutchison
Christy Shibata

1 Getting Started

Visual BASIC 6.0 is a powerful object oriented programming language. In this manual, we focus on a small subset of BASIC (shorthand for Visual BASIC 6.0 from here on) features to learn how to write programs that illustrate some important application such as simulations, computer graphics, and file processing.

When BASIC is opened, you see the screen shown in Figure 1-1. Click **Open** and you will see the window for writing programs.

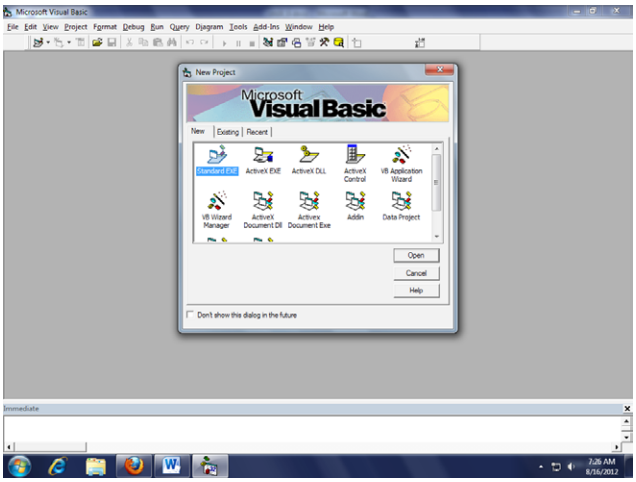


Figure1-1

BASIC will provide two windows for programming. The first window is used to write the code of the program. The second window is called a form that is used to display the output when the program is run. When we choose **Open** with Standard.EXE highlighted we see the two windows as shown in Figure 1-2. The problem at this point is that these are just templates for our use and not windows in which we can actually write code or display output.

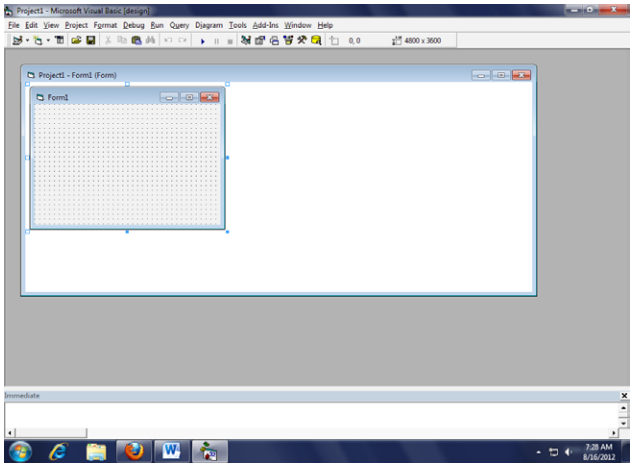


Figure 1-2

The window labeled **Project1** will be used for writing code once we can activate it. The **Form1** window will contain the output from your program. In order to enter programming code, we need to tell BASIC we want the code writing window to be displayed. The first step in getting the code writing window made available is to pull down the **View** Menu on the menu bar as seen in Figure 1-3.

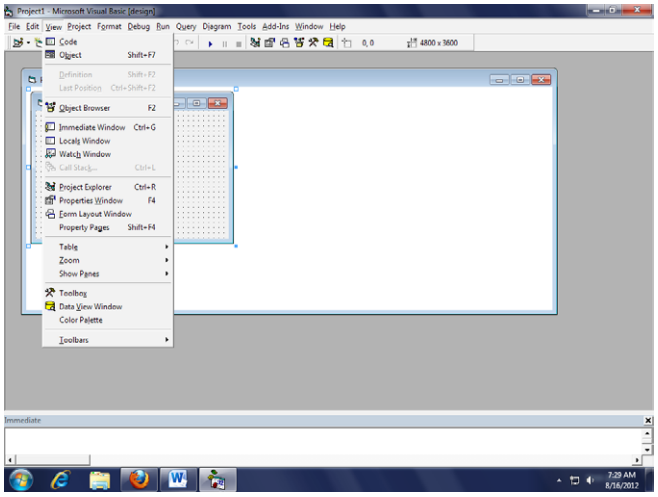


Figure 1-3

By clicking on **Code** we cause the window labeled **Project1 – Form1 (Code)** (see Figure 1-4) to be displayed in front of the other two windows. The two windows in the background will not be used.

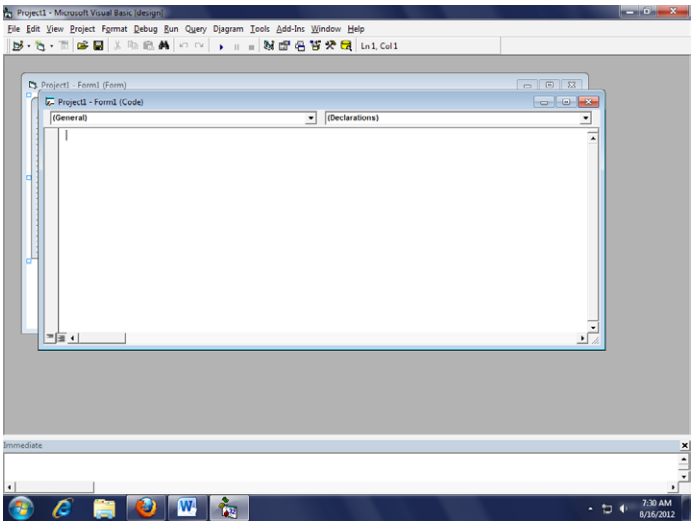


Figure 1-4

The form or window used to write code needs to have its associated output form activated so that when a program is run, there is a form on which to display the output. Activating the output form is a two step process. First we pull down the menu in which (**General**) is shown and highlight the second option **Form** as shown in Figure 1-5.

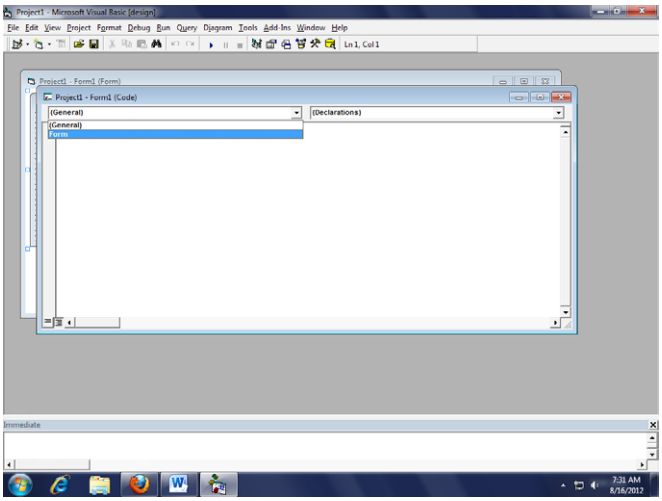


Figure 1-5

When we click on **Form** two lines of code will appear on the code window. **Load** will appear in the second pull down menu as seen in Figure 1-6. Unfortunately, this is not the option in the second menu that we actually need.

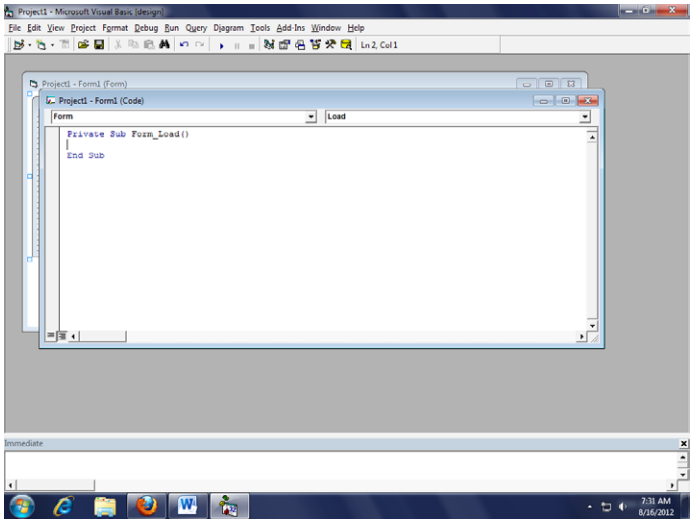


Figure 1-6

Pull down the menu on the right and find the option **Activate** and highlight it as shown in Figure 1-7.

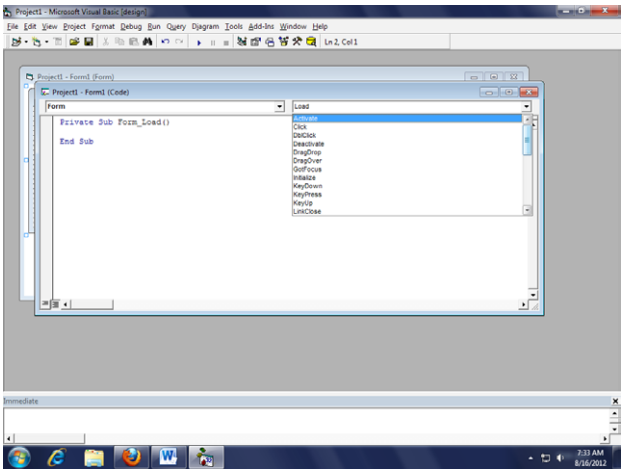


Figure 1-7

When **Activate** is chosen (see Figure 1-8), we see two more lines of code appear in the code window. This code will allow us to write and execute programs.

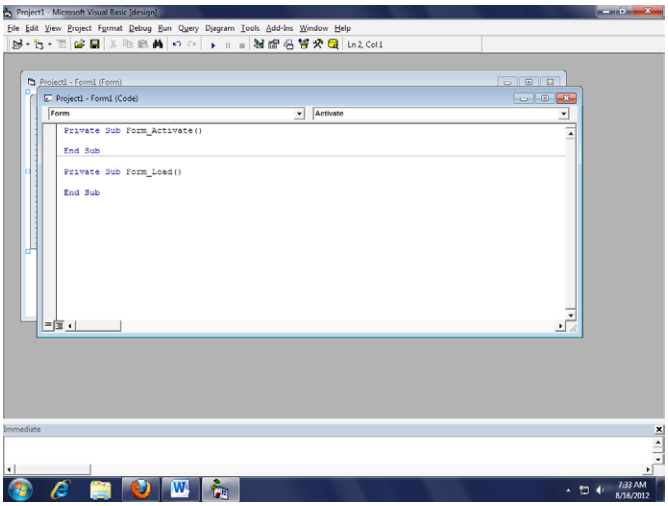


Figure 1-8

You should just write the line:

Print 3

As shown in Figure 19. This is the code needed to print the number 3 on the output screen. We do not have to understand the code at this time as we are simply trying to show what happens at this point and give some motivation for additional code that we will add to all our programs to make the output standard.

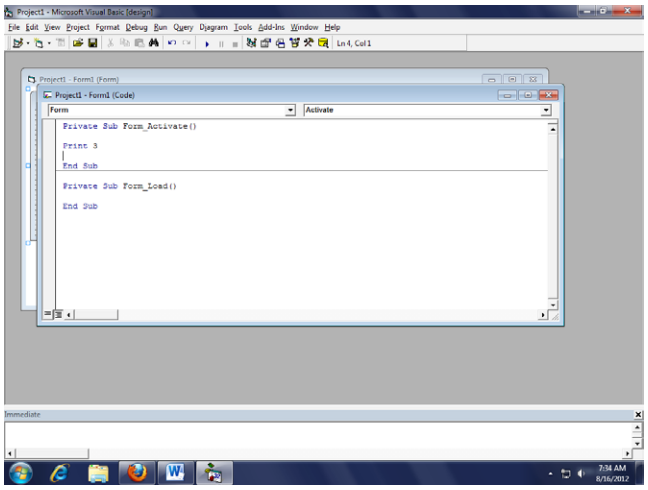


Figure 1-9

To execute a program point the cursor at the small filled in triangle that seems to be standing on its side (see Figure 1-10). You will see a message that says **Start** appear. If you click on that triangle, the program will run.

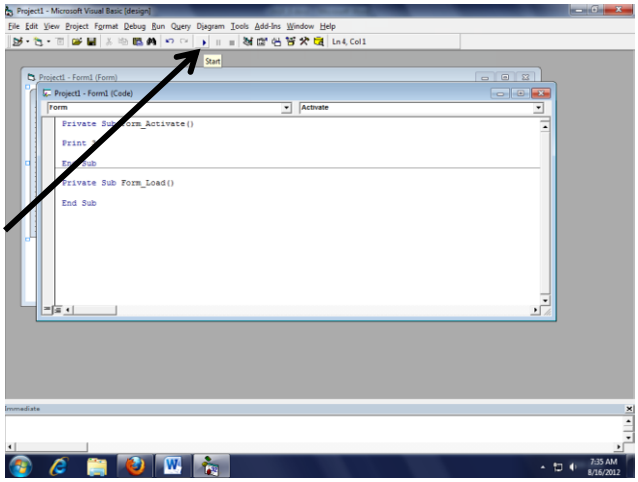


Figure 1-10

The output will be displayed in **Form1** as shown in Figure 111. The problem is that the current size of **Form1** is too small for the programs we are going to write. Consequently, we need to add a line of code to the **Activate** code that will standardize the output form to have room for 25 lines consisting of 80 print positions each.

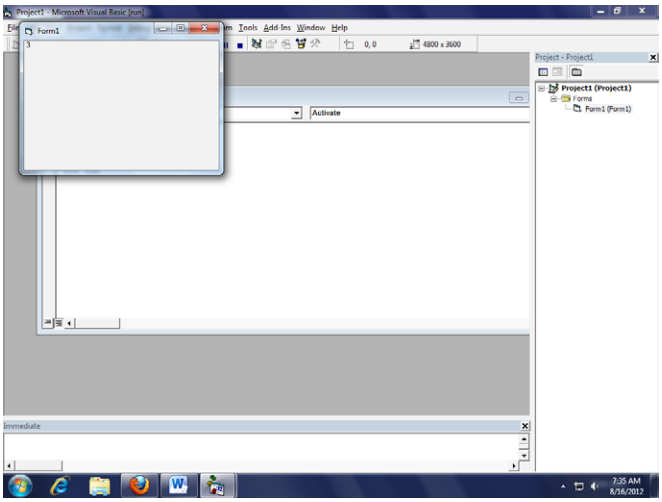


Figure 1-11

Before we can add new code to the **Activate** code, we need to end the current program's execution. We do this by clicking on the X shown in the red box in **Form1** (see Figure 1-12).

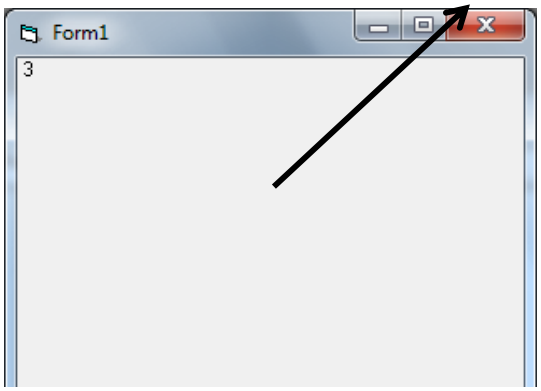


Figure 1-12

In Figure 1-13 you see the additional lines of code needed to standardize the output form. We need to identify the font for writing on the output form as well as its size so we get 25 lines and 80 print positions per line on the output form.

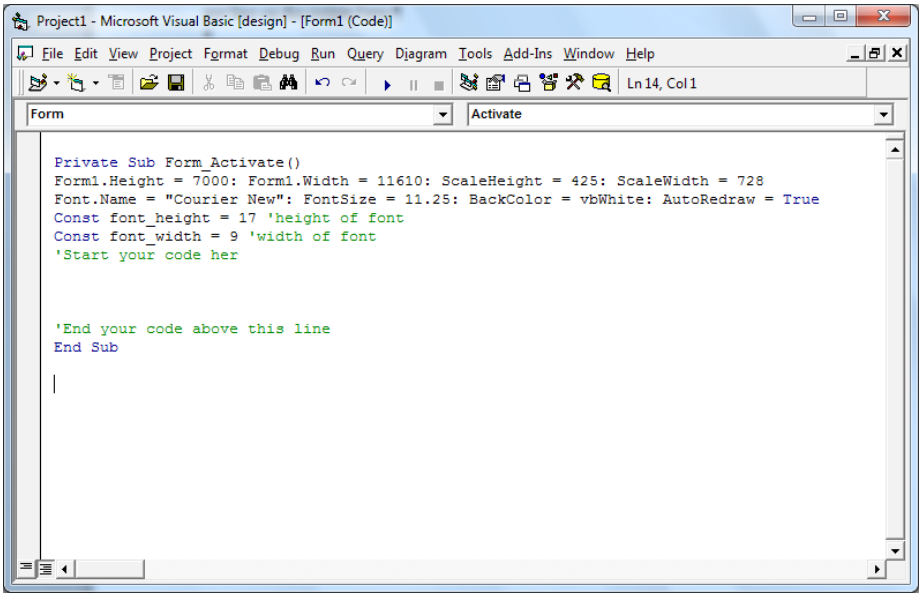


Figure 1-13

Now when you run the program, the output form appears as in Figure 1-14.

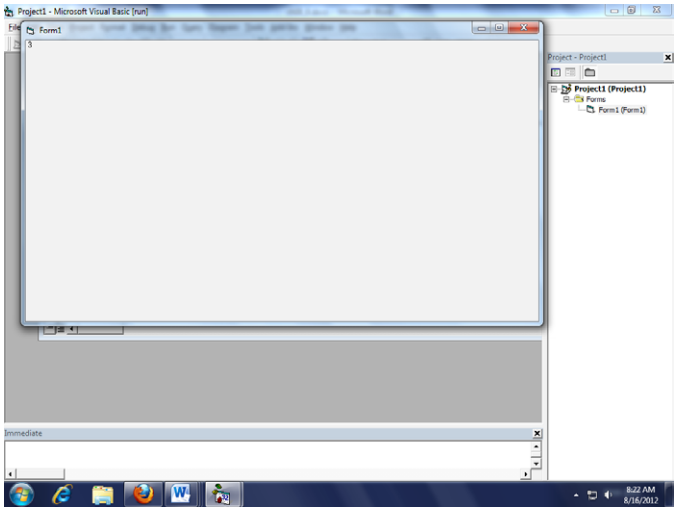


Figure 1-14

One last problem to resolve and all the technical system level detail will be over and done. At this point programs can execute and display output on the output form but we have no way to output the results to a printer. We need to add the following line of code after the **End Sub** code that is associated with **Activate:**

Private Sub Form_DblClick( ) : PrintForm : End Sub

This line of code sets up communication between your program and the printer available in your system. The code should appear as in Figure 1-15.

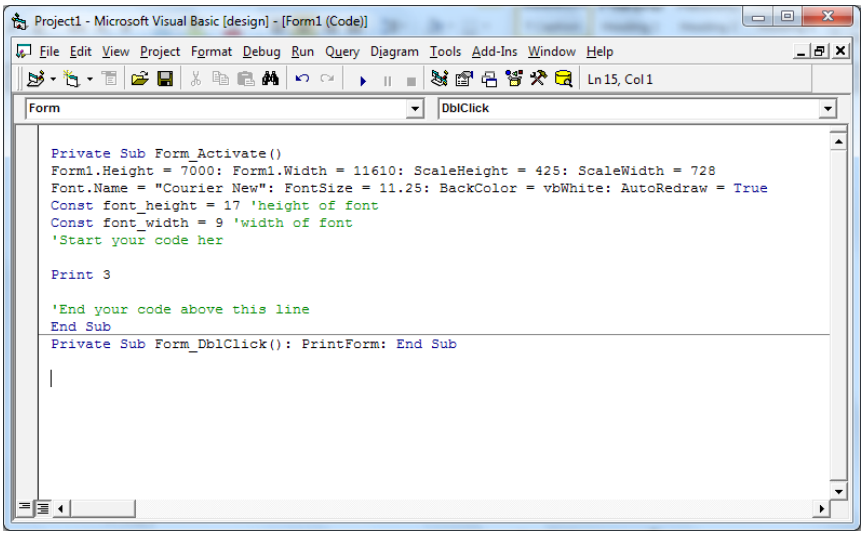


Figure 1-15

The last detail is how we tell the output form that we want its contents printed. This is simply done by double clicking the cursor anyplace on the output form that we see when we run the program (see Figure 1-16).

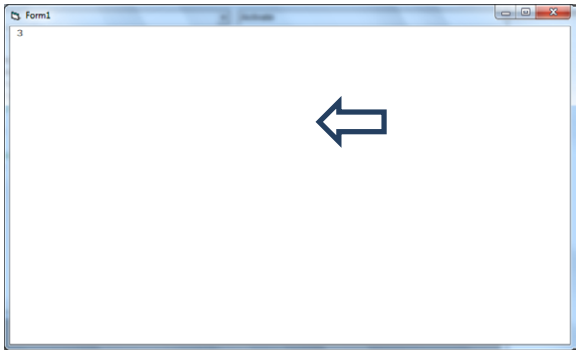


Figure 1-16

1.1 Procedure For Starting

The development of a code template should help you understand what the special code is all about. In practice the way to start a BASIC program is to carry out the first three steps shown in Figures 1-1, 1-2, and 1-3 to see the screen shown in Figure 1-17.

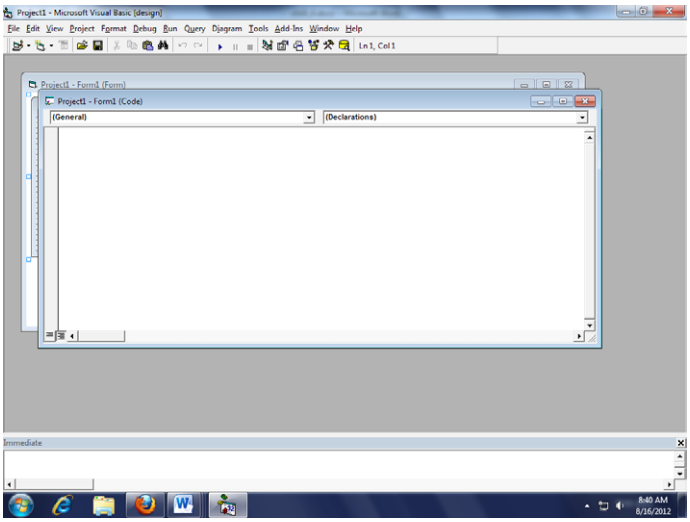


Figure 1-17

Now simply write and save the following code in a word processor so you can copy and paste it to the code form whenever you want to write a program. The code is:

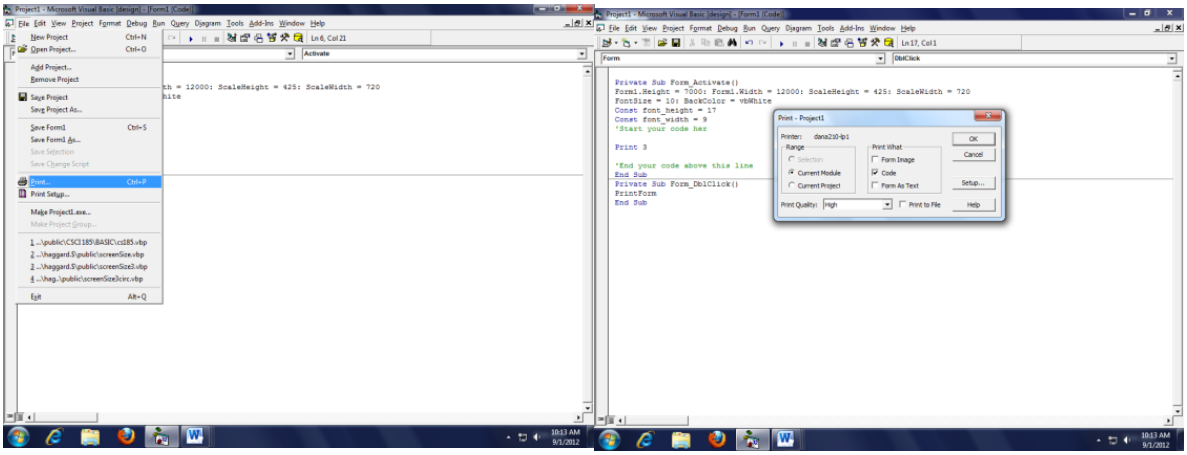
```
Private Sub Form_Activate()
Form1.Height = 7000: Form1.Width = 11610: ScaleHeight = 425: ScaleWidth = 728
Font.Name = "Courier New": FontSize = 11.25: BackColor = vbWhite :AutoRedraw = True
Const font_height = 17        'height of font
Const font_width = 9          'width of font
'Start your code here

'End your code above this line
End Sub
Private Sub Form_DblClick(): PrintForm: End Sub
```

When BASIC interprets these lines as real code you will see that BASIC has drawn a line between the **End Sub** and the **Private Sub Form_DblClick()**. This indicates the separation between the two different blocks of code. You should save this code to use every time you write a BASIC program. All programming will involve entering BASIC commands in the area between "Start your code here" and "End your code above this line."

1.2 Printing the Program

Part of the programming process involves printing out a copy of the code that was written. The printed copy can be taken away from the computer and studied or modified or used to communicate to other programmers what the program does. What is wanted is a printed copy of the code form. To print a copy of the program you first pull down the **File** menu and select **Print** as shown in Figure 1-18 (a). A dialog box will appear on the screen as shown in Figure 1-18 (b). In the dialog box be sure the **Code** option is selected and then click on **OK**. A copy of the program will then be printed.



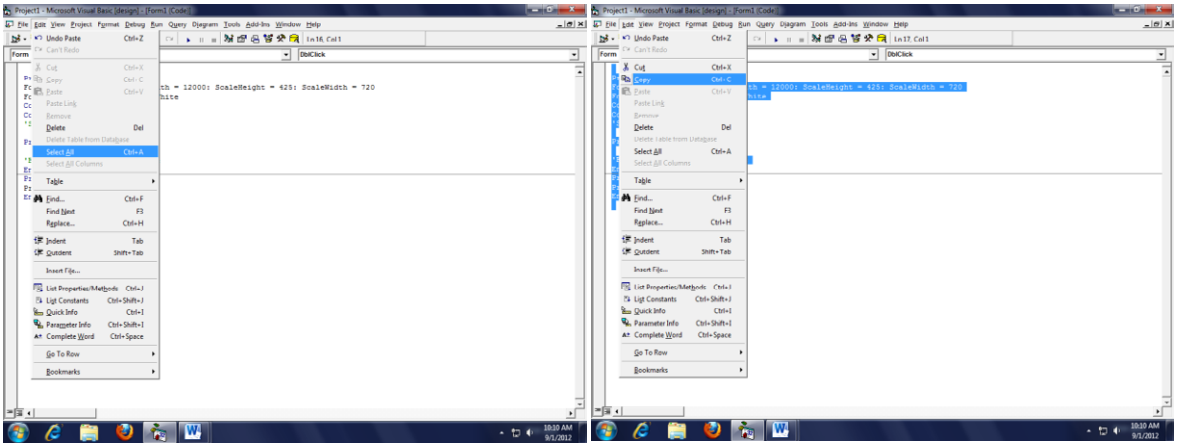
(a) (b)

Figure 1-18

1.3 Saving the Program

When you are finished or partially finished with a programming assignment, you may want to save the code so that you can start again at that point of the programming process. The easiest procedure to use is just to copy the code in the code window and paste it into a word processor and save it. Trying to save a BASIC program in BASIC involves dealing with both the code window and the output form. The complications involved in some aspects of the object oriented nature of BASIC are not necessary for the kinds of programs you learn to write in this book. It is easy to cut and past the code into a word processor document – a program that you are probably familiar with using. When you want to start up the program again, you just reverse the procedure by copying the code from the word processor and pasting it into the code window.

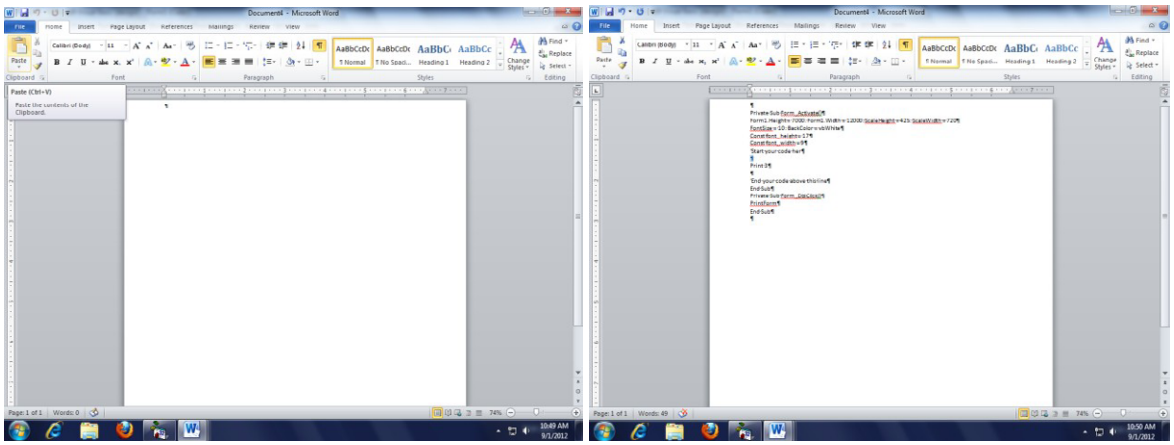
Saving the code in a word processor involves first putting the code on Windows' Clipboard. The first step shown in Figure 1-19 (a) is to pull down the **Edit** menu and click on **Select All**. When the code is selected, pull down the **Edit** menu again and click on **Copy** (see Figure 1-19 (b)). The code is now on the Clipboard and ready to be pasted into a word processor.



(a) (b)

Figure 1-19

You should now open a word processor (we show Word here but any will be similar) and choose the **Paste** option as shown in Figure 1-20 (a). The program file you copied from the code form will now appear in the word processor as shown in Figure 1-20 (b). You must now save the file in your file system.

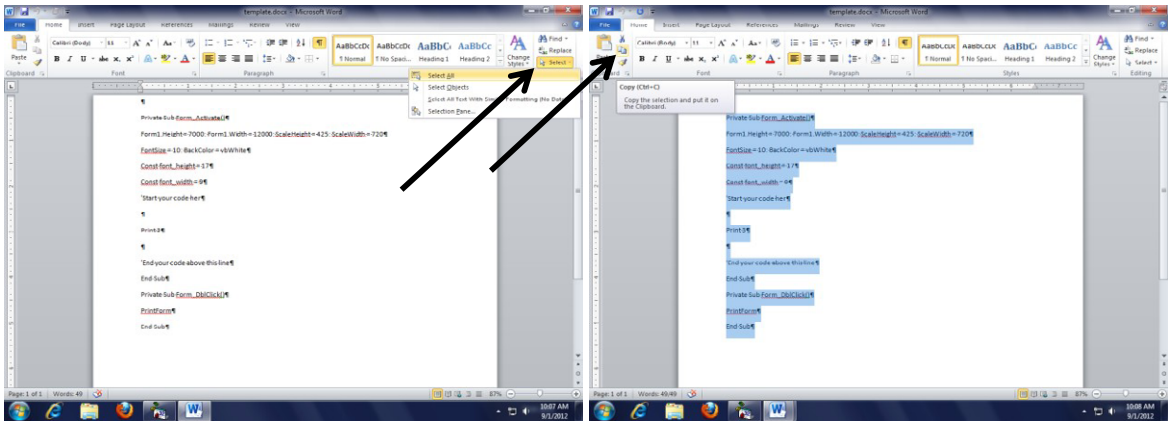


(a) (b)

Figure 1-20

1.4 Reloading a Program

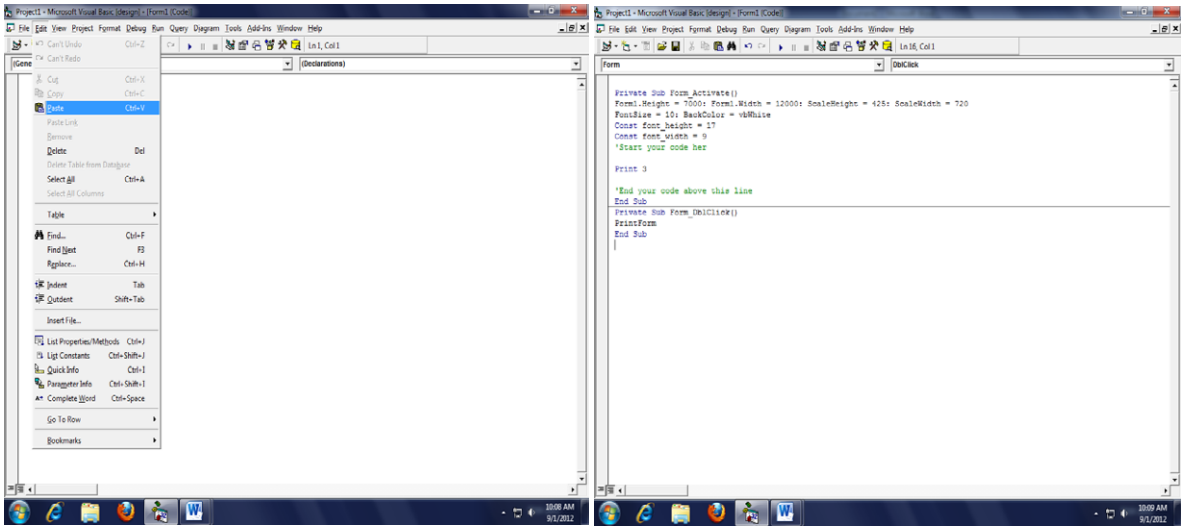
The process of reloading a program for another run or more editing starts by loading the program file into the word processor. The first step is to choose the **Select All** option for the file as shown in Figure 1-21 (a). Once the file is selected you need to click the **Copy** icon near the left side of the **Home** ribbon as shown in Figure 1-21 (b). The file is now on the Clipboard and can be pasted into the code form.



(a) (b)

Figure 1-21

Next you open the **Code** form in BASIC and pull down the **Edit** menu to choose the **Paste** option as shown in Figure 1-22 (a). The program will then be pasted into the code window and you are ready to operate on the program. See Figure 1-22 (b).

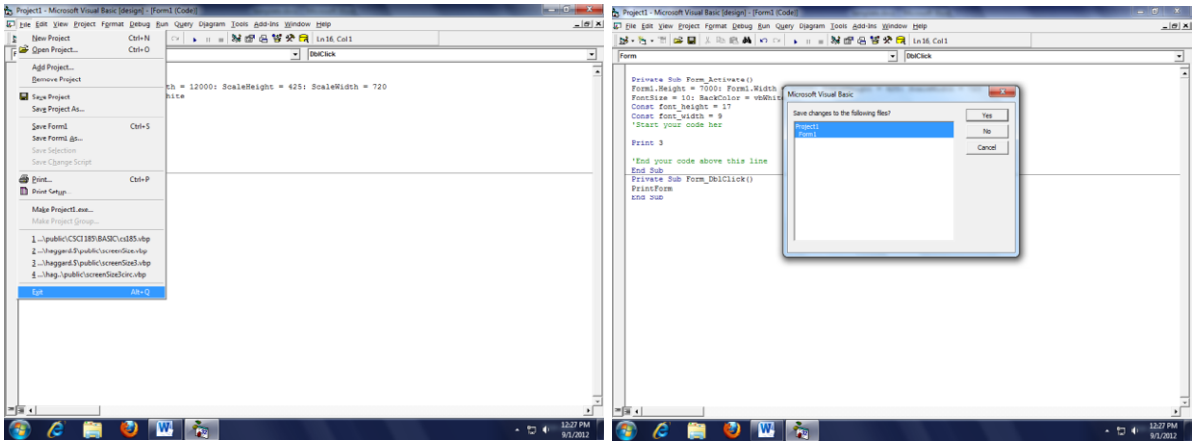


(a) (b)

Figure 1-22

1.5 Exiting BASIC

The process to end a session using BASIC involves two steps. The first step shown in Figure 1-23 (a) involves pulling down the **File** menu and choosing the **Exit** option. Before BASIC quits it puts up a dialog box asking if you want to save the code form and the output form as shown in Figure 1-23 (b). Just click **No** and the programs ends.



(a) (b)

Figure 1-23

1.6 Loading .txt Files

The procedure for saving the code you write for later use can be slightly adjusted so that reloading it can be simplified. If the program is saved as plain text using the .txt file type, there are menu choices in BASIC itself for loading such a file. The procedure starts by choosing the **Edit** menu and then selecting the option: **Insert file …** . You can browse your file system to find the file you want to load. When you **Open** the file, it is pasted into the code window.

Unfortunately, you cannot save text in the code window having type txt. However, if you do save the template as a plain text file, you can load it directly using this option instead of going through the word processor as described earlier in this chapter. Just be sure the single and double quote marks are the ones BASIC uses (to the right of L on the keyboard) in any txt file you intend to load this way.

2 Screen Output

Every program must provide the user or customer with a clear understanding of the results or information generated. The main tool for interacting with the user is a special form used to display the results. How the results are displayed is up to the programmer to decide and make happen. The objective of Chapter 2 is to learn how to control the look-and-feel of the output, i.e., how to use white space and how to position information. For example, suppose you need to display the graduation numbers for men and women students over a three year time span. The output could be displayed as shown in Figure 2.1.

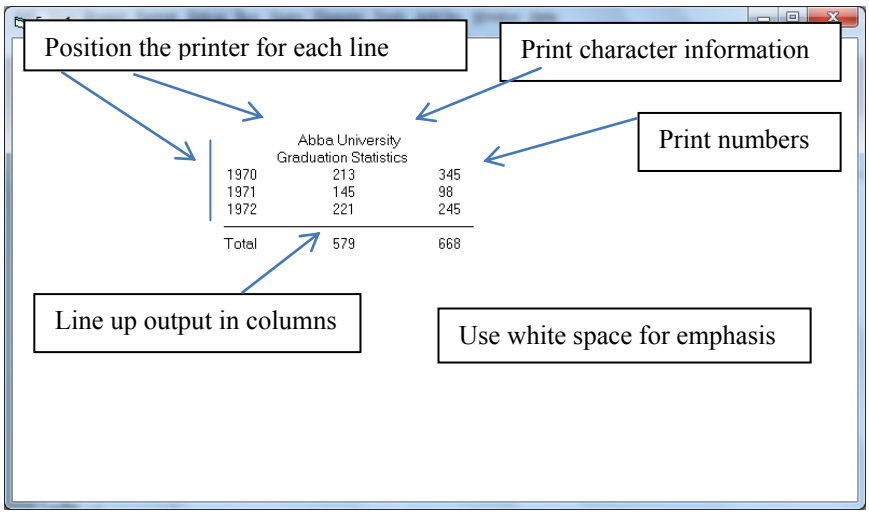


Figure 2-1: Graduation Statistics

Good programming practices and the essential commands and structures used in BASIC to display information are explained in this chapter. By the end of the chapter, you should be able to write programs that control the placement of information displayed on the screen.

2.1 Programming Practices and Conventions

Computer programs are made up of instructions to the machine. In all programming languages, the programming code communicates the step by step actions that are to be performed. Since a computer can only do what it is instructed to do, the programmer must develop an algorithm or procedure for the machine to follow. Only after designing the algorithm can a programmer begin to write the code of a computer program to implement the solution. The rules for writing correct statements for BASIC to execute is called the **syntax** of the programming language. BASIC reserves a number of words to help direct the implementation and execution of the program. These special words like **Private, Sub, Print,** and **Const** are called **keywords**.

In BASIC and all computer programming languages, a program should be easy for both programmers and users to understand. The program logic embedded in the code written needs to be easily understood by a programmer. It is not enough for only the original programmer to be able to understand the code. Other programmers should also be able to understand the code since most programs eventually need modification that is usually not carried out by the code's original programmer. On the other hand, it is equally important that the output produced should be presented clearly and precisely. Users do not want to find it hard to understand what a program's output is all about. Well-written programs themselves are not enough to qualify for a good interface between a computer program and its users, the programmer needs to use a clear and friendly approach in displaying the results of a program in order to fulfill the goals of good computer programming.

2.2 The REM Statement

When you read the lines of a program, you should be able to figure out what the program is doing (or will eventually do) without understanding all the code. (See Figure 2-2. The letters A–F on the left are not part of BASIC but just the way lines of code can be identified when a program is explained.)

A	**Rem** Prints text on the screen that will be
B	**Rem** expanded later to generate a student's
C	**Rem** grade average
D	**Rem** OUTPUT: Text on the screen
E	Cls
F	Print "A Grader's Entry:"

Figure 2-2: An Example Program

In the program in Figure 2-2, lines **A** through **D** tell us what the program will do. This important aspect of programming is called commenting. Comment statements tell someone reading the program what the program will do without requiring the person to read the actual code of the program. Comment statements in a program are non-executable; they are skipped over when the program is run. Comments serve as messages to the programmer and any subsequent programmer who must deal with the code by indicating what is happening in the program. If all the commands of a program directing the computer to do something were deleted, the comments should still give a clear outline of the program's structure. Always use commenting in your programming so that you and others reading your programs will be able to follow and understand what is happening.

In BASIC, commenting is done using the **Rem** statement ("**Rem**" is short for "remark."). Since comment lines are not executable, any combination of text may follow a **Rem** statement. As an example, line **D** of the program in Figure 2-2 is a comment indicating that the executable code in this program displays text on the screen as output:

```
D       Rem OUTPUT: Text on the screen
```

The only syntactic restriction on comments is that each line of a comment must start with the keyword **Rem**. There are certain coding conventions we follow when commenting. These conventions help provide for a uniform program structure. Notice in Figure 2-2 how all the **Rem** statements are at the left margin. Notice also how the executable statements (**E** and **F**) line up. Besides these spacing or indenting conventions, the first comment gives a description of what the program does or what problem it solves. Each line of a comment must begin with **Rem** as seen in A–D. Typically, after the comment explaining what the program does, we use a comment beginning with the word:

```
INPUT:
```

to specify what data is needed for the execution of the program. In the example program, no data is needed so this remark is omitted. We use the word

OUTPUT:

to specify what information results from the execution of the program. Examine the program in Figure 2-3 to see how these commenting conventions are incorporated in a program. (For now, do not worry about the code used to produce the input and output for the program in Figure 2-3.) **Rem** statements can appear any place in a program. Often **Rem** statements are used to indicate what task is performed by a group of statements. Just the **Rem** statements in a program should provide an outline of what and how the program accomplishes a task.

```
        Rem Prompts a user for any word and then
        Rem prints that word on the screen
        Rem INPUT: A word entered by the user
        Rem OUTPUT: Input word on the screen
A           Cls
B       Rem Get input
C           word = InputBox ("Type any word")
D       Rem Display output
E           Print word
```

Figure 2-3: Program Demonstrating Commenting Conventions

SYNTAX

Rem Statement

Rem is a keyword that begins a non-executable statement in a program. **Rem** may be followed by any message to the programmer or about the program. A **Rem** statement is ignored at execution time. **Rem** occurs at the start of a line in a program.

Notice that comments **B** and **D** give an outline of the program's logic even if lines **A, C** and **E** are not well understood yet. As an alternative to starting a line of a program with **Rem**, you can enter a single quote mark at the beginning of the line or anyplace following the statement on the line to indicate the start of a comment.

2.3 The CLS Command

In addition to creating programs that are understandable to the programmer, you should write programs that produce output that is understandable to the user. A screen with the number like 42 appearing on the output form after a program is executed will usually leave the user wondering what it means. One command used to help produce clear output is the **Cls** command. ("**Cls**" stands for "clear screen.") **Cls** is usually used at the beginning of a program as well as after the entering of data and before output is displayed. When executed, the **Cls** command clears the display area of the output form so that the output produced will be displayed on a blank form. After the **Cls** command is executed, the next output will be positioned at the upper left corner of the display area. **Cls** is a keyword and may only be used as described here.

In the program in Figure 2-4, we execute a **Cls** command before the output is displayed so that the output will be displayed on a blank output form. (See Figure 2-4.) Each time the program is executed, the form is cleared before any output is displayed.

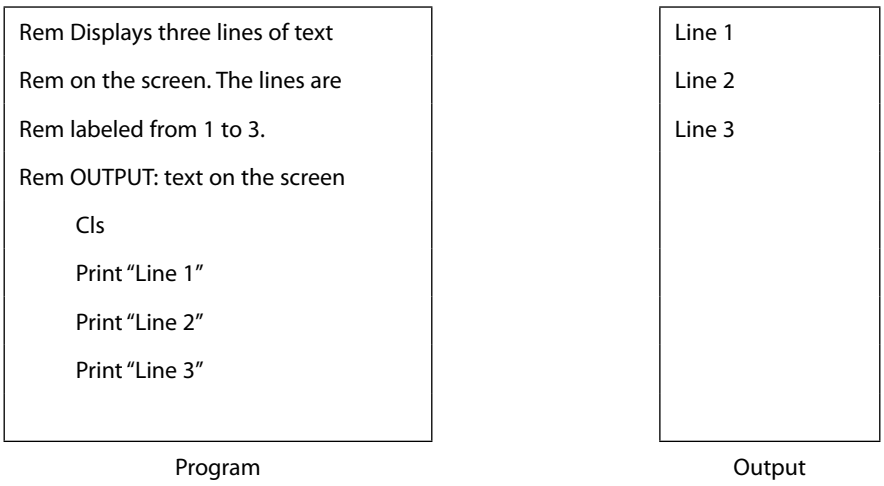


Figure 2-4 – The Program with Cls and the Output

SYNTAX

Cls Command

Cls is a command that causes the output form to be erased and the print head to be placed at the top left corner of the output form screen.

2.4 The Print Command

The **Print** command displays either text or numeric values on the output form. A text message following a Print command must be enclosed in double quotes. We call text in double quotes symbolic constants. (This notion will be defined more carefully later, for now just think of putting an English word in quote marks.) The text message will then be displayed just as it appears within the double quotes. If you want to display a numeric value, no double quotes are used. **The Print command with no output indicated has the effect of skipping a line on the output form because the end of the Print command has a hidden character that causes a carriage return action by the printer.** You can use this idea to put lines between parts of a program's output. Examples of the **Print** command are given below:

```
Print "dog"
Print 12
Print
```

The output of the first **Print** statement is simply the word dog without the double quotes around it. The double quotes simply tell BASIC that what is between the double quotes is a string of symbols. The second print command displays the number 12 on a new line and the third print command simply advances the print position to the beginning of the fourth line without printing anything.

2.5 Print with a Semicolon

With judicious use of the **Print** statement for spacing lines, we can control the line placement of text. To display the numbers 100, 200, and 300 one after another on a single line, we need another mechanism. BASIC uses semicolons and commas to position output within a line. For example, if we use the following **Print** statements:

```
Print 100
Print 200
Print 300
```

we will get the following output on three different lines:

```
100
200
300
```

However, if we place a semicolon at the end of the first two **Print** statements as follows,

```
Print 100;
Print 200;
Print 300
```

we will get the following output:

```
100 200 300
```

A semicolon following text in double quotes or a numeric value in a **Print** statement causes the next output of numbers and/or text to be displayed on the same line starting at the next available print position. For text items there is no space between successive pieces of text. For numeric values a space appears because BASIC wants to be able to put a sign before a number if it is needed. Rather than implementing three separate **Print** statements, as used above, we can produce the same output by using one **Print** statement with semicolons between the items:

```
Print 100; 200; 300
```

This output of this Print statement is again:

```
100 200 300
```

2.6 Print with a Comma

In the BASIC output form, the font and the window size have been organized so that there are 25 rows each consisting of 80 positions across. Each position on a line can hold one character-a letter of the alphabet, a numeral, or a special character. Each row is subdivided into zones that contain 14 positions each (see Figure 2-5).

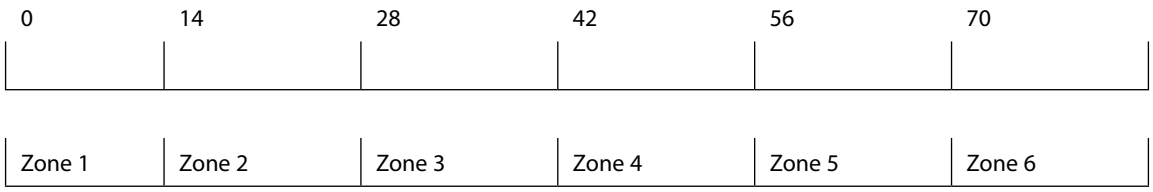


Figure 2-5 – Print Zones

There is a syntax option in the **Print** statement that lets us use this line organization. For printing a list of values, a comma following a value in the list causes the print mechanism to advance to the next empty print zone before printing the next value. For example, the **Print** statement:

```
Print 100, 200, 300
```

will produce the following output:

```
100    200    300
```

Each item is displayed at the left boundary of a separate print zone.

When text messages are printed, the message may be too long to fit into a single print zone. A long message will be printed completely and the comma in the **Print** command following the message signifies that the next piece of information should appear at the start of the next empty **print** zone.

The following are examples of valid Print commands: (There will be four lines of output.)

```
Print 34
Print,
Print “Number”;
Print 34, “Jerry”;
Print 56, 90; 24, 67
Print “Gross”, “Net”
Print
```

When a comma appears at the end of a **Print** statement as in

Print 35, 25,

the next **Print** statement encountered in the execution of the program will have its first value displayed in the next empty print zone on the line used for printing 35 and 25. If the Print statement ends with a semicolon, as in

Print 35, 25;

the next element printed by the program will appear beginning in the next print position on the line on which 35 and 25 were printed.

SYNTAX
Print Command

Print list of elements to print

Elements of the list are separated from each other using commas and semicolons. A comma causes the print head to move to the next empty print zone and the ";" causes the print head to move to the next print position.

EXAMPLE: Print 3 "ABC"; 5

2.7 On Your Own

1. Enter and execute the following three-line programs that demonstrate the use of commas and semicolons in a **Print** command**.**

a)	Print 100,	b)	Print 100;	c)	Print 100
	Print 200,		Print 200;		Print "Text"
	Print 300		Print 300		Print 300

When using the **Print** command to display several items, it is a good idea to include punctuation. If you forget to include punctuation such as a comma or a semicolon, BASIC will automatically include a semicolon. Thus, if you forget to include punctuation between items, a semicolon will be put in for you, and the items will be displayed one after the other on the same line.

The variations of the **Print** command that have been explained are particularly helpful in producing tables.

Example 2-1. Produce a table that shows the graduation numbers of males and females at a university for the years 1970, 1980, and 1990.

SOLUTION: The output required is the table:

	1970	1980	1990
Male	476	470	490
Female	364	465	560
	-----	----	----
Total	840	935	1050

The code that will produce this table is:

```
        Rem     Produce a table that shows the number of
        Rem      males and females who graduated in 1970,
        Rem     1980, and 1990.


        Rem     OUTPUT: Table with information
                Cls
        Rem     Display output
A               Print , 1970, 1980, 1990
B               Print
C               Print "Male", 476, 470, 490
                Print "Female", 364, 465, 560
                Print , "----", "----", "----"
                Print "Total", 840, 935, 1050
```

The program uses commas between items in a **Print** statement to display certain items in consecutive print zones on a single line. Since each piece of output information fits in a print zone, the effect of the **Print** statements with commas between elements is to make the output appear in tabular form. Notice that in line **A**, there is a comma appearing before any information to print. This comma causes 1970 to be printed in print zone 2. Here **B** causes a blank line to appear between the dates and the first set of numbers. In line **C** the program needs to print "Male" in print zone 1 and so the string is listed and then followed by a comma so that the first numeric value is printed in print zone 2 as required.

2.8 On Your Own

1. Write a program to produce the following table that gives information about three states. The area is given in square miles.

State	Capital	Population	Area
Alaska	Juneau	570000	570374
New York	Albany	18058000	47224
Texas	Austin	17349000	261914

Do not output the lines surrounding the table.

2.9 Positioning Output on the Screen

Though we have been concerned with the placement of text in a line, we have not concerned ourselves with any special placement of the text on the output display. The first output of a program has always been displayed beginning in the default position in the upper left corner of the display. For variation in output displays, programs can position output at a specific place on the display form.

The **output form** can be thought of as a grid composed of 25 rows each with 80 columns. The rows are numbered from 0 to 24 and the columns are numbered from 0 to 79. Before each **Print** command BASIC must know on what line and in what column on that line the printing should begin. The variables **CurrentX** and **CurrentY** are the variables (a formal definition comes later, for now just assume BASIC uses these names to keep track of some information it needs) used to know how to position the print head. BASIC uses **CurrentY** to hold the number of the row and **CurrentX** the column in that row where printing should begin. When a Print command is finished executing CurrentX and CurrentY are given values so that the next **print** command will start at the left margin of the next line of the output form. When a program begins to execute, BASIC sets both **CurrentX** and **CurrentY** to zero indicating that printing will start at the top left position of the output form. **CurrentX** is the distance from the left side of the display. **CurrentY** is the distance from the top of the display. A program may override the automatic positioning of the print head and explicitly change the values of **CurrentX** and **CurrentY** to control the location of the next piece of output on the display. The computer measures horizontal and vertical distance on a screen in terms of pixels (the smallest resolution unit on a display device). Rather than think in terms of pixels that are too small to hold a symbol we can see, we have organized the screen to consist of lines that have a fixed height in terms of the number of pixels and such that each print position in a line has a fixed width in terms of the number of pixels. Each row is set to be 17 pixels high and each column is 9 pixels wide. These dimensioning values are chosen so that one character of the font used for output will fit in a box of this size. (See Figure 2-6 for the code in the code template that defines this "box" size.)

```
Const font_height = 17     ' height of font
Const font_width = 9       ' width of font
```

Figure 2-6: Constants for Converting to Pixels

To print beginning at row 19 and column 56, change the values of **CurrentX a**nd **CurrentY** before printing as shown in Figure 2-7.

```
CurrentY = 19 * font_height
CurrentX = 56 * font_width
  Print "Over here!"
```

Figure 2-7: Example of Positioning Output

The output is printed in row 19. The letter "O" is printed in column 56, "v" is in column 57, and so on. (The equal sign is used to set the value of **CurrentX** and **CurrentY**.) CurrentY is set to 19 times the height of a single line and **CurrentX** is set to 56 times the width of a single print location. Now when the print command is ready to execute it will first move the print head to the position on the output form represented by the current values of **CurrentX** and **CurrentY**.

To demonstrate the effect of changing the values of **CurrentX** and **CurrentY**, we will show the output of programs that do and do not change these values. First, refer to Figure 2-8 which shows the output of a program that does not change the values of **CurrentX** and **CurrentY**.

```
Rem     Prints text on the screen that will be
Rem     expanded later to generate a student's
Rem     grade average.


Rem     OUTPUT: text on the screen
        Cls
Rem     Display output
        Print "A Grade Calculation:"
```

```
A Grade Calculation:
```

Figure 2-8: Program that Uses Default Positioning for Output

We now change the values of **CurrentX** and **CurrentY** before the **Print** statement in the above program to change the position of the output text. (See Figure 2-9.)

```
Rem     Prints text on the screen that will be
Rem     expanded later to generate a student's
Rem     grade average.


Rem     OUTPUT: text on the screen
        Cls
Rem     Set output location to be row 4 and column 15
        CurrentY = 4 * font_height
        CurrentX = 15 * font_width
Rem     Display output
        Print "A Grade Calculation:"



                A Grade Calculation:
```

Figure 2-9: Program that Positions Text Output

Notice that in the output of the program in Figure 2-9 the line of text is displayed beginning in column 15 of row four of the display. An important part of programming is to design visually pleasing output displays that readily convey the important information. We will change the values of **CurrentX** and **CurrentY** to produce clear and concise program output with adequate spacing to make the information easy to understand.

SYNTAX

CurrentX and CurrentY

Commands used to position the print head on the output form. The values are given in terms of the pixel structure of the output screen. The next print command begins at that position.

2.10 Summary

In this chapter, we have discussed elementary features of the output command **Print**. We are interested in writing programs that effectively present information. By setting the values of the **CurrentX** and **CurrentY** coordinates and using **Print** and **Cls** commands, we can position text anywhere on the output display. With judicious use of these techniques we can even produce simple pictures. We will continue to focus on using features of BASIC to provide an effective way to present output so that the user readily understands what information is presented. We are interested in well-designed interfaces between the user and a program. We call such interfaces **user-friendly.**

2.11 Putting It All Together

1. Write a **Print** statement to display your name on the screen. Hint: Figure 2-9 is a very good model for this problem. Set the values of the **CurrentX** and **CurrentY** to determine the output location. The output location should be near the center of the display. (Recall that the rows are numbered from 0 to 24 and the columns are numbered from 0 to 79.)
2. Write a program to print your first name at the left margin of line 13 and your last name so that the last letter is in column 79 of line 21.
3. (a) Write a program that prints your first name using a print statement that ends with a comma. Then write a second print statement to display your last name.
   (b) Repeat part (a) but now use a semicolon at the end of the first print statement.

4. Write a program to produce the following table that shows game results for State U's basketball team for the past three years. (This program does not need to change the default values of **CurrentX** and **CurrentY**.)

Year	Overall Record	Wins at Home	Wins Away
2012	19-9	13	6
2013	21-6	14	7
2014	16-10	11	5

5. Write a program to produce the following table that shows coffee sales at the University's snack bar in the library. The table should use print zones 2 through 5 on lines 10 through 15.

COFFEE SALES			
DAY	SMALL	MEDIUM	LARGE
Monday	95	68	51
Tuesday	88	72	46
Wednesday	112	83	47

6. Write a program to produce the following table which shows the sales record of each sales person in a company. (This program does not need to change the default values of **CurrentX** and **CurrentY.)**

Sales Person	Today's Sales	Week Total	Commission
Brown	509.85	2367.98	710.39
Jones	366.91	1982.82	594.62
Roberts	430.11	2284.43	685.33
Smith	399.08	1857.68	557.35

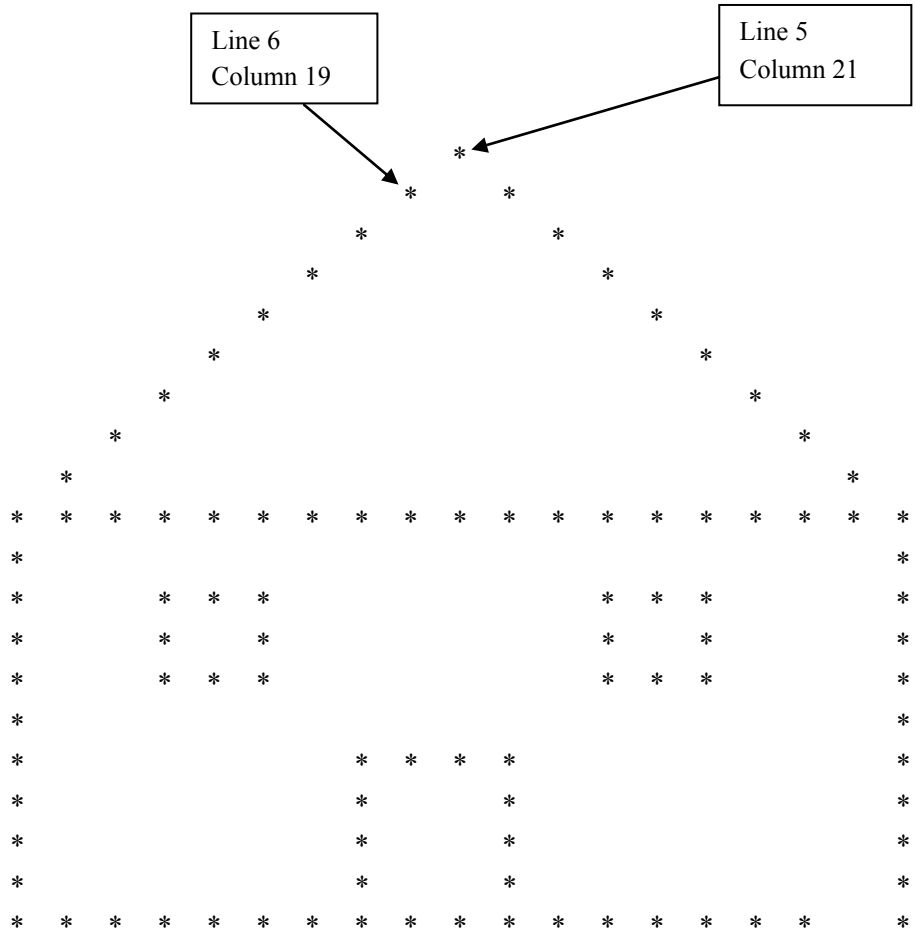
7. Write a program to draw a house centered on the display as shown in the figure below.. Hint: The executable part of the program that draws the house consists of commands to set the values of the **CurrentX** and **CurrentY** before each **Print** command. The center point of the roof could be printed using

```
CurrentX = 21 * font_width
CurrentY = 5 * font_height
Print "*"
```

The next line of the roof could be printed using:

```
CurrentX = 19 * font_width
Print "* *"
```

There should be three spaces between the pair of "*"s.



3 Input Values and Output Displays

BASIC programs usually consist of more than just commands to display output. Programs use commands that cause the user to be asked for some information that the program needs for its execution. Programs also do arithmetic with values supplied by the user as well as with values created by the program in the course of its execution. Still other problems require the program to ask questions about a current value in a storage location to determine what to do next. By the end of this chapter, you should understand how to write programs that use dialog boxes for input and display menus for output. Since the dialog boxes and display menus are the interface between the user and the computer, they should be designed to be effective and user-friendly. Using arithmetic operations and asking questions about values are dealt with in Chapters 4 and 5, respectively.

A typical application using user supplied information is a sales receipt. The idea is to display all the information and whatever information is derived from the input values so that the user understands. The example shown in Figure 3-1 gives order information for nine coffee mugs.

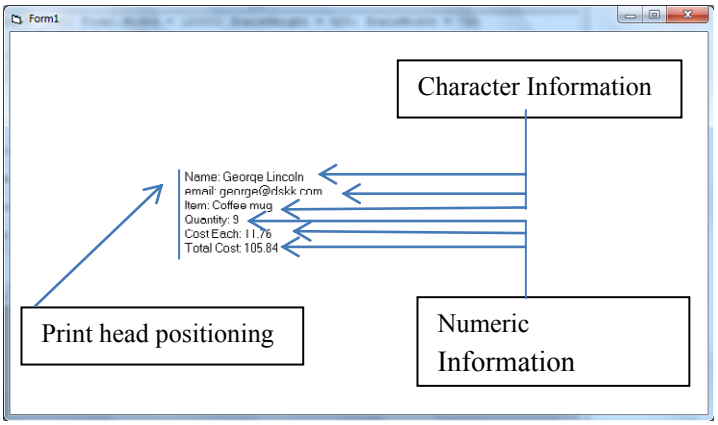


Figure 3-1: Sales Information

3.1 Data Type Values

A **literal** is a value that is either a number, such as 3; or a string of characters such as "the end." To distinguish data types we refer to a numeric literal as a numeric value. A string literal is a sequence of characters enclosed in double quotation marks. We also refer to a string literal as a **word** or **symbolic constant** or **string constant**.

3.2 Variables

A **variable** is a user supplied name for a location in computer memory. A variable's location in memory can contain a **literal** such as a number or a string of characters. BASIC takes care of determining what memory location will be assigned to a variable. BASIC also manages the operations of putting a value into a storage location and accessing a value from a storage location. BASIC will use the value in a storage location whenever the variable name that refers to that location occurs in the program.

Programs in this book will deal with two types of data or literals: numeric and string. **Numeric data** is simply a whole number or a number that includes a decimal point. **String data** will be any sequence of symbols. A variable can represent a storage location for either type of data and BASIC keeps track of the type of data that a variable represents. You should not use the same variable to store numeric data at one point in a program and string data at another point. We call a variable that represents a string value simply a **string variable**. A variable that represents a numeric value is simply called a **numeric variable**. A numeric or string expression is simply a combination of literals, variables, or operations for a given type of data. Operations for data types will be dealt with in later chapters.

BASIC requires variable names be formed using the rules that are shown in the SYNTAX box.

SYNTAX

Variable or Variable Name

A **variable name** can be up to 255 characters long. A **variable name** must begin with a letter of the alphabet and consist of only letters, digits, and underscores. After the first letter, the rest of the variable name can be any combination of letters, digits and underscore characters. BASIC **keywords** cannot be used as variables. Variables are not enclosed in quotation marks but represent an address of a storage locations in memory. Variable names are often referred to as a **variable**.

Keywords like **Int**, **Count**, and **Name** often seem like natural names for variables. If you use a keyword other than as BASIC intends, you will be given an error message about the line of code containing the keyword. You may look at the line of code and wonder what is wrong! (If you type a variable name using lowercase letters and BASIC automatically capitalizes the first letter, this is a strong sign that the name is a keyword. For this reason, it is suggested that you begin variable names with lowercase letters.) Notice that neither a hyphen nor a period may be used in forming a variable name.

Variable names should describe the values they represent. The following are examples of valid variable names that indicate the values they represent:

payment
name1
total_Sales

It is clear that variable names like *X* and *Y* do not convey as much information about what the variables represent as do *payment* and *total_Sales*. (You can play games with names and have the name imply something that is not true as using tax to represent a person's surname. In the long run you will be better off using names that infer the value the variable represents.)

On Your Own

1. Explain why the following variable names are invalid:

185CSCI	*Print*
WVBU90.5	*#one*
zip code	*Zip-code*

   Write a corrected version of each of these names.

3.3 Assignment Statements

An **assignment statement** in a programming language is an instruction that assigns a value to a variable, i.e., tells BASIC to place a value in the storage location associated with the variable. The **semantics** of a stqatement is the result of executing the statement-what are we intending to have happen when this statement is executed. The semantics of an assignment statement is that a value is placed in a storage location and can be used by referencing the name of the storage location, i.e. the variable name. The assignment statement consists of a variable name, an equal sign and then an expression or a literal. The variable name represents the storage location that will hold the value of the expression or literal that occurs on the right hand side of the equal sign. For instance, the following is an example of an assignment statement involving a numeric value:

```
payment = 20
```

In this assignment statement, *payment* is the variable name while 20 is the value assigned to the storage location that *payment* is assigned by BASIC.

Variables as well as literals can appear on the right side of an assignment statement:

```
grade1 = 100
grade2 = grade1
```

After the execution of these two statements, both *grade1* and *grade2* will have 100 stored in the memory locations they are assigned. Variables on the right side of an assignment statement should have previously been assigned a value so that the assignment of a value to a storage location makes sense. The semantics of the second assignment statement is to take the value in the storage location *grade1* and put a copy of this value in the storage location represented by *grade2*. The contents of *grade1* is not changed. The value assigned to a numeric variable may also be a number calculated by a mathematical operation. We discuss mathematical operations in the next chapter.

In an assignment statement involving a string variable, a character string that is being assigned to a string variable must be surrounded by double quotes to distinguish it from a variable name. For example, the following lines are assignments to string variables:

```
nam = "Mary Kay"
friend = nam
```

After the execution of these two statements both *nam* and *friend* will have the string "Mary Kay" stored in the memory locations they represent. We first put Mary Kay in the storage location *nam* (we identify the name of the storage location with the storage location). Notice that the string "Mary Kay" has eight characters. The space between *y* and *K* is treated as a character just as a comma, a period, or any other character that the computer recognizes. We are tempted to use **Name** as the variable name but if we do we get an error because **Name** is a keyword.

SYNTAX

Assignment

variable = literal or variable

The left side of an equal sign always has the name of exactly one storage location. The right hand side generates a value to put in that storage location.

In the next chapter we will explain how more complicated expressions can also be part of an assignment statement.

On Your Own

1. Explain why the following assignment statements are invalid. Write a corrected version of each of these statements.

```
town = "Lewisburg"
num , total = 12
x, y = 3
x = y = 3
```

3.4 Dialog Boxes

A **dialog box** is a window that appears on the screen to ask the user for a response and then is removed after the user responds. Dialog boxes allow users to input data to be processed by the program. In most cases, the computer will "ask" a user for certain data as the program is executing and will wait for the user to respond before continuing the execution of the program.

In BASIC, the input procedure for data uses the built-in-function **InputBox().** The **InputBox()** function uses a short message called a **prompt** to indicate to the user the type of data to provide. The **InputBox()** function brings a value into the program that must be stored in some variable. Consequently, we assign the result of this built-in-function to a variable name just as we saw how to assign any other value to a variable.

As an example, when the statement

surName = **InputBox**("Enter your last name.")

is executed, a dialog box with the prompt at the top left is displayed on the output form. The dialog box is shown in Figure 3-2.

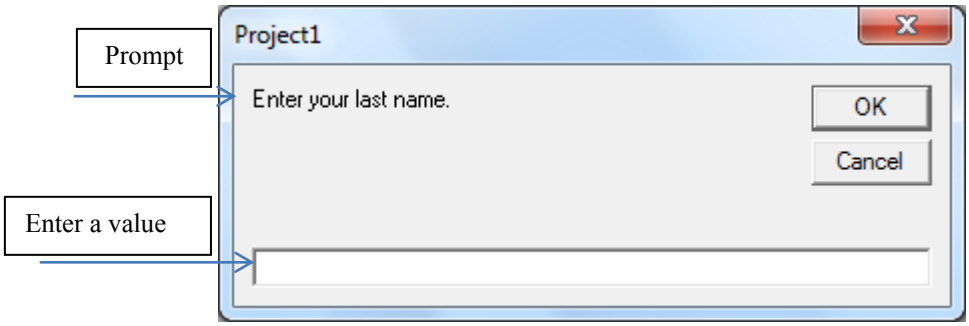


Figure 3-2: BASIC Dialog Box for InputBox

The prompt is shown in the top left corner of the dialog box and indicates to the user what kind of data to enter in the area pointed to at the bottom. The string entered by the user is assigned to the variable *surName*. A string variable can also be used as a prompt:

```
kindOfInfo = "What's the frequency of this event (Often/Rare)?"
frequency = InputBox(kindOfInfo)
```

When this code is executed, the message "What's the frequency of this event (Often/Rare)?" appears as the prompt message in a dialog box. The data entered by the user is assigned to the variable **frequency**. Notice how informative the prompt is in indicating the responses wanted. Without the prompt the user would not know what to enter.

The syntax of **InputBox()** is summarized here:

SYNTAX

InputBox()

variable = **InputBox**("prompt")

InputBox() accepts a string of characters entered from the keyboard and stores them in variable. The prompt string tells the user what kind of information is to be entered.

EXAMPLE: surName = InputBox("Enter a surname:")

In the example, "Enter your surname:" is used as a prompt to tell the user what kind of information is needed. The prompt is always enclosed in a pair of double quote marks.

On Your Own

1. Explain why the following statements are invalid. Write a corrected version of each of these statements.

   zip = **InputBox**
   s = **InputBox** (Enter your size.)

2. The following statement is syntactically correct, but what is its semantics: X = InputBox()?

3.5 Displaying Values

Variable values can be accessed and displayed by the **Print** command. The variable name of the storage location containing the information to be displayed follows the keyword **Print**. Do not put quotation marks around the variable name. (What will be printed if you do?) This usage of the **Print** command is shown in the following code:

```
rating = "Poor"
Print rating
```

When these instructions are executed, the word "Poor" is displayed on the screen.

A text message may also be displayed with values of variables. A semicolon separates the text message from the variable if the variable value is to be displayed adjacent to the text. If the variable value is to be placed in the next print zone, a comma is used. (If no punctuation is specified, BASIC automatically includes a semicolon.) The following code prints text and the value of a variable:

```
temp = 44.4
Print "It's"; temp; "degrees outside."
```

Run this program and determine what needs to be changed to make the output more readable.

When using a semicolon in a **Print** command, you may need to supply spaces as part of the string so the output reads as intended. For example, the two print statements:

```
Print 35; "degrees"
Print 35; "degrees"
```

have the following outputs:

```
35degrees
35 degrees
```

Example 3-1. Using variables and the **InputBox()** function, write a program that inputs a person's name, hometown, and three favorite flavors of spice drops. Display the data entered in the center of the output display as shown:

Name: Chandler Bing
Hometown: New York

Favorite flavor: Apple
1st runner up: Strawberry
2nd runner up: Lemon

SOLUTION:

```
Rem Display favorite spice drop flavors
Rem INPUT: Person's name, hometown, and three favorite spice drop flavors
Rem OUTPUT: Input data centered on the display

    Cls
    Rem Get input
        nam=InputBox("Name:")
        home=InputBox("Hometown:")
        flavor1=InputBox("Favorite space drop flavor #1:")
        flavor2=InputBox("Favorite spice drop flavor #2:")
        flavor3=InputBox("Favorite spice drop flavor #3:")
    Rem Display output
        CurrentY = 8*font_height : CurrentX = 30 * font_width
        Print "Name: ";nam
        CurrentX = 30 * font_width
        Print "Hometown: "; home
        Print
        CurrentX = 30 *font_width
        Print "Favorite flavor: ";flavor1
        CurrentX = 30 * font_width
        Print "1st Runner Up: "; flavor2
        CurrentX = 30 * font_width
        Print "2nd Runner Up: ";flavor3
```

COMMENTS: Notice the extra blank spaces included before the closing quote marks in the string constants in the last two **Print** commands. These spaces cause the names of the flavors to line up when they are printed. What would the output look like if the last three print statements had commas in the place of semicolons? Try it! Also observe that **CurrentY** is not set again after the first print statement. We do not need to make any special line settings because the output is to occur on consecutive lines and **CurrentY** is always set to a value that moves the print head to the left margin of the next line.

On Your Own

1. Write a program to input your name and address as an entry in an address book. Print the entry in the center of the display using the following format:

Lisa Simpson

742 Evergreen Terrace

Springfield, NT 49007

3.6 Numeric Values As Input

Data entered in a dialog box using the command **InputBox()** is of type string. BASIC treats the data as a sequence of characters. Typically programs also use numeric values to carry out calculations. Numeric values also need to be input when the program is executing. Values used in calculations must be numeric values, not strings. Data entered in a dialog box for the command **InputBox()** are brought into the computer as a string of characters regardless of whether the data is string data or numeric data. If the value is numeric and will be used in a computation BASIC needs to convert the string to a numeric representation. The **Val()** function as described in the SYNTAX box does just that.

SYNTAX

Val()

variable = Val(“numeric literal**”)**
where a numerical literal consists of digits and
possibly a decimal point and possibly a sign (+/-)

EXAMPLE: *Pi* = **Val(“3.14159”**)
The argument for the Val function is normally the result of the execution of the **InputBox**() function as:

There are cases when **Val**() is forgotten where BASIC does not make the string to numeric representation conversion and unexpected results may be produced.

On Your Own

1. For the program

   X = **Val**(**InputBox**(“enter salary”)
   Print “salary, X”

   What is printed?

2. For the program

   X = **Val**(**InputBox**(“salary”))
   Print Salary = , “X”

   What is printed?

3. What will be the output of the following program segment:

   X = **InputBox**(“Enter 1234”)
   Y= **InputBox**(“Enter 4567”)
   Print X + Y

3.7 Creating Output Menus

An **output menu** is a combination of labels and values positioned on the output form to make it easy for users to view and understand information. In Example 3-1, we saw a program that set the values of **CurrentX** and **CurrentY** and used these settings along with the **Print** command to create an **output menu**. The program in the solution of Example 3-1 is a good example of a program that creates user-friendly output using an output menu.

Example 3-2. Write a program that inputs earnings information for a surfing instructor and produces a menu that displays this data as well as the total earnings.

OUTPUT:

Name: Phoebe Buffay

Number of sessions given: 8
Payment per session: 30
Total tips: 40.25

Total earnings: $280.25

SOLUTION:

```
Rem Generate an earnings report
Rem INPUT: Name of employee and earnings data:
REM            no. of customers and amount of tips
Rem OUTPUT: The input information and earnings data
      Cls
Rem Get input-name, no. of customers, and total of tips
      nam = InputBox("Name:")
      num = Val(InputBox("Number of sessions given:"))
      fee = Val(InputBox("Payment per session:"))
      tips = Val(InputBox("Total tips:"))
Rem Enter total earnings
      total = Val(InputBox("Enter total of all earnings"))
Rem Display output
      CurrentY = 9 * font_height
      CurrentX = 25 * font_width
      Print "Name: "; nam
      CurrentY = 11 * font_height
      CurrentX = 25 * font_width
      Print "Number of sessions given:  "; num
      CurrentX = 25 * font_width
      Print "Payment per session:        "; fee
      CurrentX = 25 * font_width
      Print "Total tips:                 "; tips
      CurrentY = 15 * font_height
      CurrentX = 25 * font_width
      Print "Total earnings:             $"; total
```

COMMENTS: The prompts used with the **InputBox()** commands are clear in describing what values are to be entered. The prompt is designed as an argument for the **InputBox()** command so the user enters the appropriate data. If a user enters an incorrect type of data, the program will use the incorrect data as if it were correct! Similarly, note that the output labels clearly identify the values printed so that the results of the program are easily understood. We will learn how to output a value automatically in a monetary format when we discuss the **Format()** option in the next chapter.

The **Val()** function is applied to the result of the **InputBox()** function only when the program expects the user to enter a numeric value.

Note that **CurrentX** is changed more often than **CurrentY**. After each **Print** command in this program, the **CurrentX** and **CurrentY** values are automatically set to position the print head at the beginning of the next line. When we want the next output to appear in this next line, **CurrentY** is already correctly set and does not need to be changed. In these cases, only the value of **CurrentX** needs to be changed to place the output correctly at a position away from the left edge of the output form.

3.8 Putting It All Together

1. Write a program that accepts as input a dog's name, breed, and favorite toy. Display this data in the center of the screen.

2. Write a program that has a name and the age of the person as input. Display this information in the middle of lines 18 and 19. The name should appear in line 18 and the age in line 19.
3. Write a program that has a name and an address (street, city, state, zip) of two different people as input. Display the information as two columns of output with the columns' left margins being locations 6 and 48.
4. Write a program that has a student's name and social security number as input. Print the name starting at column 18 of line 9 and the social security number in the middle of the next line.
5. Input the name of a course and the average grade for each of the three hourly exams. Display the information in the middle of the output form with the course name on line 10. The grades should be labeled: Average Exam I:, Average Exam II, and Average Exam III. The grade labels should start in column 30 of lines 12, 13, and 14.
6. For input enter a student's name and the four courses the student is enrolled in. Display the name followed by a blank line followed by each course on a different line. Center all the output in the center of the output screen.
7. Enter the name of two teams competing against each other and the final score for each team. Display the information as shown in the middle of the output form.

Team1	vs	Team2
xxx		xxx

8. Write a program that accepts as input the name of a major league baseball team and its current won-loss record. Display the information in the center of the output form. Take 32-19 as the won-loss record.
9. Input the name of a city and the high and low temperatures for one day (enter as name of month and day of the month). Title each piece of information and display on different lines starting with line 10. All titles should start in column 25

4 Numeric Calculations

Programming involves more than creating effective user interfaces. The programmer must also know how to process input data (such as number of hours worked and pay rate) to generate results (such as payroll information). This chapter takes another step in giving you the tools needed to solve problems that involve computations. Computational problems typically involve a combination of numerical operations and the use of predefined mathematical functions.

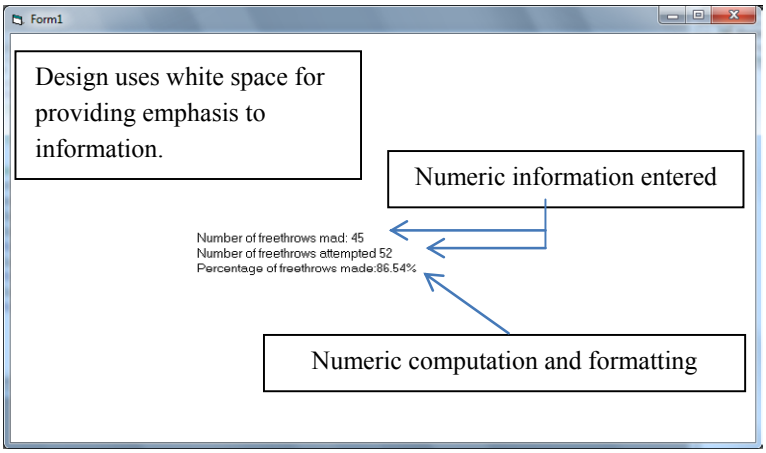


Figure 4-1: Sports Statistic Computation

4.1 Operations, Functions and Expressions

BASIC programs often include mathematical operations performed on values input as well as on other values generated by the program but not directly provided by the user. In BASIC, numerical operations may be performed on numbers and the values stored in locations representing numeric values. BASIC numerical operators that act on one or two values are listed in Table 4-1. A command that combines explicit values, constants, variables, operators, and functions is called an **expression.** A simple assignment statement is an expression but in general, expressions will involve more than a single term.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
-	Number negation
^	Exponentiation

Table 4-1: BASIC Numerical Operators

4.2 Operation Hierarchy

To evaluate an expression in a program containing only arithmetic operations, there are two issues to be understood. The first is to know which values will be used in the computation. The second is to know the order in which the operations will be performed. In an expression involving only operations and constant values, the operations are performed on the values in the expression. If an expression includes a variable, the computer must access the current value of the variable and use that value in the computation in place of the variable name.

The question of how the computer will execute a series of arithmetic operations used in an expression is a bit more complex. If you are faced with a computation without parentheses such as

$$-A * B + C / D \wedge E - F$$

there are several possible ways to evaluate the expression. Proceeding from left to right and doing the next operation is one possibility. Such an approach does not fit the rules we are familiar with for evaluating expressions. The key idea is to recognize that some operations are more important than others and we must do the more important operations first. The order of importance or priority for arithmetic operators is shown in Table 4-2.

1st priority	^	Exponentiation
2nd priority	-	Number negation
3rd priority	* /	Multiplication and division
4th priority	+ -	Addition and subtraction

Table 4-2: Priority of Arithmetic Operations

The operation of changing the sign of a value, called **negation**, is the only operation to act on a single value. The negation operator is different than subtraction which operates on two values. Typically, negation occurs on the first value in an expression or before a parenthesis indicating the result of an expression in parentheses is to have its sign changed. We do not order operations such as * and / between themselves.

When we are evaluating operations at the same priority level, we evaluate any operation at that level as it occurs moving from left to right. The rule the computer follows for executing expressions will give the result you expect. The evaluation process starts by scanning the expression from left to right doing all the operations involving the highest priority operators that occur in the order they occur. Then the values calculated are substituted into the expression forming a simpler expression only involving operations at a lower priority. Now, the computer again proceeds from left to right scanning this intermediate expression and evaluates operations at the highest priority remaining in the order in which they occur. The computer continues to scan the resulting simpler expressions in this way until a single value results.

For example, to evaluate the expression 3 * 4 + 5 ^ 2 / 5 – 8, the steps needed are shown in Table 4-3.

3 * 4 + 5 ^ 2 / 5 – 8	Initial expression
3 * 4 + 25 / 5 – 8	After exponentiation
12 + 5 – 8	After multiplication and division
17 – 8	After the addition
9	Final result after the subtraction

Table 4-3: Example Arithmetic Expressions

The asterisk is used to signify multiplication in BASIC. Just entering

12 num

to multiply the value of *num* by 12 will give an error. The correct expression is

12 * num or num * 12

When variable names are involved in assignment statements involving arithmetic expressions, the expression looks like normal algebra but has a different meaning in many cases. The assignment statements shown in Table 4-4 are valid.

A	x = 0
B	x = x + 1
C	y = x * 2
D	total = x + y

Table 4-4: Assignment Statements

Statement **B** needs a bit of an explanation. The variable *x* occurs on both sides of the equal sign. It looks as if the statement means 0 = 1 which is obviously not true. BASIC takes some liberties with our normal expectations. Here the equal sign does not mean equality. The language designers had only the keyboard symbols to use, and they knew that they would need a symbol to represent *"assign the value of an expression to a variable*." What they did was use the equal sign to mean different things in different contexts. This is called **symbol overloading**.The equal sign in **B** means: evaluate the expression on the right side of the equal sign and store the result in the location of the variable whose name occurs on the left side of the equal sign. When evaluating an expression, BASIC first sets aside the storage location on the left side of the equal sign before evaluating the expression on the right side of the equal sign. Consequently, the expression on the right hand side is evaluated independently of any knowledge about where the result will be stored. Thus, in expressions like **B**, the same variable can occur on both sides of the equal sign without causing any difficulty provided we understand the semantics of the statement. The meaning in **B** is to add 1 to the current value of variable *x* and store the new value at *x*'s location. Another meaning of the equal sign is discussed in Chapter 5. Notice also that before and after **B** is executed, the storage location *x* contains different values! A storage location's value is always the last value stored at that location with no memory of any value that was previously in that storage location.

4.3 Subexpressions

In BASIC, operations within a numeric expression are performed following the order shown in Table 4-2. As in mathematical expressions, parentheses may be used to specify a part of an expression that should be evaluated as an independent expression. See Table 4-5 for examples of arithmetic expressions that demonstrate the evaluation order of operators with or without parentheses.

Expression	After Operation 1	After Operation 2	After Operation 3	After Operation 4	Final Result
3 * 2 + 5	6+5	11			11
3 + 2 * 5	3 + 10	13			13
(3 + 2) * 5	5 * 5	25			25
3 * 2 - 6 / 2	6 - 6/2	6 - 3	3		3
-2 * (3 - 1) ^ 2	-2 * 2^2	-2 * 4	-8		-8
((5 + 4) - 7) + 4 / 2	(9 - 7) + 4/2	2 + 4/2	2 + 2	4	4

Table 4-5: Example Arithmetic Expression Evaluations

The rule for evaluating arithmetic expressions involving subexpressions in parentheses is first to evaluate any subexpression within a pair of parentheses. The computation always proceeds from the innermost and left-most pair of parentheses and works out until only a single value remains to represent the complete expression that is contained in parentheses. In general, an expression may or may not contain one or more subexpressions. Table 4-6 indicates how to proceed in evaluating an expression with parentheses. We see how this rule is carried out in the expression:

Original expression:	(3 + (2 - 4)) * (12 / 6) ^ 3
Evaluate (2 - 4)	(3 - 2) * (12 / 6) ^ 3
Evaluate (3 - 2)	1 * (12 / 6) ^ 3
Evaluate (12 / 6)	1 * 2 ^ 3
Evaluate 2 ^ 3	1* 8
Evaluate 1*8	8

Table 4-6: Evaluating Arithmetic Expression with Parentheses

4.4 Built In Functions

Functions that involve nontrivial computations are often part of a programming language that can be viewed as "black boxes." The programmer supplies the values needed by a function and BASIC returns the required value using built-in code for the computation. Some of the useful functions in BASIC are shown here:

SYNTAX

Built In Functions

BASIC FUNCTION	NAME
Abs()	Absolute value
Exp()	Exponential function
Log()	Natural logarithmic function
Sqr()	Square root

On Your Own

1. Evaluate the following expressions.
   (i) 2 + 8 - 6 * 3
   (ii) (3 * 4) ^ 2 - (-5) + 9
   (iii) 4 ^ 3 * 3 ^ 2 * 8 / 4
   (iv) 10 / 5 * 8 - 9
   (v) 5 * 3 * (8 - 3) / 4

2. Explain why the following statements are invalid and provide corrections for each.
   x, y = 3 + z
   x + y = z

3. List the order the operations are performed for each expression:
   a) 3 - 2 / 4 * 6 ^ (2 - 3) + 5
   b) (x1 * x2) / x3 ^ x4 - x5 + x6 ^ x7
   c) x1 / (x2 + x3) * x4 / x5 ^ (x6 - x7)

A numerical expression may also contain any previously defined numeric variables. The following is a valid assignment statement that contains a numeric expression.

overtime = (hours – 40) * 1.5 * wage

This statement computes an overtime payment based on the number of hours worked over 40. The computer uses the hourly wage rate of one and one-half the hourly wage rate for each of the first forty hours for the overtime wage rate. The result of the computation is assigned to *overtime* (puts the result in the storage location assigned to the variable *overtime*).

Example 4-1. Write a program that inputs a long jumper's name and country along with three distances jumped measured in meters. Using the three input distances, calculate the average distance jumped. Use an output menu to display the input data and the calculated average.

OUTPUT:

Name: Samantha Jones
Country: USA

Jump 1 distance: 6.7
Jump 2 distance: 6.82
Jump 3 distance: 7

Average distance: 6.84

SOLUTION:

	Rem Displays a long jumper's distances for three Rem jumps and the average of the distances Rem INPUT: Personal info and three distances Rem OUTPUT: Input values and average distance Cls Rem Get input nam = InputBox("Enter name:") country = InputBox("Enter country:") jump1 = Val(InputBox("Enter first distance:")) jump2 = Val(InputBox("Enter second distance:")) jump3 = Val(InputBox("Enter third distance:"))
A	Rem Calculate average ave = (jump1 + jump2 + jump3) / 3 Rem Display output Rem Output personal info CurrentY = 8 * font_height CurrentX = 30 * font_width Print "Name: "; nam CurrentX = 30 * font_width 'line is 9 Print "Country: "; country Rem Output three jumps CurrentY = 11 * font_height CurrentX = 30 * font_width Print "Jump 1 distance:"; jump1 'line is 11 CurrentX = 30 * font_width Print "Jump 2 distance:"; jump2 'line is 12 CurrentX = 30 * font_width Print "Jump 3 distance:"; jump3 'line is 13 Rem Output average CurrentY = 15 * font_height CurrentX = 30 * font_width
B	Print "Average distance:"; ave

COMMENTS: In the solution program, line **A** is an assignment statement that contains a numeric expression. The statement computes the average of the three given jumps. Notice that the program includes **B** to display the value of the variable *ave* after it has been calculated.

On Your Own

1. Write a program that asks the user to enter a number of yards and a number of feet. The program then calculates the number of inches that is equivalent to the given distance and outputs that number.
2. Write a program that asks the user to enter the number of quarters, dimes, nickels, and pennies that the user has. The program then calculates the total value of the coins as a number of pennies and outputs that amount.
3. Write a program that converts miles per hour into kilometers per hour. A kilometer is .62 of a mile.
4. Write a program that converts kilometers per hour into miles per hour. A kilometer is .62 of a mile.
5. Write a program that converts hours, minutes, and seconds into seconds.
6. Write a program that converts years, months, and days into days. Assume 30 days in every month.

4.5 Concatenation

Concatenation is a BASIC operation performed on two strings. Concatenation is the operation of appending one string to the other. The ampersand (&) is the concatenation operator. For example, the following lines of code concatenate strings:

```
s1 = "gotta"
s2 = "getaway"
s3 = s1 & s2
Print s3

OUTPUT:
gottagetaway
```

Before the **Print** command, *s3* has the value "gottagetaway" without a space. To insert a space between the two words, you could include a space character as follows:

```
s3 = s1 & " " & s2
```

Concatenation can also be done with numeric values. To do this, BASIC first converts the numeric value to a string and then performs the concatenation. For example:

```
price = 0.55
s = "One roll costs $"
t = s & price
Print t

OUTPUT:
One roll costs $0.55
```

In this example *price* has a numeric value. When *price* is concatenated with *s*, BASIC converts the value 0.55 to the string "0.55" and uses the string in the concatenation.("0.55" consists of four symbols: 0, ., 5, and 5.)

SYNTAX

Concatenation

Any combination of two literals or variables X1 and X2 joined by an ampersand (&) generates a string with the character representation of X1 followed by the character representation of X2.

EXAMPLE: "abc" & " def" = "abcdef"

4.6 Formatting Output

One of the problems programmers face when designing effective output schemes for a program involves displaying computed results in an appropriate format. The code

```
x = 1 / 11
Print x
```

generates the output 9.090909E-02. This form is called **exponential notation**. The E-02 says to multiply the number 9.090909 by 10^{-2} to get the normally expected value for this number. Computers usually display numbers in exponential notation rather than having leading zero digits following the decimal point. The notation is often referred to as a **normalized** representation of the value.

Although the exponential notation is well defined, the visual effect of such a number format is very poor. BASIC provides the **Format()** function to control the format of displayed numbers and strings of symbols. The idea is to prescribe the recipe for displaying digits or symbols of a value. We will only discuss how **Format()** is used with numeric values. The value being printed may have quite a different representation in memory. The **Format()** command does not change the memory representation of a value but just describes how many of the digits or symbols in an internal representation will be displayed. The example 1/11 will be more meaningful printed as 0.09 than as 9.090909E-02. When using the **Format()** command, print positions on a line that will contain numeric values are described using zeros. For example, if the value should be displayed with one digit to the left of the decimal point and two decimal digits to the right of the decimal point, the format pattern would be represented by "0.00." The code that causes the formatting is supplied by the built-in-function **Format**(). The layout instructions are written inside double quotes.

The formal syntax for this function involves two arguments, a value and a recipe. The SYNTAX box for **Format()** gives the details.

SYNTAX

Format()

Format(variable or literal, "recipe")
EXAMPLE: x = 1/11

Print **Format**(x, "0.00")
RESULT: 0.09

The **Format()** function simply produces a string following the description given as the second argument of the function. The **Format()** command can appear at any location in a list of values to be printed or on the right side of an assignment statement. The code

```
mpg = 200 / 7
s = Format(mpg, "0.0")
Print "Miles per gallon = "; s
```

produces this output:

```
Miles per gallon = 28.6
```

Notice that even though the format pattern only specified one digit to the left of the decimal point that two digits are printed. The **Format()** function retains all of the digits to the left of the decimal point. Digits to the right of the decimal point are rounded to match the specified precision. A dollar sign ($) and a comma (,) can be included in the format pattern. The following code

```
total = 0.55 * 3030
s = Format(total, "$0,000.00")
Print "Total cost is "; s
```

produces this output:

```
Total cost is $1,666.50
```

Notice that the trailing zero is printed. The zeros in the format pattern that follow the decimal point specify the number of digits that will be printed.

BASIC has several useful built-in formats. Instead of specifying a pattern, you can use the string "**currency**" as a format pattern. The following code

```
total = 0.55 * 3030
s = Format(total, "currency")
Print "Total cost is "; s
```

produces the same output as the previous example. If the value has fewer digits to the left of the decimal point than are specified by zeros in the format pattern then zeros will be produced. The following code:

```
total = 902.1
s = Format(total, "$0,000.00")
Print "Total cost is "; s
```

produces this output:

```
Total cost is $0,902.10
```

BASIC has a different method of making columns of values while suppressing leading zeros. Use the @ sign instead of 0's in the display description. If the number is not large enough to take up all the print positions, leading blanks are printed. The following code shows how two numbers can be printed in six print positions each, regardless of the size of the numbers.

Print Format(123,"@@@@@@");**Format**(23456,"@@@@@@")
OUTPUT: ___123_23456

In addition BASIC also has a built in recipe to display percentages. For example,

Print Format(.091, "percent")

and the value will be displayed as 9.1%. To display a number with just two decimal digits use the word **standard** in double quotes just as **percent** was used above.

Example 4-2. Write a program that produces a table showing the sale prices of shovels ($3.78), axes ($4.98), and hammers ($5.65). The program should calculate and display the price of each item and the price of each item after a 6% sales tax has been added.

OUTPUT:

	Shovel	Axe	Hammer
Original	$3.78	$4.98	$5.65
Taxed	$4.01	$5.28	$5.99

SOLUTION:

```
Rem Produces a table that shows the original
Rem and taxed prices of three items.


Rem OUTPUT: Table of prices
   Cls
   shovel = 3.78: axe = 4.98 : hammer = 5.65
Rem Display the table header
   Print ,
   Print, "Shovel", " Axe", " Hammer"
   Print "Original",
   Print , Format(shovel, "currency"),Format( axe, "currency"),Format( hammer, "currency")
Rem Calculate the taxed prices
   priceShovel = shovel * 1.06
   priceAxe = axe * 1.06
   priceHammer = hammer * 1.06
Rem Format the output
   fPriceShovel = Format(priceShovel, "currency")
   fPriceAxe = Format(priceAxe, "currency")
   fPriceHammer = Format(priceHammer, "currency")
Rem Display the taxed prices
   Print "Taxed",
   Print ,fPriceShovel,fPriceAxe, fPriceHammer
```

COMMENTS: In the solution program, the header of the table is first displayed using the **Print** command. The taxed prices are calculated based on the given original prices. Note the use of commas in the last **Print** command to display the taxed prices below the original prices.

On Your Own

1. Form the following strings using concatenation where the numerical value is stored in the variable nonString.
   a) There are 18 students in the class.
   b) WalMart operates 3165 stores.
   c) The rugby team won by 35 points.
2. Write a program that asks the user for a bet on a horse in a horse race. For the prompt concatenate a question with a numerical value of how much the user has. Store the concatenation result in a variable and then use it in place of the prompt in **InputBox**().

4.7 Putting It All Together

Use **CurrentX**, **CurrentY**, and **Format()** statements to enhance the output for each of these programs.

1. Write a program that will prompt the user for five numbers and then display the sum and average of those five numbers. Include appropriate commenting. Assume the numbers will have between one and three digits after the decimal point.
2. Write a program to prompt the user for a temperature in Fahrenheit (for example, 30 degrees Fahrenheit). Convert that temperature to Celsius and display both values. Clearly label the output and use comments to make your program readable. Use the equation

   celsius = (fahrenheit – 32) * 5 / 9

   to convert a Fahrenheit degree into an equivalent Celsius degree.
3. Write a program to prompt the user for a temperature in Celsius (for example, 30 degrees Celsius). Convert that temperature to Fahrenheit and display both values. Clearly label the output and use comments to make your program readable. Use the equation:

   fahrenheit = 9/5 celsius + 3

   to convert a Celsius temperature into its Fahrenheit equivalent.
4. Suppose the current currency conversion factor for converting US dollars into British pounds is

   1 Us = 0.6155 GBP.

   For any amount of US dollars entered by the user, output the value of these dollars in terms of the British pound.
5. The ABC Bakery charges $3.85 for a loaf of white bread, $4.65 for a loaf of whole wheat bread, and $6.58 for a dozen dinner rolls. Write a program that inputs the number of loaves of white bread (M), the number of loaves of whole wheat bread (N), and the number of dozens of dinner rolls (Q) ordered. Compute the total cost of the order. Display the charge for each item and for the cost of the total order. Use **Format()** for displaying the currency values.
6. Write a program that accepts as input the number of miles traveled and the number of gallons of gas purchased. Output this data along with the number of miles per gallon for this trip. Miles per gallon is the quotient of miles and gallons. Use an output menu for the output.

7. Write a program that inputs two numbers and acts as a calculator. The program should create an output menu that displays the sum, difference, product and quotient of the two numbers. The output displayed should be similar to the following if 10 and 2 were the two input numbers:

10 plus 2 is 12
10 minus 2 is 8
10 times 2 is 20
10 divided by 2 is 5

8. Write a program that prompts the user for the current amount due for a credit card. The program then calculates the finance charge and determines the new balance. The finance charge each month is 1.5% of the past due amount. The new balance is the sum of the past due amount and the finance charge. For an input amount of 255.50, the output should be similar to this:

Past Due	Finance	New
Amount	Charge	Balance
$0,255.50	$003.83	$0,259.23

9. Input the total number of miles a fleet of trucks drive in a week. If a truck get 7.86 mpg, compute the number of gallons of gas the fleet uses in a week.
10. Write a program to compute the monthly payments for a loan. Run the program for the three case given. The formula for computing the monthly payment for a loan of *A* dollars at *annualInt* rate of interest for a length of *months* is:

Payment = A * (annualInt/12)/(1 + (annualInt/12))^(-months)

a) Joe Brown took a loan of $20,000 to buy a new car. If the payback period is 60 months and the interest rate is 1.9%, how much are the monthly payments?
b) The parents of Joe Smith took out a $35,000 loan to pay his tuition. If the interest rate is 6.85% and the payback period is 120 months, how much will the monthly payments be?
c) Jil Meyer graduated from college with $25,600 in student loan debt. If the annual interest rate is 3.4% and the payback period is 10 years, how much are the monthly payments?

11. Find the standard deviation of the following set of numbers: 8,12,9,11,7,5,4,3,13,11,9,18,16,1
The formula is:

var = 1/n*(x1^2 + … + xn^2) – ((x1 + … + xn)/n)^2
stDev=sqr(var)

where n is the number of numbers and x1, x2, … , xn are the actual numbers.

5 Decision Making

In programming languages, decision making is a fundamental feature that involves comparisons between two values and/or questions about the current state of a computation. As a result of asking questions about values in a program, the program can either alter the flow of control or choose a proper computation option for the value being processed. In this chapter, we introduce decision making in BASIC.

In a program to determine a letter grade for a set of scores the average of the grades is first computed. To determine the letter grade the average is then compared with various ranges of numbers that correspond to values representing the same letter grade. The results of the grade program are shown in Figure 5-1.

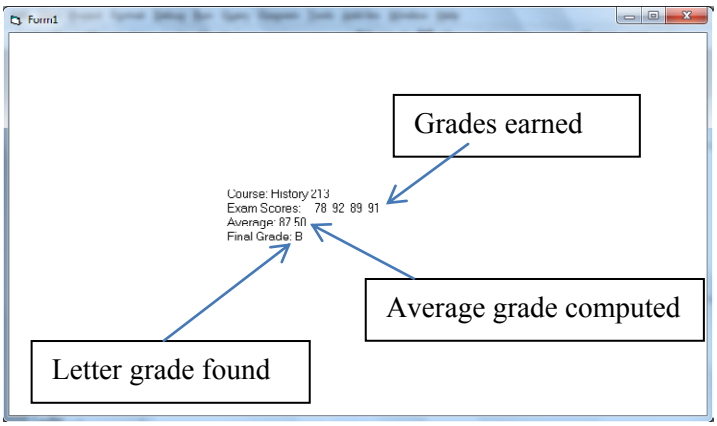


Figure 5-1: Computing a Letter Grade

5.1 Simple Comparisons

In programming languages, decisions are made based on **comparisons**. A comparison is evaluated as either true or false. True and false are called **logical** or **boolean values**. Knowing the result of the evaluation, a program can choose appropriate actions. In BASIC, comparisons can be made between pairs of numeric values (numbers, numeric variables, and values of numeric expressions) or between pairs of string values (character strings, string variables, and values of string expressions). Based on these comparisons, the program chooses the code to be executed next.

5.2 Numeric Comparisons

Numeric comparisons are those which involve the comparison of two numeric values. The relational operators used in a numeric comparison are the same as those used in mathematics. In BASIC, numeric comparisons are formed in a manner similar to comparisons in mathematics. We can compare pairs of numeric values, variables or expressions. For instance, the following are valid numeric comparisons:

```
3 > 1
10 = 10
4 + 9 <> 11
2 <= 6 * 5
```

The BASIC relational operators and their numeric meanings are given in Table 5-1.

Operator	Meaning
=	equal to
<>	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Table 5-1: BASIC Relational Operators

You may already be familiar with the above relations when evaluating numeric comparisons involving numeric values. However, you may be unfamiliar with evaluating comparisons that contain variable names. The following are comparisons of numeric values represented by variables:

```
x = 10
y + 9 <> 11
w <= z * 5
```

w, x, y and *z* are variables with numeric values. In these comparisons, the computer first accesses the value in the variable's storage location. The accessed value is then used in the comparison. If a computation involves an expression, the computation is carried out before the comparison is made. Comparisons are always carried out with two values. A variable involved in a comparison must already have an assigned value whether it is involved in a computation or not.

On Your Own

1. Explain why the following numerical comparisons are incorrect. Write correct versions of these comparisons.

   < 98

   "dog" < 47

2. Evaluate the following numeric comparisons as TRUE or FALSE. When evaluating variables in these comparisons, assume the following assignments: A = 4, B = 9, C = 0.7, D = 22.

   99 > 56

   B <> A + 5

   C >= 0.07

   1.002 <= 1.02

   A * B – C / D = 32

   C * D < C * A

5.3 Strings

In BASIC, we can apply the relational operators to character strings as well as to numeric values. Comparisons involving character strings are called **string comparisons**. However, before we can compare strings, we must be familiar with the representation of character strings. With string comparisons it may seem odd to say one string is less than another. When you understand that "smaller" means occurring before in a dictionary of all possible strings, the idea is easier to grasp. A clear difference in BASIC is that we do not have only 26 letters, but upper and lower case letters, numerals, and special symbols like ; that are all lined up in order just as we think of the English alphabet ordered from A to Z with A as the first or smallest and Z as the last or largest.

5.4 Character Representation

A set of symbols is called an **alphabet**. A sequence of symbols from an alphabet written one after another is called a **word** or a **string**. For now, let us use the capital letters of the English language (A, B, C, …, Y, Z) as our alphabet. In general, we require an alphabet to have all the symbols ordered relative to each other so that it is clear which symbol in the order occurs first when comparing two symbols. In the English alphabet, we naturally order the letters from A to Z with upper and lower case letters equivalent to each other. The computer represents each letter of this alphabet by a code value consisting of a sequence of 0's and 1's that make up what is called a **binary code**. One code created to represent symbols on a computer is called the ASCII-8 code. The ASCII-8 code for the capital letters of the English language is given in Table 5-2.

Letter	Binary Form	Letter	Binary Form
A	10100001	N	10101110
B	10100010	O	10101111
C	10100011	P	10110000
D	10100100	Q	10110001
E	10100101	R	10110010
F	10100110	S	10110011
G	10100111	T	10110100
H	10101000	U	10110101
I	10101001	V	10110110
J	10101010	W	10110111
K	10101011	X	10111000
L	10101100	Y	10111001
M	10101101	Z	10111010

Table 5-2: ASCII-8 Code for A–Z

When string variables or literals are used in a comparison, it is really the codes for the individual symbols that are being compared.

5.5 Dictionary Ordering

Evaluating string comparisons is similar to looking up words in a dictionary. (The alphabet may be quite different from the normal English alphabet.) The computer looks at the first letter in each of the words and if they are different, evaluates the words according to which of these two letters occurs first in the ordering of the alphabet (whatever set of characters is used). For instance, the following comparison is true because "B" precedes "J" in the alphabet:

"BROWN" < "JONSON"

It is not always possible to evaluate a condition by only looking at the first letter of a word. If two words that begin the same but end differently are being compared, the computer performs the comparison at the first letter position in the words at which different letters occur. This is just as in looking up words in a dictionary. For example, in comparing the words "ANTS" and "ANTHRACITE" the first three letters of both words are identical. If we were placing these words in alphabetical order (in a dictionary), we would compare the "S" in "ANTS" to the "H" in "ANTHRACITE" to determine which word should occur first in the dictionary. Since "S" occurs after "H" in the computer ordering of the upper case letters, we conclude that the comparison

"ANTS" > "ANTHRACITE"

is true. The computer begins at the leftmost character of each string and compares each corresponding letter of each word until it finds a letter position in which the two letters of the two words are different. In this case, the computer must compare the fourth letter of each of the words to conclude that this comparison is true. Notice that the length of the words compared did not determine the order. Another way to evaluate a string comparison is to first translate the symbols to their ASCII-8 codes and then compare the numbers these codes represent. For example, let us examine the comparison from above:

"BROWN" < "JONSON"

Since the first letters of each of these words are different, we need only to compare the first letters. If we look up the ASCII-8 code for the first letter in each string, we find the binary number 10100010 represents "B" and similarly the binary number 10101010 represents "J." We can look at the ASCII-8 code for each letter as a string from the alphabet {0, 1} and look for the first position in which these words have a different letter of this alphabet. For just the codes for B and J we see

$$B = 1\ 0\ 1\ 0\ \boxed{0}\ 0\ 1\ 0$$
$$J = 1\ 0\ 1\ 0\ \boxed{1}\ 0\ 1\ 0$$

where the fifth position has a 1 in "J" and a 0 in "B." We conclude

"J" > "B"

If we wish to compare two words which begin the same but end differently as in the comparison,

"ANTS" > "ANTHRACITE"

we can simply compare the ASCII-8 value of "S" to the ASCII-8 value of "H". We find that

$$S = 1\ 0\ 1\ \boxed{1}\ 0\ 0\ 1\ 1 > 1\ 0\ 1\ \boxed{0}\ 1\ 0\ 0\ 0 = H$$

is true, and thus, the original comparison, "ANTS" > "ANTHRACITE," is true as well.

For words of different lengths, we pretend that the shorter word has blanks attached onto the end of the word so that each of the words is viewed as having the same length. The ASCII-8 code assigned to a blank is always less than the code of any letter of the alphabet. So if we are comparing two words that begin identically and only differ when one of the words runs out of letters to compare, the shorter word will be less than the longer word. Again, this is as in a dictionary. For example, the comparison,

"CAT" < "CATS"

is true because the "fourth" letter of the two words are a space and an "S." (The ASCII-8 code for a space is less than the code for "S.")

C	A	T	Space
C	A	T	S

Figure 5-2: "CAT" < "CATS"

Computers use a code for the symbols to correspond to our expectations about the order of the letters of the English alphabet, one code, the ASCII-8 code, arranges binary representations so that

space <"A" < "B" < "C" < ··· < "Z"

This alphabetic ordering allows us to compare character strings just as we would look up two words in a dictionary. However, we may encounter certain symbols that are not listed in the alphabet of English letters. For example, we may choose to compare symbols such as (, }, ?, or !. We may also compare character strings that contain digits. The digits all follow the letters of the alphabet in the ASCII-8 code. Also, the ASCII-8 code is assigned so that

"0" < "1" < "2" < ··· < "9"

Thus, the following comparison is true:

"318A" > "215B"

In addition, BASIC distinguishes between uppercase and lowercase letters in a character string. Uppercase letters and lowercase letters have different ASCII-8 codes. Lowercase letters follow the uppercase letters in the ASCII-8 code. So, for example, the following comparisons are true:

"yes" <> "YES"
"Yes" > "YES"

Keep in mind that variables involved in comparisons must already have assigned values. Also, note that programs should not compare numeric values to character strings and vice versa. For more examples of BASIC comparisons and their results, see Table 5-3.

Comparisons	Results
5 > 3.9	True
-3 < -2	True
“cat” >= “cats” space < “s”	False
“5B” <= “and” “5” < “a”	False
“hello” = “HELLO” “h” > “H”	False
“conf-90” < “conf-91” “0”<”1”	True
“Jane” < “jane” “J” < “j”	True

Table 5-3: Comparisons and Results

On Your Own

1. Evaluate the following string comparisons as true or false. When evaluating variable comparisons, assume the following assignments: name1 = “Smith”, name2 = “Jones”, name3 = “Smithe”.

   “dog” < “dogs”

   name2 <= name1

   “house” = “HOUSE”

   “(example)” > “example”

   “6G” >= “7G”

   name1 <> name3

6.6 String Comparisons

Now that we know how characters are represented and how their codes are ordered so that we can compare strings, we need to list all the relational operators and their meanings for string comparisons. See Table 5-4.

Operator	Meaning
=	identical to
<>	different from
<	precedes alphabetically
>	follows alphabetically
<=	precedes alphabetically or is identical to
>=	follows alphabetically or is identical to

Table 5-4: Relational Operators in String Comparisons

Just as with numeric comparisons, string comparisons can be performed on either character strings or on variables representing character strings. For instance, the following are valid BASIC string comparisons assuming the character variables name1, phone1, and phone2 have been assigned values.

"cat" > "ball"
"sale-89" <= "sale-90"
name1 >= "Jane"
phone1 <> phone2

6.7 Conditional Statements

A fundamental feature of BASIC is the syntax that allows conditions to be evaluated by a program as a means of controlling the order of execution of statements in a program. Often decision making is done based on a statement that involves a comparison. Comparisons are evaluated to be either true or false. The idea is to be able to carry out certain actions when a condition evaluates to be *true*. This kind of statement is called a **conditional statement**. In BASIC, conditional statements are implemented using an **If** block.

5.8 Simple If Blocks

The syntax of a simple **If .. Then .. End If** block is:

SYNTAX

Simple Condition

If .. Then .. End If

If ***condition*** **Then**
    Action when ***condition*** **is true**
End If

EXAMPLE: If surName >= "M" **Then**
    surNameEdited=surname &", "
    Print surNameEdited; firstName
End If

A block of code as shown in the **SYNTAX** box contains **markers**, which are necessary **keywords** that indicate the beginning and end of certain parts of a block of code. An **If**-block always contains at least three markers. These three markers are **If**, **Then**, and **End If**. Following the **If** marker is a *condition* that is evaluated. The condition is a comparison, either numeric or string. Following the condition is the **Then** marker that indicates the end of the condition and the beginning of the *true range*.

The true range consists of statements that are executed when the condition is true. We sometimes refer to **Then** as a guard for the true range that only lets the execution into the true range when the condition is evaluated to be true. Ending the true range is the **End If** marker.

When an **If** block is executed, the condition is first evaluated. If the condition is true, the actions specified in the true range are performed. The program then continues execution at the statement following the **End If** marker. If the condition is false, the program will continue execution of the program at the statement following the **End If** marker. See Figure 5-3 for an example of a simple **If** block.

```
a = Val(InputBox("Enter a number"))
If a > 0 Then
   sq = a * a
   Print a;" squared is "; sq
End If
```

Figure 5-3: Example of a Simple If block

In Figure 5-3, notice the spacing conventions used within an **If** block. The statements following the **Then** marker are indented. This convention clarifies which actions should be performed if the condition is true and also where the statement(s) in the true range of the **If**-block ends. One key to understanding programs involves identifying the different parts of the code. This identification process is made easier by a consistent indentation style for different kinds of statements. Regardless of the indentation conventions followed, every true range of a simple **If** ends with the marker **End If**.

On Your Own

1. Write a program that inputs the outside temperature. Use an **If**-block to check if the temperature is greater than 70. If it is, print a message saying "Wear shorts today."
2. Write a program that inputs the weight of a salmon caught. Use an **If**-block to check if the weight is greater than 10 pounds. If the salmon weighs more than 10 pounds, print a message saying "Great fish!"

5.9 The Else Option

Sometimes in an **If** block, you may want to specify one set of actions to be performed if the condition is true and a different set of actions if the condition is false. In this case, we use the **Else** marker as shown in the following SYNTAX box:

SYNTAX

Two Way Condition

If .. Then .. Else .. End If

```
If condition Then
        Action when condition true
Else
        Action when condition false
End If

EXAMPLE: If     surName >= "N" Then
                surNameEdited=surname & ", "
                Print surNameEdited; firstName
        Else
                Print "Name out of required range"
        End If
```

The condition in the example is true if the value of *surName* starts with any letter greater than "M" (after M in the English alphabet). The **Else** marker appears after the true range and marks the beginning of the *false range,* which consists of the statements to be executed if the condition is evaluated to be false. Ending the entire **If**-block as well as the false range is the **End If** marker.

If the condition is true, program control is passed to the first statement in the true range. The statements in the true range are then executed. When those statements have been completed, the program encounters the keyword **Else**, the program then continues execution at the statement following **End If**. If the condition is evaluated to be false, program control is passed to the first statement in the false range which is the first statement after the keyword **Else**. When those statements following **Else** and preceding **End If** have been executed, program control is passed to the first statement following **End If**.

Figure 5-4 contains an example of an **If .. Then .. Else .. End If** block. Notice the indentation for this block. It is helpful to follow these spacing conventions in programs you write.

```
answer = InputBox("Enter a word:")

If answer = "anthracite" Then
    prize = 100
    Print "Word matches! You win: ", prize
Else
    prize = -100
    Print "No match. You lose: ",prize
End If
```

Figure 5-4: Example of an If Block with an Else Option

As you can see from the code, the program is checking to see if an answer given matches the program's answer. If the answer is correct (the condition is true), you win $100. If the answer is incorrect (the condition in false), you lose $100. Observe that there are no $-signs in the code. A $-sign appears only in output when using a **Format** statement or including a $-sign in a string constant.

Example 5-1. Use an **If**…**Then**…**Else**…**EndIf** block to write a program that inputs two different numbers and prints the two numbers in decreasing order.

SOLUTION:

```
Rem Determine which of two unequal
Rem numbers is the larger.


Rem INPUT: Two numbers
Rem OUTPUT: The numbers entered in decreasing order
   Cls
Rem Get input
   num1 = Val(InputBox("First number:"))
   num2 = Val(InputBox("Second number:"))
Rem Decide which is large
   If num1 > num2 Then
      Print num1; " is greater than "; num2
   Else
      Print num2; " is greater than "; num1
    End If
```

On Your Own

1. Write a program that inputs a person's age. Print the age. Also print the message "Vote Nov. 5!" if the age is greater than 18. Otherwise, print "You're too young to vote."
2. Write a program that inputs a number indicating which meeting group you are in. If your number is less than 3, print "You get special seating." If the group number is 3 or greater, print "You are on standby."

5.10 Compound Conditional If Blocks

Simple conditions do not always reflect what is needed to be known in a program. In such cases, an **If** block based on a combination of conditions being true or false is needed. For example, if you want to know if a person is male and older than 18, you will need two conditions to be satisfied at the same time. The logical operators used to combine simple comparisons are **And**, **Or** and **Not**.

You have seen how to determine whether a numeric or string condition is true. To determine whether a compound condition involving two conditions is true or false, the computer must determine the truth value for each of the simple conditions separately and then combine those truth values as indicated in Table 5-5 using the logical operator **And**.

And Condition	Truth Value For Condition 1	Truth Value For Condition 2	Truth Value For Condition 1 **And** Condition 2
POSSIBLE	False	False	False
TRUTH	False	True	False
VALUE	True	False	False
	True	True	True

Table 5-5: Truth Table for the And Operator

Table 5-5 is called a **truth table** for **And** since it represents in tabular form how the truth values of the individual conditions are combined to find the truth value of the compound condition.

Figure 5-5 shows an **If** block with two conditions joined by the **And** operator. On each side of **And** there must be a correctly written condition.

```
color3="orange"
color1=InputBox("Enter a color")
color2=InputBox("Enter a color")
If (color1 = "red") And (color2 = "blue") Then
    color3 = "purple"
End If
```

Figure 5-5: Code with If Block and the And Operator

In the example in Figure 5-5, both of the conditions must be true in order for the assignment of "purple" to *color3* to be executed. Also, notice in Figure 5-5 that if more than one condition is tested, each condition is surrounded by parentheses. Surrounding the simple conditions with parentheses can make them easier to read.

WARNING: As another example, suppose we want to test whether a variable has a value greater than 0 and less than 200. The condition could be written as:

$$0 < \text{years} < 200$$

as one does in algebra. Although this is perfectly correct in mathematical contexts, BASIC does not allow this syntax as a substitute for what was shown in Figure 5-5 for testing the two colors. This condition must be written as

$$(0 < \text{years}) \text{ And } (\text{years} < 200)$$

Along the same lines, we can also use the operator **Or** between conditions. When **Or** is used to join two simple conditions, at least one of the individual conditions must be true in order for the compound condition to be true. See Table 5-6 for the truth table of the **Or** operator.

Or Condition	Truth Value For Condition 1	Truth Value For Condition 2	Truth Value For Condition 1 **Or** Condition 2
POSSIBLE	False	False	False
TRUTH	False	True	True
VALUES	True	False	True
	True	True	True

Table 5-6: Truth Table for the Or Operator

The **Not** operator negates the value of a comparison: if the comparison is true then the **Not** operator changes the result to false and vice versa. The truth table for **Not** is shown in Table 5-7.

Not Condition	Truth Value For Condition	Truth Value For **Not** Condition
TRUTH	False	True
VALUES	True	False

Table 5-7: Truth Table for the Not Operator

An example of using the **Not** operator is shown here:

Condition 1: miles < 5000
Condition 2: Not (miles < 5000) is the same as miles >= 5000

The condition

Not ((A < 3) **And** (B > 4))

is logically equivalent to

Not (A < 3) **Or Not**(B > 4)

which is logically equivalent to

(A >= 3) **Or** (B <= 4).

This condition is not the same as

Not (A < 3) **And Not** (B > 4)

as you can readily see if you let A = 2 and B = 3.

When you want to verify a variable, say *criterion*, has one of a small set of possible values, such as *criterion* must be 1 or 2 or 3, you can check this by first asking if *criterion* has one of the values as:

(criterion = 1) **Or** (criterion = 2) **Or** (criterion = 3)

If you are only concerned that *criterion* has some legal value and action is only required if *criterion* does not have a required value, use the previous condition along with **Not** as:

Not ((criterion = 1) **Or** (criterion = 2) **Or** (criterion = 3))

The compound condition inside the outer most pair of parentheses is true if *criterion* has a required value. The **Not** operator makes the condition false in that case. If *criterion* does not have one of the required values, the compound condition inside the parentheses evaluates to false since each of the simple conditions is false and the **Not** operator outside makes the whole condition true.

On Your Own

1. Explain why the following compound conditions are incorrect.

   a) temp > 60 **And** < 75
   b) rating = "G" **Or** "PG"
   c) answer1 **And** answer2 = 63
   d) 12 <= num1 <= 17

   Example a) is particular enticing when you want to test whether a variable has one of a set of values. In BASIC this can be done only as

   temp > 6 **And** temp < 75

2. For a variable XYZ with a numerical value, write conditions to test if the value of XYZ is: (a) greater than 30; (b) less than 80; (c) greater than or equal to 91; (d) less than or equal 14; (e) greater than 50 and less than or equal to 90; (f) greater than 60 or less than 20; (g) greater than 70 and greater than or equal to 25; (h) not less than 40 (use Not); (i) is one of the values 1, 2, 3, or 4; (j) is not one of the values 3, 5, 7, or 11 (use Not).
3. For a variable ABC with a string as its value, write conditions to test if the value of ABC is: (a) greater than "abc"; (b) less than "race"; (c) greater than or equal to "class"; (d) less than or equal to "Smith"; (e) greater than "base" and less than or equal to "call"; (f) greater than "expert" or less than "alfa"; (g) greater than "same" and greater than or equal to "equal"; (h) not less than "major" (use Not); (i) is one of the strings "ab", "cd", "sf", or "g"; (j) is not one of the strings "sue", "jane", "bill" or "sam" (use Not).
4. For variables W, X, Y, Z each containing a numerical value, write conditions to test whether the following questions about the values are true or false: (a) the value of W is greater than either the value of Y or the value of Z; (b) The value of Z is less than or equal to any of the values of the other variables; (c) the value of Z is greater than either the value of W or the value of X while the value of Y is not less than the values of either X or Z; (d) the values of W and X are each greater than the value of Z while the value of Z is strictly less than the value of Y and the value of Z is not equal to the value of X. Would there be any difference in the conditions if the variables contained strings instead of numeric values?

5.11 Multi-case If Blocks

A university often needs to process student records for current students according to the year of expected graduation. Typically, this problem becomes at least four different problems depending on the graduation year. If the information about a student contains a variable named code whose value indicates the graduation year of the student, the following commands use the coded value to determine a graduation year:

```
If code = 4 Then
   YEAR = 2014
End If
If code = 3 Then
   YEAR = 2015
End If
If code = 2 Then
   YEAR = 2016
End If
If code = 1 Then
   YEAR = 2017
End If
```

We have simply written four separate **If** blocks, one for each case. Each value of *code* yields a value of true for exactly one of the four conditions. However, this code is inefficient since the computer will check each condition, even if the first condition is true. For example, if *code* = 4 is true, we assign 2014 to YEAR and then ask if the *code* has value 1, 2, or 3. Thus, for efficiency, we can subdivide an **If** block to include more than two cases by using the syntax associated with the **ElseIf** marker. We see this construction in the next example.

Example 5-2. Write a program that inputs a student's name and class code. Associate the input class code with a graduation year and display the year of graduation. Use an **If** block with the **ElseIf** marker to produce a case statement.

SAMPLE OUTPUT:

Bart Simpson

Class of 2015

SOLUTION:

```
     Rem Inputs a student's name and
     Rem class code and output the student's name
     Rem and graduation year.


     Rem INPUT: Name and class code
     Rem OUTPUT: Name and graduation year
       Cls
     Rem Get input
       nam = InputBox("Student's name.")
       code = Val(InputBox("Class code (1, 2, 3 or 4)."))
     Rem Determine year
A      If code = 4 Then
         yr = 2014
B      ElseIf code = 3 Then
         yr = 2015
C      ElseIf code =2 Then
         yr = 2016
D      Else
         yr = 2017 'assume code has a valid value
E      End If
     Rem Display output
F      CurrentY = 8 * font_height
       CurrentX = 30 * font_width
       Print nam
       CurrentX = 30 * font_width
       Print "Class of"; yr
```

COMMENTS: In the solution program, the user is asked to enter a class code with value 1, 2, 3 or 4. This input is assigned to the variable *code*. The **If** block beginning in **A** and ending at **E** then acts as a case statement and associates each code with a graduation year: 2014 if *code* = 4; 2015 if *code* = 3; 2016 if *code* = 2; and 2017 if *code* = 1.The marker pairs in the **If** block are **If .. Then (A), Then .. ElseIf (A-B), ElseIf .. Then (B)**, **Then .. ElseIf (B-C), ElseIf .. Then (C), Then .. Else (C-D)**, and **Else .. End If (D-E)**. Starting with **A**, the value of code is compared to 4. If *code* = 4 is true, then *yr* is assigned the value 2014 and control passes to **F**. If *code* = 4 is false, then the condition *code* = 3 is tested. If *code* = 3 is true, then *yr* is assigned the value 2015 and control passes to **F**. If *code* = 3 is false, then the condition *code* = 2 is tested. If the condition *code* = 2 is true, then *yr* is assigned the value 2016 and control passes to **F**. If *code* = 2 is false, the code between the two markers **Else** and **End If** is executed. In this case *yr* is assigned the value 2017 and control then passes to **F**.

The syntax of BASIC requires complete conditions between **If .. Then** and between each **ElseIf .. Then**. There is no condition following **Else** and this is interpreted to be any case not singled out by the explicit conditions tested. For example if *code* could be one of the values 1, 2, 3, 4, 5, or 6, then for all the occurrences of the values 1, 5, and 6, the variable *yr* would be assigned the value 2017. If *code* would have an incorrect value, you could test directly for 1 and use the else code to indicate an error. Using the **ElseIf** marker to write this block is more efficient than writing four separate **If** blocks, one for each of the four situations. Rather than testing four different conditions, the variable *yr* receives a value as soon as the *code* is identified, and then program control drops out of the **If** block. The **ElseIf** marker must be written as a single word, not as **Else If**.

We have seen several forms of the standard **If** block: the **If** block with the **And** operator, the **If** block with the **Or** operator, and the **If** block with more than two cases. The **If** block gives a powerful tool for program design.

SYNTAX

Multi-Case Ifs

If .. Then .. ElseIf .. Else .. End If

```
If condition1 Then
        Action1
ElseIf condition2 Then
        Action2
ElseIf
.
.
.
Else
        Action
End If

EXAMPLE:
If num >= 90 Then
        Print "A"
ElseIf num >=80 Then
        Print "B"
Else
        Print "C"
End If
```

On Your Own

1. Explain why the following **If** blocks are invalid.

(a)
```
If total = 90 Then
   Print total
```

(b)
```
If i < 15
      Print "Fifteen"
   End If
```

(c)
```
If mat = 3 Then
   sum = 12
ElseIf mat = 4 Then
   sum = 15
Else mat = 6 Then
   sum = 18
End If
```

2. Identify the output of the code:

```
If x > .9 Then
        die = 1
ElseIf x > .8 Then
        die = 2
ElseIf x > .7 Then
        die = 3
Else
        die = 6
End If
```

For input values x = .67, .81, .45, and .94.

Example 5-3. A local internet cafe charges $0.50 for the first 30 minutes online and $0.15 for each additional 10 minutes or fraction of an additional 10 minutes. Using an **If** block, write a program that inputs the number of minutes of the session and outputs the cost and length of the session.

SAMPLE OUTPUT:

The cost is $1.80 for using the internet for 47 minutes.

SOLUTION:

```
    Rem Calculate the cost for internet connect session. The rate is $0.50
    Rem for the first 30 minutes and $0.15 for each additional ten minutes

    Rem or part of an additional 10 minutes.



    Rem INPUT: Number of minutes logged on.
    Rem OUTPUT: The cost and the number of minutes charged
      Cls
    Rem Get input
      minutes = Val(InputBox("Minutes of internet use:"))
    Rem Calculate cost
A   If minutes = 0 Then
      cost = 0
B   ElseIf minutes <= 30 Then
      cost = 0.5
    Else
C     over = (minutes – 30) / 10
      If over > Int(over) Then
        over = Int(over) + 1
      End If
D     cost = 0.5 + over * 0.15
    End If
    Rem Display output
      Print "The cost is "; Format(cost, "currency");
      Print " for using the internet for"; minutes; "minutes."
```

COMMENTS: The condition in line **A** determines whether or not we logged in correctly. If the length of time was zero minutes, there is no charge. The condition in line **B** ensures that if the session was completed within 30 minutes, the cost of the session is $0.50. Otherwise, an additional charge is calculated for each 10 minutes or fraction of 10 minutes over the original 30 minutes (lines **C** and **D**). The **ElseIf** syntax does the same thing as

```
If minutes = 0 Then
   cost = 0
End If
If (minutes > 0) And (minutes <= 30) Then
    cost = 0.5
End If
If minutes > 30 Then
   over = over / 10
   If over > Int(over) Then
      over = Int(over) + 1
   End If
   cost = 0.5 + 0.15 * over
End If
```

On Your Own

1. Write a program to handle a savings account withdrawal. Request from the user the current balance and the amount of the withdrawal. If the withdrawal is greater than the original balance, the program should display "Withdrawal denied." If the withdrawal is allowed, the program should then display the new balance. If the new balance is less than $50.00, the program should also display "Balance below $50.00".

5.12 Putting It All Together

1. Write a program that takes as input a sales person's name and number of sales for the day. If the number of sales is over 300, print the following message:

   Good job! You will get a bonus!

   Otherwise, print the following message:

   Too bad! Try harder next time!

2. Ajax drivers get $0.75 per mile for each of the first 300 miles they drive. They are reimbursed at the rate of $1.19 per mile for all miles over 300. Write a program to compute a driver's payment if the total number of miles driven is input.
3. Ajax keeps a log of the number of miles each truck has been driven. Every two months (8 weeks) the company totals up the miles for each truck to determine what maintenance is needed. Suppose that a truck that has been driven more than 350 miles is scheduled for an oil change, a lube, and a washing. If a truck has been driven up to 350 miles, just a wash is scheduled. Write a program to compute the total miles a truck was used and output both the total miles and the kind of maintenance to be scheduled for that truck.

4. Write a program to enter a student's name and percentage grade on an exam. The program should then display the student's name, percentage grade, and a corresponding letter grade. Use the following table as a guide to the cutoff points for the corresponding letter grades:

Cutoff Point	Letter Grade
90	A
80	B
70	C
60	D
below 60	F

5. Write a program that calculates the cost of sodas for a picnic. Sodas cost 21 cents each when fewer than 100 sodas are purchased. The cost is 19 cents per soda when 100 or more sodas are purchased. The user should input the number of sodas purchased. Modify the program so that the company ordering the sodas is identified.
6. Write a program that calculates the gross pay of an hourly employee if the hourly wage is $10.75 for the first 40 hours and 1.5 times that for all hours over 40. The user inputs the number of hours worked.
7. Write a program that inputs integers *a* and *b*. Determine if *a* divides *b*, i.e. there is no remainder. (Use the **Mod** operator: the value of *b* **Mod** *a* is the remainder of dividing *b* by *a*. For example, 6 Mod 4 = 2.)
8. Write a program that requests the lengths of three line segments as input and tells whether the lines can form a triangle. Three lines can form a triangle if the length of the longest line is less than the sum of the lengths of the other two lines.
9. Input the number of widgets a customer buys. Widgets cost $7.16 for the first eight and $6.78 for any others, compute the total cost of a customer's purchase.
10. Input the number of widgets a customer buys. The cost is $7.95 each for the first five, $7.65 each for the next five, and $7.02 for all additional widgets. Compute the cost for sales of 4, 6, 9, 11, and 21 widgets.
11. Using a grading scale of:

A	90-
B^+	88–89
B	80–87
C^+	78–79
D	0–69

Compute a grade for a score entered by the user.

12. For any number in the range 0–500 determine the quality of the value according to the following table:

0–125 Marginal
126–385 Acceptable
386–415 Well above average
416–500 Exceptional.

For output display the range for the category of the value entered and its value.

13. Suppose the mean (m) of a set of number is 86 and the standard deviation (stDev) is 16. For any value input determine its grade using the following table:

Less Than	Greater Than Or Equal	Grade
m – 2 * stDev		Not Passing
m – .75 * stDev	m – 2 * stdev	C
m + stDev	m – .75 * stDev	B
	m + stDev	A

14. An aptitude test has two parts. The score in each part range from 0–50. Enter a person's scores on each part and determine the evaluation given by the following table where the Part I score is given if the Part II score is above the range for that Part I score.

Part I	Part II	Evaluation
0–20	0–10	Retake in 30 days
21–35	11–25	Take practical test
36–50	26–50	Certification completed

6 Branching

Often in problem solving, the execution of an algorithm comes to a point at which the next steps of the processing depend on the current value of some variable. In fact, different values may require different kinds of processing or different blocks of code. This ability to switch execution to different blocks of code depending on the current state of the program is called **branching**. Branching exists in two forms: **conditional** and **unconditional**. In **unconditional branching** whenever a branching command is encountered, the program automatically switches control to the block of code indicated. On the other hand, when using an **If** block, we can implement **conditional branching**. In this case the program branches to the block of code indicated only when the condition located between **If** and **Then** is true.

A typical application of branching involves some type of processing a set of data values. Finding the average of a set of numbers uses this strategy. Obviously, the average cannot be computed until all the values have been summed and counted. The program if Figure 6-1 has one block of code that accumulates the values and counts how many there are. A second block of code is switched to when the summing is finished so that the average can be computed

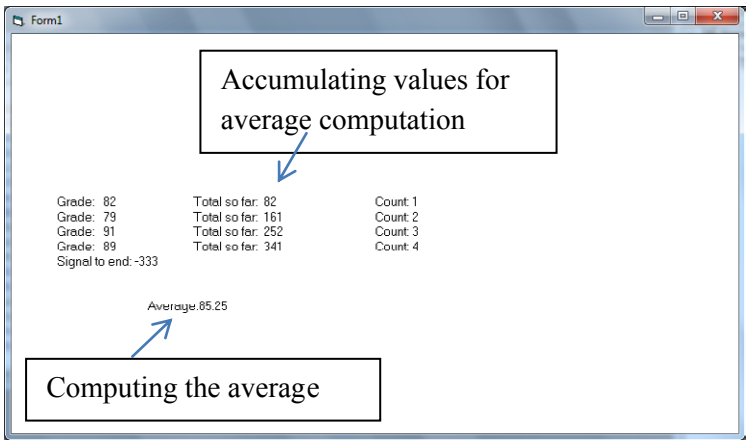


Figure 6-1: Summing and then Computing an Average

6.1 Line Labels

When branching is done in a program, there must be some way to indicate the program statement to which the program will branch. To identify a statement that is the object of a branching command, we use **line labels**. A **line label** in a program merely serves as a marker telling BASIC to remember the location of this statement in the program.

For clarity and uniformity, we use a restricted form for line labels. Except as a parameter for a transfer statement, a line label may only appear before the first executable instruction in a line of a program. Secondly, though line labels may be formed using the same rules as for forming variable names, the labels we use always begin with the capital letter *L* followed by an integer value of one or two digits.

Finally, a BASIC requirement is that the label must be followed by a colon (:). The colon is used to separate statements on a single line or to separate a line label from a statement. For example

L1 : X = 2

is valid because the line label and the assignment statements are separated by the syntactic unit (:) that is used only for that purpose. If a line label is being used as an argument in a branching command, no colon is needed. The following are examples of line labels:

L1
L13
L77

SYNTAX

Line Label

Variable separated from a statement by a colon. The value of the line label is the address of the storage location for that line of code.

6.2 Unconditional Branching

In BASIC, **unconditional branching** results by using the **GoTo** command. This command allows branching from any statement in a BASIC program to any other statement identified with an appropriate line label. The **GoTo** command takes one line label as an argument. When a **GoTo** command is encountered, the control of the program is automatically and immediately transferred to the line specified by the line label. The following are valid **GoTo** statements:

GoTo L1
GoTo L13
GoTo L77

In a **GoTo** statement there is no colon following the line label because the line label is not being used to mark a line of code. In this case the value of the line label is the location of the next statement in the program to be executed.

SYNTAX

GoTo command

A command that transfers control to the location that is the value of the variable.

On Your Own

1. Explain why the following **GoTo** statements are invalid.

GoTo
GoTo L1, L3

The program in Figure 6-2 demonstrates unconditional branching with a **GoTo** statement and a line label. Pay close attention to the conventions used in line labeling.

```
     Rem Program to demonstrate the
     Rem GoTo command.
     Rem OUTPUT: text on the screen
          Cls
          Print "Hello World!"
          Print "How are you?"
A         GoTo L1
B         Print "I'm fine."
C    L1: Print "Goodbye!"
```

Figure 6-2: Unconditional Branching with GoTo

The output of the program is the following:

Hello World!
How are you?
Goodbye!

Notice in the execution of the program that line **B** was not executed. When the program executed the **GoTo** command in **A**, the control of the program jumped to the line labeled *L1* (**C**) where the program continued execution. This branching caused statement **B** to be skipped. One of the problems with using **GoTo** statements without being very careful is that you could cause an important statement to be skipped. This example program may seem trivial, but it does illustrate the semantics of the **GoTo** statement rather vividly. The **GoTo** command will be quite useful later as we discuss conditional branching and looping.

On Your Own

1. Using a **GoTo** statement and a line label, edit the following program to display only the message "Good Evening." The program should not display the message "Good Morning." Do not remove any lines from the program. You need only to add one **GoTo** statement and one line label in the correct positions.

```
Rem Program prints two lines of text
Rem but will be edited to print only one.

Rem OUTPUT: text on the screen
   Cls
   Print "Good Morning."
   Print "Good Evening."
```

2. Using a **GoTo** statement and two line labels, edit the following program to display only the message "Good Morning." The program should not display either the message "Good Evening" or the message "Good Day." Do not remove any lines from the program. You need only to add two **GoTo** statement and two line labels in the correct positions.

```
Rem Program prints two lines of text
Rem but will be edited to print only one.


Rem OUTPUT: text on the screen
   Cls
   Print "Good Evening."
   Print "Good Morning."
   Print "Good Day."
```

6.3 Repetition of Code

A fundamental problem that will be explored in this and especially the next chapter involves processing multiple pieces of data in the same way. For example, suppose a program needs to add up three data values. The program could be the one shown in Figure 6.3.

```
     Rem INPUT:Three values
     REM OUTPUT: Sum of the values entered
     REM Initialize accumulator
               sum = 0
     REM Process the first data value
A              x = Val(InputBox("Enter value"))
B              sum=sum + x
     REM Process the second data value
C              x = Val(InputBox("Enter value"))
D              sum=sum + x
     REM Process the third data value
E              x = Val(InputBox("Enter value"))
F              sum=sum + x
     REM Output sum
               Print sum
```

Figure 6-3: Summing Three Values

The first observation is that it is fortunate the program does not have to sum up 100 values! The second observation is that the two lines **A** and **B** are identical to the two lines **C** and **D** and the two lines **E** and **F.**

What would be convenient is to reuse lines **A** and **B** three times instead of writing the same code three times. Programming languages incorporate a way to do just this. We can use the same code over and over by incorporating conditional branching into the code.

6.4 Conditional Branching

In computer programming, **conditional branching** implies that program control is transferred from one statement to another only when a certain condition is true. In BASIC, conditional branching can be achieved by using the **GoTo** command within an **If** block. For instance, the code in Figure 6-4 produces conditional branching-branching only if the variable *switch* has the value 1 when the program executes this statement.

```
If switch = 1 Then
   GoTo L1
End If
```

Figure 6-4: Conditional branching

In the code of Figure 6-4, the condition *switch* = 1 is the test. If the condition is true, the **GoTo** command in the true range causes control to transfer to the line labeled *L1*. If the condition is false, the program continues execution at the statement following the **End If**.

To this point the programs shown have simply executed lines of code one after another in order from the start to the finish of the program. Usually a more complex program needs to repeat some of its lines of code several times in order to solve a problem. For example, in the next program, we want to add up a sequence of numbers that are entered by the user until the sum of the numbers entered exceeds 100. The program will repeat part of the code until the sum of the input values exceeds 100. This will be accomplished using conditional branching

Example 6-1. Using conditional branching, write a program that inputs and adds numbers until the total of the numbers input exceeds 100. The program should print the numbers entered and the final total.

SAMPLE OUTPUT:

Number entered: 5
Number entered: 9
Number entered: 11
Number entered: 46
Number entered: 60
The sum of the numbers is 131

SOLUTION:

```
    Rem Program inputs, displays, and adds numbers until the total
    Rem of the input values exceeds 100. The final total is then displayed.
    Rem INPUT: Numbers to add
    Rem OUTPUT: Input numbers and final total
      Cls
    Rem Initialize the total
A     total = 0
    Rem Enter a numeric value
    L1: num = Val(InputBox("Enter a number:"))
    Rem Process the value entered
      Print "Number entered:"; num
B     total = total + num
    Rem Check the total
C     If total <= 100 Then
D       GoTo L1
      End If
    Rem Display output
E     Print "The sum of the numbers is"; total
```

COMMENTS: In the program the variable *total* is set to zero (**A**) so that it can be used as an accumulator for the sum of the input values. The variable *total* starts at zero because it should be equal to the sum of the values processed when the program executes (**A**). We then ask the user to input a number and add that number to *total* (**B**). The value of *total* is then compared to 100 (**C**). If the value is less than or equal to 100, the condition is true, and the **GoTo** statement (**D**)- the true range of the condition-transfers the program control to the statement labeled *L1*. Execution continues with the statement at that line. The next number is then entered and processed. The input and processing code is repeated until the condition in **C** is false. When the condition (**C**) is false, the program continues at **E** and displays the output. This program is a typical example of the use of conditional branching in a BASIC program.

Though the **GoTo** command can be helpful in several situations, we do not recommend using it often. Frequent use of the command may create complicated programs that are difficult to understand, debug and change. Thus, use the **GoTo** command sparingly and carefully.

SYNTAX

Unconditional and Conditional Branching

Unconditional branching: Simply a **GoTo** statement that is executed whenever it is encountered

Conditional branching: A **GoTo** statement that is in the true range of an If statement so that it may or may not be executed

On Your Own

1. Using one or more **GoTo** statements, write a program that displays on the screen the list of even numbers from 2 to 20 as well as the sum of these numbers. Also print the partial sum at each stage of the computation. After displaying 20, the program should print the message "Task complete."
   Hint: Use a variable called *num* and initialize it to 2. Also initialize a variable *total* to zero to use to accumulate the sum of these numbers. Also, somewhere in your program include the following line of code:

   $$num = num + 2$$

   so that the program can test the current value of *num* to determine whether the termination condition is satisfied.

6.5 Repetition a Number of Times

In addition to using conditional branching with the condition asking about the value of some variable in the program, conditional branching is often used when the condition involves how many data values have been processed. By knowing ahead of time how many data items are going to be processed, the condition can just use a counter that knows how many data values have been processed to know when all the data has been processed. An example of this technique is shown in Example 6-2.

Example 6-2. Input the 10 scores for the May baseball games of the Lions. Determine how many games the Lions won in May.

SOLUTION:

```
  Rem Program inputs 10 baseball game scores and determines
  Rem how many games the Lions won
  Rem INPUT: 10 game scores each consisting of a score
  Rem for the Lions and a score for the opponent.
  Rem OUTPUT: The scores and number of games the Lions won
     Cls
  Rem Initialize counters for games won and games played
A    gamesPlayed=0
     gamesWon = 0
  Rem Header for scores
     Print "Lions score","Opponent's score"
  Rem Enter a game score
  L1: lions = Val(InputBox("Enter a score:"))
      opponent=Val(InputBox("Enter score:"))
      Print lions, opponent
  Rem Find out who won this game
      If lions > opponent Then
B       gamesWon = gamesWon + 1
     End If
     gamesPlayed = gamesPlayed + 1
C    If gamesPlayed < 10 Then
D      GoTo L1
     End If
  Rem Display output
E    Print "The Lions record for May is: "gamesWon;"-";gamesPlayed
```

6.6 Sentinels

Programs normally require the user to be able to input a different number of data items each time the program is run. For example, the user may want to average all the grades for an exam for two different classes that each has a different number of students. If a program always processed the same number of data items, you would have to write a separate program for each different class size. The program would be more effective if it were able to ask the user after each piece of data is entered and processed if there is more data to be entered. Depending on the user's response, the program will either process more data or proceed with the computation that follows the input of all the data items. Regardless of the number of pieces of data for a particular run of the program, the same program will always be used.

To make this process more efficient, we often use a **sentinel**. A **sentinel** is a special data value entered by the user to signal the end of the valid input. A program that uses a sentinel tests each value that is input to see if it is the value that signals that all the input data has been entered.

To implement a sentinel, we add an **If** statement following the **InputBox()** line. The **If** statement is designed to test a condition that asks whether or not the value just entered was the signal to end the input operation. In case the value is not a signal to end the input operation, the program will know there is a piece of data to process. When the user enters the sentinel, the program will know to discontinue the processing of data. After determining that all the data has been entered, program control is then transferred to the statement that is designed to do any summary computation or terminate the program. A **sentinel** has the property that it is not a value that will ever be found in the user's list of input values. For example, if the user is entering grades with a range of 0-100, then any negative number or integer greater than 100 could be used as a sentinel. If one attempted to use 38 as a sentinel, there is a chance that the data could include that value. In such a case, the program would terminate when 38 was entered rather than when all the data had been entered. If the input involves names, a string such as "END" could possibly be a sentinel. The value of the sentinel is chosen by the programmer and is conveyed to the user in the prompt message for the **InputBox()** command.

Example 6-3. Write a program that allows a user to input the grades for as many students as he/she likes. Output a list of the grades. Calculate and display the average of all the grades. Use a sentinel to end the user's input.

SAMPLE OUTPUT:

Grade entered: 86
Grade entered: 77
Grade entered: 94
Grade entered: 91
Grade entered: 83
The average grade is 86.2

SOLUTION:

	Rem Program allows a user to input the grades for any number of
	Rem students. The program calculates and displays the average of
	Rem the grades
	Rem INPUT: Arbitrary number of grades and sentinel value
	Rem OUTPUT: Each grade and the average of all the grades
	Cls
	Rem Initialize num (number of grades) and
	Rem total (sum of all the grades) since the average will
	Rem be total / num after all input values are entered
	total = 0
	num = 0
	Rem Input
A	L1: grade = Val(InputBox("Enter grade (or -999 to end):"))
B	If grade = -999 Then ' -999 is a sentinel
C	GoTo L2
	End If
	Rem Process
	Print "Grade entered: "; grade
D	total = total + grade
E	num = num + 1
	Rem Repeat
F	GoTo L1
	Rem User entered -999 so finish program
	L2: ave = total / num
	Rem Display output
	Print "The average grade is "; Format(ave, "standard")

COMMENTS: Notice in **A** that the program tells the user what value to enter when he/she is done inputting data. There is no way the user can guess the sentinel, so the program needs to include that information. In this case the value -999 is indicated as the sentinel. When a value is entered, it is compared to -999. If the input value is not -999, *grade* is processed in **D** and **E**. The **GoTo** statement in **F** then causes the program to transfer control to the statement with the line label *L1* where the user is asked to enter another grade. This is an unconditional transfer. When the user finally enters –999, the condition **B** is true so the program transfers control to the statement with the line label *L2* where the average of the grades is calculated. The program then continues to the end. The transfer of control to *L2* is a conditional transfer since it only happens when the sentinel is entered making the condition *grade = -999* true.

When using a sentinel in your programs, it is important to choose a sentinel that is not a possible input value. It is equally important to tell the user what the sentinel is.

6.7 Prompt and Echo

In a program, when we prompt users to enter data for the program to process, we should make an effort to be sure that the input data is correct before we process it. Often, users unintentionally enter incorrect data. For example, a typing error is not uncommon when entering data. If a user inputs data that is incorrect in value, the program may run but produce an incorrect answer. To deal with this important problem, we introduce a coding convention called **prompt and echo**.

Prompt and echo is a coding convention that assures as best as possible that the user has entered correct data when prompted for input. Each time a user is prompted and inputs a piece of data, the data is immediately displayed on the screen and the user is asked if it is correct. If the user answers that the given data is correct, the program continues and processes the given data. If the user answers that the data is incorrect, the program ignores the incorrect data and prompts the user again for a value. Only after the user has confirmed that the data entered is a correct value does the program continue and process the data.

Example 6-4. Write a program that computes a bank account balance after a transaction has been made. Have the user input the account's previous balance, the transaction code, and the amount of the transaction. The transaction code can be either a "D" for deposit or a "W" for withdrawal. Use the prompt and echo convention to check the user's input. Allow the user to process more than one transaction.

SOLUTION: The solution to this problem will have several parts that will be more easily understood if they are dealt with separately. The first part of the program asks the user to enter the account's previous balance and uses the prompt and echo technique to check that the input is correct. The code in Part 1 shows this part of the program.

```
    Rem Program computes a bank account balance
    Rem after a transaction has been made.
    Rem INPUT: Balance, transaction code and amount of transaction
    Rem OUTPUT: The final balance
      Cls
A   L1: balance = Val(InputBox("Enter current balance."))
    Rem Prompt and echo (check if data is correct)
B     p1 = "You entered " & balance & ". Is this correct? (Y/N)"
C     answer = InputBox(p1)
D     If (answer = "N") Or (answer = "n") Then
E       GoTo L1
      End If
```

Part 1

In **A**, the user is prompted for the current balance which is a numeric value. Consequently, the **Val()** built-in-function is used with **InputBox()**. When the user enters the balance, the program creates a prompt (**B**) by concatenating text and the value just entered. The &-operator merely joins the strings *"You entered* ", a string version of balance, and the string "Is this correct? (Y/N)" to form a single string *p1*. The numeric value of balance is converted to a string of symbols before the concatenation so that all the parts of the statement are strings. In **C**, the program displays *p1* as the prompt (thus echoing the data) and asks the user if the input data is correct. If the user responds with a negative answer ("N" or "n"), the **GoTo** statement in **E** sends the program control to the line labeled *L1*, causing the input process to begin again for the variable *balance*. If the user responds to the question with anything other than "N" or "n**,**" the program assumes that the input data is correct and continues to the next part of the program.

Notice that the condition in **D** allows the user to enter either a capital letter or a lowercase letter. Since capital letters and lowercase letters have different ASCII-8 values, using **Or** avoids confusion by allowing the user to enter either "n" or "N." Otherwise, entering "n" when the condition only checked for "N" would make the condition false and the user would be required to reenter a correct value.

The second part of the program processes the user's deposits and/or withdrawals. This section of the program inputs two values that indicate the type of transaction (*code*) and the amount involved (*amount*). After entering this data, a prompt and echo is used to determine if the values are correct. The code in **Part 2** shows this code.

	Rem Data is correct. Prompt for more data.
F	L2: code = InputBox("Enter D for deposit or W for withdrawal.")
G	amount = Val(InputBox("Enter amount."))
	Rem Prompt and echo (check if data is correct)
H	p2 = "You entered " & code & " and " & amount & "."
	p3 = "Is this code and amount correct? (Y/N)"
	answer = InputBox(p2 & p3)
	If (answer = "N") Or (answer = "n") Then
I	GoTo L2
	End If

Part 2

The user is prompted for the code and the amount of the transaction (**F** and **G**). The **prompt and echo** convention begins at **H** as a string of words and a numeric value represented as symbols is constructed to form a prompt for the **InputBox()** command. This prompt is fairly long so it is built by concatenating the values of the string variables *p2* and *p3* with a string conversion of the numeric values *code* and *amount*. Again, the user is asked if the data is correct. If the input data is not correct, the **GoTo** statement in **I** sends the program to the line (**F**) labeled *L2*. The input process for a code and an amount is then repeated. If the user responds with anything other than "N" or "n," the input data is processed.

The third part of the program performs calculations with the data. The output is then displayed. The code in **Part 3** shows how this is done.

```
    Rem Data is correct. Calculate balance.
J     If (code = "D") Or (code = "d") Then
        balance = balance + amount
K   ElseIf (code = "W") Or (code = "w") Then
        balance = balance – amount
    Else
L     Print "Incorrect Code Value"
    End If
    Rem Display output
    Cls
    CurrentY = 8 * font_height
    CurrentX = 25 * font_width
M   Print "The new balance is " ; Format(balance, "currency")
```

Part 3

The code in this section contains a multi-case **If** block that matches the input transaction code to the appropriate calculation. In **J**, *code* is tested to see if is equal to "D" or "d." If the condition is true, the

transaction is a deposit, and the transaction amount is added to the balance. In **K**, *code* is tested to see if the transaction is a withdrawal. In that case, the transaction amount is subtracted from the balance. The code in **L** guarantees that if an invalid code is entered, the balance will not be changed. In **M** the new balance of the account is displayed.

You might ask why code such as

```
            If (code = "D") Or (code = "d") Then
A                   balance = balance + amount
            Else
B                   balance = balance – amount
            End If
```

was not used. In fact, this simple **If…Then…Else…End If** could have been used. As a matter of style and for the purposes of promoting correctness, this simpler code was not used because it really says if *code* is equal to "D" or "d," do **A**; and if code has any other value, do **B**. When we know that *code* will have fixed values, it is better programming practice to make the program guarantee the correct value is being processed. If an additional operation is added, it will be easier to add the code. If the **ElseIf** is not used and another operation is added,

balance = balance – amount

would be executed for both a withdrawal and the additional operation. This may well not be the processing the new operation needs.

On Your Own

1. Simulate the code of Example 6-1 parts 1-3 and process the following data:

   Previous Balance = 135.78
   Transaction 1: withdraw 55.67
   Transaction 2: deposit 76.88
   Transaction 3: deposit 335.67
   Transaction 4: withdraw 66.34

6.8 User Interrogation Technique

The final part of the program offers the user the option to process another transaction. This technique is often used to verify that a user really wants to exit a program. The code in **Part 4** shows this.

```
    Rem Interrogate user
M       answer = InputBox("Process another transaction? (Y/N)")
N       If (answer = "Y") Or (answer = "y") Then
O         GoTo L2
        End If
    Rem display output
P       Cls
        CurrentY = 10 * font_height
        CurrentX = 25 * font_width
        Print "Final balance: "; Format(balance, "currency"); " Thank you!"
```

Part 4

In this section, the program asks the user in **M** if there is another transaction to process. If the user answers positively, the **GoTo** statement in **O** causes the program to continue execution at the line labeled *L2* (**F** in **Part 2**). Otherwise, the program continues to execute at **P** and displays a message indicating the program is ending. Notice in the condition in **N** that the program again allows the user to enter upper or lower case letters.

On Your Own

1. Enter the code of Example 6-4 parts 1-4 and run the program with data you create.

6.9 Putting It All Together

1. Use conditional branching to write a program that displays the list of integers from 30 to 50 along with their squares. Be sure to include text to make the output clear. For instance, your program's output may look something like this:

   30 squared equals 900
   31 squared equals 961
   ⋮
   50 squared equals 2500

   Hint: Initialize a variable to 30. Rather than using an **InputBox** function, you can just increment by one to get the next number to square.
2. Use conditional branching to write a program that counts backwards and displays the numbers from 10 to 0.
3. Write a program that inputs a series of grades and outputs each grade after it is entered. After each grade is processed, ask the user if there are any more grades to be entered. At the end, output the highest grade.
4. Write a program that inputs the ages of several people. Calculate the average age of the people. Be sure to implement prompt and echo when acquiring the data needed.
5. The mean of a set of N numbers is 62. Enter the N numbers and determine how many are larger than the mean. Use conditional branching to test how many numbers have been processed so that the program terminates after processing N numbers.
6. Enter the monthly total of sales for Fine Wheels. Sum and print the value of all the sales. Also calculate the most expensive car price for this month.
7. Enter N names together with the two symbol state code for the person's home state. Output any name with state code NJ, PA, or WA. Count and output the number of names that satisfy the condition.

8. Enter N measurements of weekly rainfall. Use conditional branching with a sentinel to sum and print the total of all the measurements. After entering the measurements, compute the average weekly rainfall for the values given.
9. Suppose the mean (m) of a set of number is 86 and the standard deviation (stDev) is 16. For any value input determine its grade using the following table:

Less Than	Greater Than Or Equal	Grade
m – 2 * stDev		Not Passing
m - .75 * stDev	m – 2 * stdev	C
m + stDev	m - .75 * stDev	B
	m + stDev	A

Using conditional branching determine the result for any number of test takers using a sentinel of -333 to terminate the processing

7 For .. Next Loops

In the previous chapter about branching, we studied programs that repeated certain sections of code. Using conditional branching to repeat code is one form of **looping** or **repetition**. Looping is an important programming structure that allows a fixed sequence of instructions to be executed under program control as many times as needed. At each repetition of the code, although the instructions may be the same, the instructions act upon variables whose values may be different. In this section, we discuss a form of looping or repetition in which the number of times the loop repeats is determined when the loop begins and the control of looping is built into the syntax rather than programmer written conditions.

The program shown in Figure 7-1 recaps the sales records for the ten most popular items during the last sales period. The input for each item consists of the name of the item, the number of units of this item sold, and the cost for one unit of the item. The total revenue for each item includes a 25% markup. The computation for each item is the same. By using loops the code need only be written once and used as many times as needed.

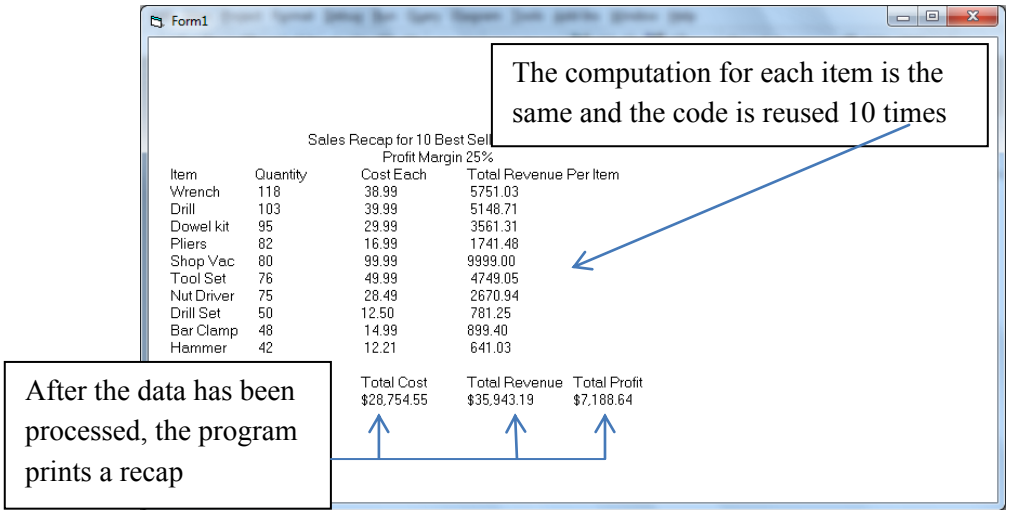


Figure 7-1: Sales Report

7.1 The For .. Next Loop

In the section on conditional branching, we saw that to repeat a block of code a fixed number of times we needed to initialize a counter, execute the block of code, increment and test the counter to see if the block of code is to be repeated, and then transfer control to the first statement in the block of code if it needs to be repeated. If the block of code is to be repeated, we execute it and repeat the incrementation, testing, and transfer steps until we have repeated the block of code the required number of times. When the required number of repetitions are completed, the program continues on. In this section we want to see how all this bookkeeping is built into BASIC syntax.

When we know exactly how many times a block of code should be repeated, we can use a **For .. Next** loop as shown in the syntax box instead of conditional branching as we saw in the previous chapter.

SYNTAX

For..Next Loop

```
For  variable = num1 To num2
            Actions to be repeated-body of the loop
Next variable
```

Note: variable is any variable name.

```
EXAMPLE:        For I = 3 To 5
                    X = InputBox("Enter a city name:")
                    Print X
                    Next I
```

Before examining the semantics of the **For .. Next** loop, we describe the syntax for this construct. A **For .. Next** loop always contains a **For** marker and a **Next** marker. The **For** marker indicates the beginning of the loop. Following the **For** marker is a variable which is referred to as a **control variable** to indicate its role in determining how many times the body of the loop will be repeated. We use *variable* as a representative for any variable name the programmer chooses to use in its place. The control variable is followed by an equal sign. On the right of the equal sign are two numeric values separated by the **To** marker. The values *num1* and *num2* can be numbers, defined numeric variables, or numeric expressions. In most cases *num1* is less than *num2*. The **body of the loop,** which is the code to be repeated**,** appears next. Following the body of the loop is the **Next** marker followed by the same variable referred to in the **For** statement.

Now we can use our understanding of this syntax to discuss the semantics of a **For .. Next** loop. When the loop begins executing, the **For** statement specifies that the control variable should be initialized to *num1*. The number range determined by the two values *num1* and *num2* following the equal sign marker is first checked by testing the condition where *variable* now has the value num1:

$$variable <= num2$$

If the condition is true, the body of the loop is executed with *variable* having its initial value. After the body of the loop has been executed and the **Next** statement is reached, the **Next** statement causes several actions, the first of which is to increment *variable* by 1. The new value of *variable* is compared with the value of *num2* by again checking the condition

$$variable <= num2$$

If the new value of the control variable is still less than or equal to the value of *num2*, the condition is true and the program control is transferred to the first statement in the body of the loop and the **body of the loop** again is executed. When the **Next** statement is encountered again, the increment and test procedure is repeated so that the program knows whether or not to transfer to the beginning of the body of the loop to repeat the body of the loop another time. When the control variable is incremented resulting in the new value being greater than *num2*, the program control is passed to the statement following the **Next** statement because the condition will evaluate to false. In total the body of the loop will execute

$$num2 - num1 + 1$$

times before the condition $num1 <= num2$ becomes false provided initially

$$num1 <= num2.$$

The values *num1, num1 + 1, … , num2* are called the range of the control variable. Figure 7-2 shows an example block of code that contains a **For .. Next** loop that will have the code (**C** and **D**) that is the body of the loop executed three times.

```
A   sumOfXes = 0
B   For I = 1 To 3
C      x = Val(InputBox("Enter a number:"))
D      sumOfXes = sumOfXes + x
E   Next I
F   Print "Sum of values:";sumOfXes
```

Figure 7-2: Example For .. Next Loop

In Figure 7-2, statement **A** initializes the accumulator variable *sumOfXes* to zero. Statement **B** marks the beginning of the **For .. Next** loop and initializes the counter *I* to 1. This line also specifies that the range of the counter will be from 1 to 3. Since the initial value assigned to *I* is less than or equal 3, the body of the loop is executed next. At **C** a number is input. Statement **D** adds the inputted number to the current value of *sumOfXes*. Statement **E** marks the end of the body of the loop and increments *I* by one. If *I* does not exceed 3, the loop repeats beginning at **C**. If *I* is greater than three, the program continues with **F** and displays the value of *sumOfXes*.

A complete trace of the execution of this program will help clarify what happens in a **For .. Next** loop. The trace shows the values of the variables after a statement has been executed. The complete trace is found in Table 7-1. (When you come to the bottom of the pair of columns labeled "Value of variables," move to the next column with this label and follow the continued execution of the program.)

Assume the input values are X1, X2, and X3.

Statement	**Value of Variables**			**Statement**	**Value of Variables**		
	SumOfXes	**I**	**x**		**SumOfXes**	**I**	**x**
A	0			D	X1+X2	2	X2
B	0	1		E	X1+X2	3	X2
C	0	1	X1	C	X1+X2	3	X3
D	X1	1	X1	D	X1+X2+X3	3	X3
E	X1	2	X1	E	X1+X2+X3	4	X3
C	X1	2	X2	F	X1+X2+X3	4	X3

Table 7-1: Trace of the Program in Figure 7-2

The actions of a **For .. Next** loop are similar to conditional branching which uses a **GoTo** statement and a counter incrementing and testing step to repeat sections of code. See Figure 7-3 for two equivalent programs that perform looping. The first program loops by using conditional branching with a **GoTo** statement within an **If** block. The second program loops by using a **For .. Next** loop.

```
                Rem Using conditional branching
                Rem this program displays five
                Rem names input by the user.
                Rem INPUT: Five names
                Rem OUTPUT: Five names
                  Cls
                Rem Initialize i
A                  I = 1
                L1: nam = InputBox("Enter a name:")
                  Print "The name is "; nam
                Rem Increment I and decide to
                Rem input another name or end
B                  I = I + 1
C                  If I <= 5 Then
D                    GoTo L1
E                  End If
```

```
                Rem Using a For .. Next loop,
                Rem this program displays five
                Rem names input by the user.
                Rem INPUT: Five names
                Rem OUTPUT: Five names
                  Cls
                Rem Loop to input/display names
F                  For I = 1 To 5
                    nam = InputBox("Name:")
                   Print "The name is "; nam
G                  Next I
```

Figure 7-3: Equivalent Programs that Demonstrate Two Forms of Looping

Both of the programs in Figure 7-3 produce the output shown in Figure 7-4 with input Cosmo, Elaine, George, Jerry, and Norman.

```
The name is Cosmo
The name is Elaine
The name is George
The name is Jerry
The name is Norman
```

Figure 7-4: Output of the Programs in Figure 7-3

If we compare the two programs in Figure 7-3, we see that they perform the same task, but one program is much simpler than the other. By using a **For .. Next** loop in the second program, we reduce the length of the program and make the structure of the program clearer. In Figure 7-3, **F** in the second program replaces and performs the same tasks as both **A** and **C** in the first program. Remember the initial statement of a **For** .. **Next** both initializes the control variable and tests to see if the body of the loop should be executed. Also, **G** in the second program replaces and performs the same tasks as **B**, **C**, **D** and **E** in the first program. These two programs demonstrate well the advantages of a **For .. Next** loop. This syntax is a case of **encapsulation**. The initialization, incrementation, testing, and transfer are packaged or encapsulated in the **For** and the **Next** statements.

For another example of a **For .. Next** loop, see Figure 7-5.

```
For K = 2 To 10
   sq = K * K
   Print "The square of"; K; " = ";sq
Next K
```

Figure 7-5: Example For .. Next Loop

In Figure 7-5 notice the spacing conventions used within a **For .. Next** loop. The body of the loop should be indented to make it clear what instructions are repeatedly executed. Also notice in the example that the number range assigned to the control variable of the loop need not begin with zero or one. Any integer numbers can be used as values for *num1* and *num2*. Variables representing decimal values can also be used as values for the loop starting and ending values. Notice also that in Figure 7-5 that the control variable value can be accessed and used for calculations within the body of the loop. However, the control variable should never be altered within the body of the loop!

Example 7-1. Write a program that uses a **For** .. **Next** loop to calculate and display the sum of the numbers from 1 to 10.

SOLUTION:

```
Rem Use a For .. Next loop to display
Rem the sum of the numbers from 1 to 10
Rem OUTPUT: The sum of 1, 2,…,10


   Cls
Rem Initialize total
   total = 0
Rem Loop to calculate total
   For I = 1 To 10
     total = total + I
   Next I
Rem Display output
   Print "The sum of 1 to 10 ="; total
```

COMMENTS: If we think about how we would code this program using conditional branching, we should see that using a **For .. Next** loop is easier and more convenient. The program is easy to code, read and follow, and it does not seem to "jump around" as with **GoTo** statements.

On Your Own

1. Use a **For .. Next** loop to write a program that displays all the numbers from 10 to 20.
2. Use a **For .. Next** loop to write a program that displays the product of the integers in the range 18 to 41. Output the total product of those numbers. To calculate the product, initialize a variable outside the loop to have a value of one.

7.2 The Step Parameter

In the **For .. Next** loops that we just studied, the control variable in the loop was always incremented by one when the **Next** statement is encountered. In some situations, we may wish to increment the control variable by a different value. To do this, we can use the extension of the syntax for the **For** statement to include the **Step** marker.

SYNTAX

For .. Next With Step

For *variable* **=** *num1* **To** *num2* **Step** *increment*
Code to repeat-body of the loop
Next variable

EXAMPLE: **For** I = 1 **To** 30 **Step** 3
Print I, I^2, I^3
Next I

Some of the output:

I	I^2	I^3
1	1	1
4	16	64
7	49	343

The **Step** marker and an increment value follow the number range in the first line of a **For .. Next** loop. Following the keyword **Step** is a value or expression that specifies the amount by which to increment the control variable in the loop. This value can be any negative or positive value, a variable representing a numeric value, or an expression to be evaluated when the **For** statement is executed. When the **Step** parameter is used with a **For .. Next** loop, the loop executes as normal except that the control variable is incremented when the **Next** statement is encountered by the value specified by the increment in the **Step** option. If the increment value is positive, the number range must be from low to high *(num1 <= num2*) as it was previously. If the increment value is negative, the specified number range (*num1* To *num2*) must be from high to low (*num1 >= num2*) so that the counter will be within the range as it is decremented. The test will be

$$variable >= num2$$

with a negative increment value rather than

$$variable <= num2$$

with a positive increment value. For a negative increment value, the loop terminates when

$$variable < num2.$$

Keep in mind that if the **Step** parameter is not used, the counter is incremented by one, the default value. If the increment is negative and *num1<= num2*, the body of the loop will not be executed even once!

A	p=0 'initializes an accumulator
B	For I = 15 To 5 Step -4
C	p = p + I
D	Print p
E	Next I
F	Print "Final sum ="; p

Figure 7-6: For .. Next loop with Negative Increment (Step Size)

In the program in Figure 7-6 the increment is a negative integer. Thus, the number range specified is from 15 to 5 rather than from 5 to 15. The values assigned to *I* will be 15, 11, 7, and 3. Remember that when a negative counter is specified by the **Step** parameter, the number range must go from high to low so that the counter may be decremented rather than incremented. The idea is clearer when you trace the execution of this code as shown in Table 7-2. (Read this table following the directions given for Table 7-1.)

Statement	Variables		Statement	Variables	
	p	**I**		**p**	**I**
A	0		D	26	11
B	0	15	E	26	7
C	15	15	C	33	7
D	15	15	D	33	7
E	15	11	E	33	3
C	26	11	F	33	3

Table 7-2: Trace of program in Figure 7-6

The program Figure 7-6: For .. Next loop with Negative Increment (Step Size)7-6 prints the values of *p* in the body of the loop while the loop parameter takes on the values 15, 11, and 7. Notice that the last value for *I* when *p* is printed is 7 and not 5, even though the number range goes to 5. Since the decrement specified is minus four, only every fourth number will be assigned to *I*. When the control variable has reached 7, it is decremented again by four giving a value of 3. At this point, the control variable is less than 5, and the program continues execution at the first statement following the end of the loop.

On Your Own

1. Write a program to display the odd integers between 7 and 34 in increasing order. Repeat the problem but display the integers in decreasing order.

2. Write a program that computes the product of the even integers in the range N to 60. You can test if an integer is even by the condition

   $$I \bmod 2 = 0.$$

   What if N > 60?
3. Explain why the following **For..Next** loops are invalid.

```
(a) For k = 5 To 10
      Print 1 / k
    Next x
```

```
(b) For n > 30 To 15 Step -4
      p = p * n
    Next n
```

Example 7-2. Using a **For** .. **Next** loop with a **Step** parameter, write a program that calculates and displays the product of all the odd numbers from 1 to 19.

OUTPUT:

```
The product of all the odd
numbers from 1 to 19 is 654729075
```

SOLUTION:

```
Rem Display the product of all the odd numbers
Rem from 1 to 19
Rem OUTPUT: Text with the product of 1*3* … * 19
Cls
Rem Initialize
   product = 1 'accumulator for multiplication
Rem Loop to calculate product
    For I = 1 To 19 Step 2
      product = product * I
   Next I
Rem Output
   Print "The product of all the odd"
   Print "numbers from 1 to 19 is"; product
```

On Your Own

1. Use one **For .. Next** loop with the **Step** parameter to write a program that displays the numbers from 20 to 2 (in that order) on the screen. Also display the sum of the squares of those numbers.
2. Input a positive integer N. Compute and print the product of all positive even integers less than or equal to N. For example, if N = 13, the output is

$$2 \cdot 4 \cdot 6 \cdot 8 \cdot 10 \cdot 12 = 46080$$

3. Write a program that computes the sum of all the odd integers between N and 60. The value of N is supplied by the user.

7.3 Program Applications

The BASIC **For .. Next** loop with the **Step** parameter can be very helpful in many applications. We can find the interest earned by an investment over a period of time or the balance between economic supply and demand.

Example 7-3. The future value of an investment is the value of that investment after earned interest has been added at the end of each compounding period. Prepare a table that shows the future value of $100 after 1, 2, 3 and 4 years when interest is earned at each of the rates 3%, 3.25%, 3.5%, …, and 6% and paid at the end of the year.

OUTPUT:

Rate	Year 1	Year 2	Year 3	Year 4
3.00%	103.00	106.09	109.27	112.55
3.25%	103.25	106.61	110.07	113.65
3.50%	103.50	107.12	110.87	114.75
3.75%	103.75	107.64	111.68	115.87
4.00%	104.00	108.16	112.49	116.99
4.25%	104.25	108.68	113.30	118.11
4.50%	104.50	109.20	114.12	119.25
4.75%	104.75	109.73	114.94	120.40
5.00%	105.00	110.25	115.76	121.55
5.25%	105.25	110.78	116.59	122.71
5.50%	105.50	111.30	117.42	123.88
5.75%	105.75	111.83	118.26	125.06
6.00%	106.00	112.36	119.10	126.25

SOLUTION:

```
    Rem Calculate and display the future value of $100 at yearly intervals
    Rem with compounding at interest rates between 3% and 6%
    Rem with .0025% increments.
    Rem OUTPUT: Table of future values
       Cls
    Rem Display the table header
       Print "Rate Year 1 Year 2 Year 3 Year 4"
    Rem Calculate future values as interest rate
    Rem varies from 3% to 6% at 0.25% increments
A      For interest = .03 To .06 Step 0.0025
         year1 = 100 * (1 + interest)
         year2 = 100 * (1 + interest) ^ 2
         year3 = 100 * (1 + interest) ^ 3
         year4 = 100 * (1 + interest) ^ 4
B        Print Format(interest, "percent");
C        Print Format(year1, " 000.00");
D        Print Format(year2, " 000.00");
E        Print Format(year3, " 000.00");
F        Print Format(year4, " 000.00")
       Next interest
```

COMMENTS: In the program, the **For .. Next** loop beginning in **A** calculates the future values of $100 for four years. The figures are calculated for different interest rates each time the loop is executed. In **A**, the **Step** parameter with the **For .. Next** loop increments the interest rate by 0.0025 (0.25%) each time the loop is repeated. In **B**, the interest rate is displayed using the built-in **percent** formatting pattern. In lines **C** through **F** spaces are used in the formatting pattern so the displayed values line up with the headings. Note that each **Print** command ends in a semi-colon except the last one. This causes the interest rate and the four values for the different years to be printed on one line one after another. After the fourth value is printed, the printer advances to the beginning of the next line. Each iteration of the loop prints one row of the table.

Another application of **For .. Next** loops is in the economic theory of supply and demand. With the use of **For .. Next** loops and the **Step** parameter, we can write a BASIC programs to find the point of equilibrium between a supply curve and a demand curve. In economics the equilibrium point is the value at which the supply curve intersects the demand curve. Supply curves are always increasing functions while demand curves are always decreasing functions. With the price of a product less than the equilibrium price, the model implies that the demand is greater than the supply. A price higher than the equilibrium price indicates the supply exceeds the demand for the product.

Example 7-4. The supply and demand curves for gyzmos for the Ark Gyzmo Co. have been determined as the following functions:

$$S(p) = -1 + 0.10\ p^2$$
$$D(p) = 1 - 0.04\ p$$

Here *S(p)* is the supply curve which tells how many units are produced when the price per unit is *p*. *D(p)* is the demand curve which tells how many units are wanted by the consumer at the price *p*. In this example *S(p)* and *D(p)* are quantities of gyzmos and *p* is the price for one gyzmo. The graph in Figure 7-7 shows the supply and demand curves for gyzmos. *S(p)* gives the number of gyzmos that are being produced when the selling price is *p*. *D(p)* is the number of gyzmos that could be sold at price *p*. As supply increases, the excess demand will decrease until the supply and demand are in balance at the equilibrium point. Obviously, if the demand is higher than the supply, more suppliers will enter the market to meet the demand. The price of the additional units produced will be lower as unmet demand decreases.

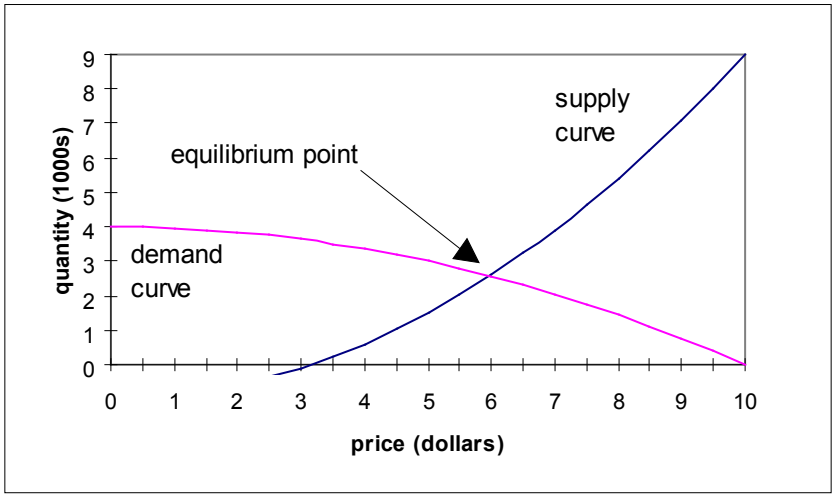


Figure 7-7: Supply and Demand Curves

Write a program that finds the first value for quantity and price (to the nearest penny) greater than $4.00 for which the number of gyzmos supplied is greater than the number of gyzmos demanded by the marketplace. Display in thousands how many gyzmos should be produced and what the price of each gyzmo should be in order to approximate equilibrium. We see from graphing the functions that the answer lies between 4 and 7 so we use those values in the **For** statement.

OUTPUT:

For equilibrium, 2,576 gyzmos should be
produced with a selling price of $5.98 each.

SOLUTION:

```
    Rem Find the equilibrium point for a given
    Rem pair of supply and demand curves.
    Rem OUTPUT: Number of gyzmos and the price to
    Rem the nearest penny of a gyzmo at equilibrium price
      Cls
    Rem Find supply and demand at increasing prices
A     For p = 4.0 To 7.0 Step 0.01
        supply = -1 + 0.1 * p ^ 2
        demand = 2 – 0.04 * p
B       If supply >= demand Then
          price = p
          GoTo L1
        End If
      Next p
    Rem Display output
    L1: Print "For equilibrium, ";
      Print Format(supply * 1000, "0,000");
      Print "gyzmos should be"
      Print "produced with a selling price of ";
      Print Format(price, "currency");
      Print "each."
```

COMMENTS: In the program the **For .. Next** loop beginning at **A** uses the price of a gyzmo as its control variable, *p*. It begins with a price of $4.00 and has a step size of one penny. For each price, the supply and demand are calculated using the equations given. The supply curve is always an increasing function and the demand curve is always a decreasing function. In the **If** block at **B**, the value of the supply function is compared to the value of the demand function. At the value of *p* for which the supply value is greater than or equal to the demand value, the value of the equilibrium point has been passed by at most 0.01 and the program transfers control out of the loop. If the supply value is less than the demand, equilibrium has not yet been reached. In this case, the loop repeats with an increased price. When the program finally does drop out of the loop, the current values of *supply* and *p* represent the number of gyzmos and the price of a gyzmo at the approximation of the equilibrium point. From the graph we drew in the analysis of the problem, we know that the equilibrium point will be found before *p* has a value greater than 7.0. Thus we will not terminate the loop by having *p* take on a value greater than 7.0 but by having the condition in **B** becoming true.

7.4 Generalized Functionality

A very important use of **For** .. **Next** loops is to make a program become a solution for a class of problems. For example, the simple program of summing a list of numbers could be programmed for 10 values as shown in Figure 7-8.

```
Rem Sum 10 numbers
Rem INPUT: Ten numbers one at a time
Rem OUTPUT: The sum of the 10 numbers entered

sum = 0
For I = 1 To 10
        num = Val(InputBox("Enter a number"))
        sum = sum + num

Next I
Rem Output sum
Print "The sum of 10 numbers = ";sum
```

Figure 7-8: Loop to Process 10 Values

The problem with this program is obvious when someone asks for a program that sums 12 numbers, or 15 numbers, or 23 numbers, …. There is no point in writing a program for a fixed number of numbers when the program can be written to handle any number of numbers. To write a more general program we have to think of the **For .. Next** idea extended to include the ability to work for any number of numbers. We do this in the program shown in Figure 7-9.

```
Rem Sum N numbers
Rem INPUT: The number of numbers N and each of the numbers
Rem OUTPUT: The sum of the N numbers entered

sum = 0 'initialize an accumulator
N = Val(InputBox("Enter the number of numbers to process:"))
For I = 1 To N
        num = Val(InputBox("Enter a number"))
        sum = sum + num

Next I
Rem Output sum
Print "The sum of ";N;" numbers = ";sum
```

Figure 7-9: Loop to Process Any Number of Values

The program now will work regardless of the number of numbers that need to be processed. Granted the user will have to count the number of data values to be processed to supply the correct value when the **InputBox()** command prompts for that value. This does seem to be a small price to pay for the greater flexibility. Programs are usually designed and written to handle any number of input values since a **For** loop does not care if the body of the code is repeated 10 or 20 or 200 times.

7.5 Nested Loops

The body of a loop can contain any sequence of commands including another **For .. Next** loop. The code in this case will look something like:

```
For I = num1 To num2

        …
        For J = num3 To num4

        Next J
        ….

Next I
```

Figure 7-10: Model of a Nested Loop

The loop with *I* as a control variable is called the **outer loop.** The loop with *J* as a control variable is called an **inner loop**. The only restriction on an inner loop is that the variable name used as a control variable cannot be the same as the variable name of the counter variable of the outer loop and the **Next** statement of the inner loop must occur before the **Next** statement of the loop containing it.

As an example it is known that the sum of N odd numbers starting with 1 gives a value of N^2. In Figure 7-11, we verify this for all values less than or equal to an input value *N* by adding up the odd numbers 1, 3, … , 2*I – 1 for each value of I in the range 1, 2, …, N.

```
     N = Val(InputBox("Enter an Upper Bound"))
     Rem Print header for columns of values
     Print " I","Sum of odds less than I"
A    For I = 1 To N
B        sum = 0
C        For J = 1 To I
D            sum = sum + 2 * (J – 1) + 1
E        Next J
F        Print I, sum
G    Next I
```

Figure 7-11: Computing N^2 as a Sum of Consecutive Odd Integers

Tracing the program for *N* = 3 will give an insight into how the nesting of the loops work.

Statement	Variables			Statement	Variables			Statement	Variables		
	I	**J**	**sum**		**I**	**J**	**sum**		**I**	**J**	**sum**
A	1			D	2	1	1	E	3	2	1
B	1		0	E	2	2	1	D	3	2	4
C	1	1	0	D	2	2	4	E	3	3	4
D	1	1	1	E	2	3	4	D	3	3	9
E	1	2	1	F	2	3	4	E	3	4	9
F	1	2	1	G	3	3	4	F	3	4	9
G	2	2	1	B	3	3	0	G	4	4	9
B	2	2	0	C	3	1	0				
C	2	1	0	D	3	1	1				

Figure 7-12: Tracing Program in Figure 7-11

7.6 Putting It All Together

1. Write a program to compute and display the sum of

$$\frac{1}{2}+\frac{2}{3}+\frac{3}{4}+\frac{4}{5}+\cdots+\frac{9}{100}$$

   Display the output with three decimal digits.

2. Input a pair of integers with the first representing the Kings' score for a game and the second representing the opposition's score. For each input pair, determine which team won. At the end, output the total number of wins and losses for the Kings. Suppose the Kings play 20 games in a season.
3. Ask the user to input the cost of tuition at a university this year. Suppose the tuition increases by 5.2% per year for the next fifteen years. Using a **For** .. **Next** loop, calculate the tuition costs for each of the next fifteen years. The output of your program may look something like this:

   The tuition in 2015 will be $00,000.00
   The tuition in 2016 will be $00,000.00
   ⋮
   The tuition in 2029 will be $00,000.00

4. A bank pays 5.6% interest on passbook savings. Suppose that on January 1, 2017, a savings account contained $2,500. Interest is credited to the account on the first day of each month on the balance of the account on the 15th of the previous month. On the 15th day of each month the customer deposits an additional $250. Calculate and display the balance of the account on the first day of each month for the next two years.
5. The Ark Gyzmo Co. produces the world's finest gyzmos. Direct production costs per gyzmo amount to $120. Secondary costs are $0.75N^2 per day where N is the number of gyzmos produced that day. Gyzmos are sold for $212 each. Write a program that calculates the profit for production levels that range from ten to thirty gyzmos per day.
6. The Board of the Huxville Symphony makes a list of all donors and the amount each has given at the end of the fiscal year. Suppose there are N donors. Output a list of the names and amounts given for each of the donors.

7. Compound interest is calculated using the formula

$$\text{interest} = \text{principal} * (1 + \text{rate}) \wedge \text{years}$$

   Calculate the interest value for $100, $200, and $300 at the end of each of the next ten years if the interest rate is 8%.
8. Write a program to find the average of any number of numbers. Each time the program is run, the user should input the number of numbers to be entered.
9. For each integer N in the range 1 to 10, compute the sum of the squares of all the integers less than or equal to N.
10. For each integer N in the range 10 to 15, compute the sum of all the odd integers less than or equal to N.

8 Random Numbers

Random numbers are used to implement simulation models and games of chance. For example, random numbers can be used to simulate the tossing of a coin or the rolling of a pair of dice. BASIC provides a built-in-function to generate random numbers that can be used in programs that randomly determine the next event of a simulation or the next play of a game.

Gaming machines pay off if the picture on each of the spinning wheels ends up the same in the display window at the end of the spin. Simulations of this game, often called a "one-armed bandit," uses random numbers and statistical distributions to determine what the final picture is on each wheel. The programmer then checks to see if the pictures are the same on all three wheels to know if there is a winner. The output in Figure 8-1 is the result of code that simulates a gaming device with three wheels.

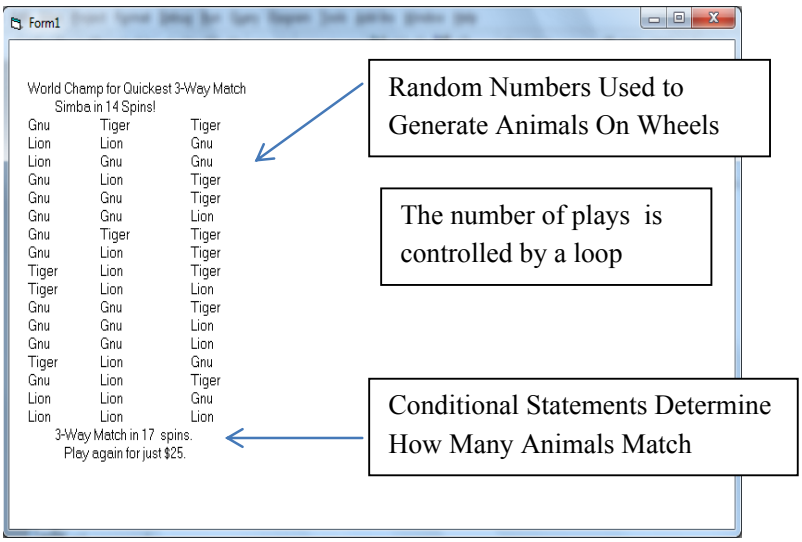


Figure 8-1: One-Armed Bandit

8.1 The Rnd Function

The BASIC function to generate a random number is **Rnd**. The **Rnd** function returns a randomly selected number between 0 and 1 (including 0 but not including 1). For instance, the following statement

randomNo = **Rnd**

assigns a number between 0 and 1 to the variable *randomNo*. Each time the **Rnd** function is executed in a program, a number between 0 and 1 is produced. In most cases when a random number is generated, it is assigned to a variable since every time **Rnd** is encountered in a program a different random number is generated. For example, if we have the code

randomNo = **Rnd**

and then the code

Print *randomNo*, **Rnd**

We will get two different values printed.

Example 8-1. Using the **Rnd** function, write a program that generates 10 random numbers between 0 and 1.

SOLUTION:

```
Rem Generate 10 random numbers between 0 and 1
Rem OUTPUT: 10 random numbers
  Cls
  For I = 1 To 10
    Print Rnd
  Next I
```

OUTPUT:

0.7055475

0.533424

0.5795186

0.2895625

0.301948

0.7747401

1.401764E-02

0.7607236

0.81449

0.7090379

COMMENTS: In the program, we use a **For .. Next** loop to call the **Rnd** function ten times. Each time the **Rnd** function is called, a different random number between 0 and 1 is produced. The format 1.401764E –02 is scientific notation to represent 0.01401764. Normally, computers represent numbers so that the first significant digit of a number is a non-zero digit. A number like 1.401764E-02 is the computer representation for 1.4014764 * 10^(-2) or 0.014014764.

SYNTAX

Rnd Function

Rnd returns a random number from [0, 1).

8.2 Using Randomize

Though the previous program produces ten different random numbers, the same ten numbers are produced in the same sequence each time we start and run the program. This program behavior would lead the user quickly to learn an optimal strategy for a program or game using random numbers to determine outcomes. To write a program to play a game or simulate a sequence of events, it is important that each time the program runs the sequence of events is different. Otherwise, the user will learn how to win the game by just remembering what the schedule of the events is. Using random numbers with the **Rnd** function begins generating a sequence of random numbers when BASIC supplies an initial value to the code of the **Rnd** function. If the same starting value is supplied every time the **Rnd** function starts generating a sequence of random numbers, the sequence of random numbers will always be the same. This description of how the **Rnd** function gets started tells us how we can generate different sequences of random numbers. We just need to supply the **Rnd** function with a different starting value each time a program begins to execute. In BASIC this is done by using the key word **Randomize** at the start of the program. When a program includes the **Randomize** command, a different starting value is used each time the program executes. In each instance that **Rnd** is used, the numbers generated will have the same properties but if the program includes **Randomize**, the sequence of numbers will be comprised of different numbers. The starting value for **Rnd** is called a **seed**.

When **Randomize** is executed, the program interprets the contents of some memory location as a number and sets the random number seed to that value. The function **Rnd** then uses the seed as a starting value. Different seeds will give different starting values and different sequences of random numbers. The entire process of finding an initial value for **Rnd** is called seeding the random number generator.

Figure 8-2 shows the output from two more executions of a program like the one in Example 8-1 but including **Randomize** before **Cls**. Notice that the executions produce different sequences of numbers. Both sequences of numbers will have the same properties but each is produced using a different seed.

Sample output #1	Sample output #2
0.2620508	0.3679773
0.4732891	0.1242596
0.9317983	0.0726369
0.7170985	1.333261E-02
0.566215	0.5212626
0.3413142	0.9449214
0.2361704	1.909882E-02
0.3286405	0.1986051
0.4076144	0.1414706
3.903973E-02	0.8062211

Figure 8-2: Random Numbers Generated Using Randomize

On Your Own

1. Write a program to display and add five random numbers between 0 and 1. Display the sum of the five numbers. Run the program three times. Use the **Randomize** command so that each time you run the program, you should get a different result.
2. What is the range of numbers generated by 17*Rnd, 21*Rnd, and 52*Rnd?
3. Write a program to show that there are different values generated each time **Rnd** is called in a program. The program should have just two lines of code:

```
X = Rnd
Print X, Rnd
```

SYNTAX

Randomize

Randomize generates a seed for **Rnd**. Each time the program is executed a different seed is found.

8.3 Coin Tossing

Random numbers can be used to simulate the tossing of a coin. When a coin is tossed, there are two possible results: heads or tails. Each result has a 50% chance of occurring, providing the coin is fair, i.e. each outcome is equally likely to occur. To simulate a coin toss, we want to use the **Rnd** function. Since the **Rnd** function produces numbers between 0 and 1, we must decide which numbers generated by **Rnd** will represent flipping a head and which numbers will represent flipping a tail. Since the two events are equally likely and all numbers between 0 and 1 are equally likely to result from the **Rnd** function, we need to have half the numbers possible to represent flipping a head and the other half of the numbers possible to represent flipping a tail. Thus, since the interval [0, 0.5) is half of [0, 1), we can assign the values of **Rnd** less than 0.5 to mean flipping a head and the numbers greater than or equal to 0.5 to mean flipping a tail. (See Figure 8-3.)

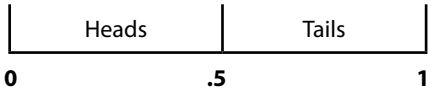


Figure 8-3: Number Representation of Heads and Tails

Example 8-2. Write a program that simulates the tossing of a fair coin ten times. The outcome should indicate the result of each toss.

SOLUTION:

```
    Rem Simulate the tossing of a coin and
    Rem display the result of each toss.
    Rem OUTPUT: Text with the results of the coin toss
A     Randomize
      Cls
B     For I = 1 To 10
C       flip = Rnd
D       Print "Flip "& I & ": "; 'concatenate for style
        If flip < 0.5 Then
          Print "The coin came up heads."
        Else
          Print "The coin came up tails."
        End If
      Next I
```

OUTPUT:

```
Flip 1: The coin came up tails.
Flip 2: The coin came up tails.
Flip 3: The coin came up heads.
Flip 4: The coin came up tails.
Flip 5: The coin came up heads.
Flip 6: The coin came up heads.
Flip 7: The coin came up tails.
Flip 8: The coin came up heads.
Flip 9: The coin came up tails.
Flip 10: The coin came up heads.
```

COMMENTS: Because the program includes the **Randomize** (**A**) command, each time the program is run a different set of 10 random numbers will be generated. The sequence of outputs shown may never show up again! The **Rnd** function in **C** of the program generates a number between 0 and 1 (excluding 1) and assigns that number to *flip*. The **For .. Next** loop beginning in **A** repeats the loop ten times for ten tosses of the coin. The string concatenation in **D** removes the following space when *I* is printed as a number. Try this print statement with a semicolon replacing "&" to see the difference.

On Your Own

1. Write a program that simulates 100 tosses of a coin. Record and display the number of heads and the number of tails that resulted. Display the final count. Run the program 10 times and compare the different outputs.

8.4 Tossing a Biased Coin

Suppose that we toss a biased coin that comes up heads 35% of the time and tails 65% of the time. To simulate a toss of this coin, we must assign 35% of the values that could be produced by **Rnd** to represent heads and 65% of the numbers that could be produced by **Rnd** to represent tails. Therefore, let us say that any random number generated that is less than 0.35 will represent flipping a head, and the random numbers greater than or equal to 0.35 will represent the event of flipping a tail. (See Figure 8-4.)

Figure 8-4: Number Representation of a Biased Coin

There are generally two steps in writing a program to simulate events. The first step is to build a model of the possibilities by dividing the interval [0, 1) into appropriate sized pieces to represent the likelihood of each event. The second step is to write the program that uses the model built.

Example 8-3. Write a program that simulates ten tosses of a biased coin where 35% of the time a flip is expected to produce a head and 65% of the time a flip is expected to produce a tail. Output the result of each flip.

SOLUTION:

```
A   Rem Simulate ten tosses of a biased (.47-.53)
    Rem coin and displays the results.
    Rem OUTPUT: Text with the results of the toss
      Randomize
      Cls
      For I = 1 To 10
        flip = Rnd
        Print "Flip " & I & ": ";
        If flip < 0.35 Then
          Print "The coin came up heads."
        Else
          Print "The coin came up tails."
        End If
      Next I
```

OUTPUT:

Flip 1: The coin came up tails.
Flip 2: The coin came up tails.
Flip 3: The coin came up tails.
Flip 4: The coin came up heads.
Flip 5: The coin came up heads.
Flip 6: The coin came up tails.
Flip 7: The coin came up heads.
Flip 8: The coin came up tails.
Flip 9: The coin came up tails.
Flip 10: The coin came up tails.

COMMENTS: The **Rnd** function at **A** generates a number between 0 and 1 (excluding 1) and assigns that number to *flip*. If the number produced is less than 0.35, the coin is said to come up heads. If the number produced is greater than or equal to 0.35, the coin is said to come up tails. We can use this kind of reasoning with the **Rnd** function to simulate any situation in which we know the probability of the occurrence of each possible event (in this case, two events-heads and tails). Observe that the probability of all the events should add up to 1.

On Your Own

1. Liam has a batting average of .292. Write a program that simulates Liam's at bats in baseball games during which he bats 100 times. Display the total number of hits made by Liam. A batting average of .292 means that 29.2% of all the times at bat the batter got a hit. (Sacrifice flies, bunts and walks are not counted in the total number of at bats.)

8.5 Die Rolling

In the last section, we represented the occurrences of events with numbers ranging between 0 and 1 that are generated by **Rnd**. We can use this method to simulate the roll of a die. A regular die has six sides to produce a total of six possible equally likely outcomes: 1, 2, 3, 4, 5, 6. Each outcome has a 1/6 chance of occurring. For this situation, we must break up the numbers between 0 and 1 into six equally sized groups. Thus, we will place divides at 1/6, 2/6, 3/6, 4/6, 5/6 and 1 in the model of this process. We then assign to each of these intervals the event that each value in the interval is taken to represent. For example, all the random numbers between 0 and 1/6 could represent rolling a die and having 1 on the top face when the die comes to rest. See Figure 8-5 for one possible model for rolling a die. (There are 6 * 5 * 4 * 3 * 2 * 1 possible ways to assign outcomes to the six equal sized subintervals.)

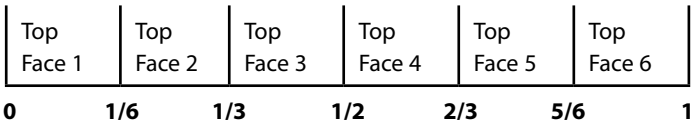


Figure 8-5: Number Representation of the Results of a Roll of a Die

When a random number is in [0, 1/6) we will say a 1 has been rolled. When a random number is in [1/6, 2/6) we will say that a 2 has been rolled. This pattern continues up to the resulting roll of six when the random number is in [5/6, 1).

The assignment of events to intervals can be done in many different ways when all the events are equally likely because any interval can represent any one of the events. For example, Figure 8-6 gives another model for rolling a die.

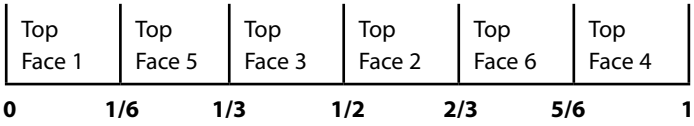
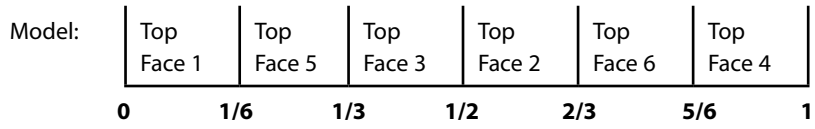


Figure 8-6: Number Representation of the Results of a Roll of a Die

When different events have different likelihood of occurring, the interval that represents a particular event can be placed anywhere in the interval [0, 1) or even be a collection of disjoint intervals as long as the length of the interval or intervals assigned to a particular event is equal to the probability of the event happening. Any event must be represented by subintervals of [0, 1) that are disjoint from the subinterval(s) representing any other event.

Example 8-4. Write a program that simulates ten rolls of a die. The program should print the result of each roll.



SOLUTION:

	Rem Simulate ten rolls of a die and display the results
	Rem OUTPUT: Text with the results of the rolls
	Randomize
	Cls
A	For I = 1 To 10
	Rem Get a random number
B	x = Rnd
	Rem See what value represents the random number
C	If x < 1 / 6 Then
	roll = 1
	ElseIf x < 2 / 6 Then
	roll = 2
	ElseIf x < 3 / 6 Then
	roll = 3
	ElseIf x < 4 / 6 Then
	roll = 4
	ElseIf x < 5 / 6 Then
	roll = 5
	Else
	roll = 6
	End If
	Rem Print the value on the top face of the die
D	Print "Roll " & I & ": The top face ended up as " & roll & "."
E	Next I

SAMPLE OUTPUT:

Roll 1: The top face ended up as 3.
Roll 2: The top face ended up as 5.
Roll 3: The top face ended up as 5.
Roll 4: The top face ended up as 4.
Roll 5: The top face ended up as 4.
Roll 6: The top face ended up as 2.
Roll 7: The top face ended up as 2.
Roll 8: The top face ended up as 5.
Roll 9: The top face ended up as 5.
Roll 10: The top face ended up as 5.

COMMENTS: The **For .. Next** loop beginning in **A** and ending with **E** causes its body to be repeated ten times to simulate ten rolls of the die. In **B** the **Rnd** function generates a random number between 0 and 1. Beginning in **C**, the **If** block associates the random number with an event. Each case of the **If** block identifies one of the six possible results of rolling a die. Once the result of the roll is determined, the program transfers control to **D** where the result is printed. The condition could also be coded as six **If** .. **Then .. End If** statements, the first two of the form:

```
If (x < 1/6) Then
        roll = 1
End If
If (x >= 1/6) And (x < 1/3) Then
        roll = 2
End If
```

Remember the mathematical notation 1/6 <= x <= 1/3 is not valid in BASIC. Also, in an **If .. ElseIf** statement as soon as a condition is found to be true some action is taken and control passes to the first statement following **EndIf**. In the **Print** statement the string concatenate operation makes the output look better-try a semicolon to see the difference.

8.6 Scaling the Rnd Function

The program in Example 8-4 correctly simulates the roll of a die but it is a bit difficult to understand. The program would be more convenient and easier to follow if we could produce random numbers in ranges other than 0 and 1. There are several ways in which we can scale the **Rnd** function so that it produces random numbers with different ranges. For example, if we multiply the **Rnd** function by a number *N*, the range of numbers produced will be stretched to represent the interval [0, N). Specifically, the following statement

stretchedNum = 6 * **Rnd**

assigns to *stretchedNum* a random number between 0 and 6 (since multiplying a number in [0, 1) by 6 will result in a number in [0, 6)). The previous command would produce numbers such as those shown in Table 8-1 using the random numbers shown.

Rnd	6 * Rnd
0.3482477	2.089486
0.7303234	4.381941
0.0138204	0.082922
0.3116572	1.869943
0.8606638	5.163983

Table 8-1: Random Numbers Scaled by 6

If we prefer to deal with integer numbers rather than with decimal numbers, we can use the **Int()** function to remove the digits to the right of the decimal point. The **Int()** function truncates a decimal number or "throws away" the digits to the right of the decimal point. See Table 8-2 for examples of the **Int()** function applied to the numbers in the second column of Table 8-1.

Operation	Result
Int(2.089486)	2
Int(4.381941)	4
Int(0.082922)	0
Int(1.869943)	1
Int(5.163983)	5

Table 8-2: Example Results of the Int() Function

Thus, if we apply the **Int()** function to the values of **Rnd** that are scaled by some factor N, we can produce random integers within the range 0, 1, …, N-1. For example, the following statement

$$\text{die} = \textbf{Int}(6 * \textbf{Rnd})$$

assigns to *die* one of the following integer values: 0, 1, 2, 3, 4 or 5. Now say that we want to generate integers that simulate the roll of a die. We want to generate random integers between 1 and 6 inclusive and not 0 to 5. To do this, we can simply add one to the range of numbers we just produced. For example, the following statement

$$\text{die} = \textbf{Int}(6 * \textbf{Rnd}) + 1$$

assigns to *die* one of the following integer values: 1, 2, 3, 4, 5 or 6. See Table 8-3 for examples of this addition operation applied to the numbers in the second column of Table 8-2.

Operation	Result
Int(2.089486) + 1	3
Int(4.381941) + 1	5
Int(0.082922) + 1	1
Int(1.869943) + 1	2
Int(5.163983) + 1	6

Table 8-3: Example Results of Int(6 * Rnd) + 1

Thus, the above statement simulates the roll of a die and assigns the value of the roll to *die*. This scaling technique works when all the events are equally likely. Notice that we do not need a sequence of if statements to determine the interval that contains the random number so the program is much simpler as shown in Example 8-5.

Example 8-5. Use the scaling process on the output of the **Rnd** function to write a program that simulates ten successive rolls of a die and displays the results.

SOLUTION:

```
       Rem Simulate ten rolls of a die and
       Rem display the results.
       Rem OUTPUT: The results of repeated rolls of a die
           Randomize
           Cls
           Print "Output of 10 rolls of a die."
           For I = 1 To 10
               roll = Int(6 * Rnd) + 1
A              Print roll;
           Next I
B          Print
```

COMMENT: The print statement in **B** causes the printer to advance to the beginning of the next line as all the random numbers generated and printed in **A** appear on a single line. (Can you tell why?)

The game of craps involves computing the sum of the number of spots on the top faces of two die. This is actually one of the "fairest" casino games. You win if you subsequently roll two die that have the same sum for the values on their top faces. You lose when the sum of the top faces is seven except rolling seven with your first roll is an automatic win.

Example 8-6. Modify the program from Example 8-5 to roll a pair of dice ten times and calculate the sum of the spots on the top faces of the two die when they come to rest.

OUTPUT: 6 9 5 9 7 12 6 11 8 7

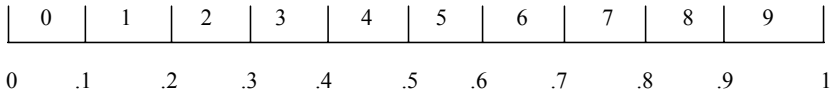
SOLUTION:

```
    Rem Simulate ten rolls of a pair of dice and displays
    Rem the results (sum of spots on top faces).
    Rem OUTPUT: Sum of the spots on the top faces of two die
        Randomize
        Cls
        Print "The sum of the top faces of two die rolled 10 times."
        For I = 1 To 10
A           die1 = Int(6 * Rnd) + 1
B           die2 = Int(6 * Rnd) + 1
C           roll = die1 + die2
            Print roll;
        Next I
        Print
```

COMMENTS: In the program there is no need for a large **If** block to associate the random number with a result of a roll. The statements in **A** and **B** generate integers in the range 1 to 6. Thus, each statement directly simulates the result of rolling one die. To find the results of the roll of the pair of dice, the two separate results are added in **C**.

Example 8-7. Write a program that simulates a lottery drawing in which four ping pong balls are selected from four separate bowls each containing 10 ping pong balls. Each ball is marked with an integer from 0 to 9. (This program is similar to the die rolling program except that the random numbers generated are in a different range.)

MODEL: The value in the interval represents the event represented by every number in the interval.



SAMPLE OUTPUT:

The winning lottery number for Saturday is:
6 0 7 3

SOLUTION:

```
    Rem Simulate a lottery drawing in which four random
    Rem numbers R1, R2, R3, and R4 from 0 to 9 are used to generate
    Rem the four digit number R1 R2 R3 R4.
    Rem OUTPUT: Text with the winning number
        Randomize
        Cls
        Print "The winning lottery number for Saturday is:";
        Print
A       For I = 1 To 4
B           num = Int(10 * Rnd)
C           Print Format(num, "@@");
        Next I
D       Print
```

COMMENTS: The **For .. Next** loop beginning in **A** repeats four times to produce the four digits needed to determine a winner of the lottery. The expression in **B**, **Int**(10 * **Rnd**), first generates a random integer in [0, 1) and then scales the value to be in [0, 10). The **Int** function produces one of the integers 0, 1, 2, … , 8, and 9. The semicolons in the print statement (**C**) cause the four digits to be printed on the same line close together with the first number the coefficient of 1000, the second number drawn is interpreted as the coefficient of 100, the third number drawn is interpreted as the coefficient of 10, and the final digit drawn is interpreted as the ones digit of the winning number. The print statement (**D**) causes the printer to advance to the next line. The **Format()** command is used to give a consistent spacing to the answer.

On Your Own

1. Write a program that simulates rolling a *twelve*-sided die (with the numbers 1 through 12 on the sides). Each side is equally likely to occur. Roll the die 15 times and display the results of the rolls.

2) Write a program that simulates dealing three five card hands from the standard deck of 52 cards (don't worry about how the cards are named). Run the program 50 times and see if the first card is ever dealt again.
3) Write a program that totals the amounts N customers spend if the amount spent by each customer is a randomly generated number between 0 and $12.00. Use 12 ***Rnd** to compute how much a customer spends.

8.7 A Simulation

If we know the probability of the occurrence of all possible events in some game or scenario, we can often use random numbers to simulate these games. For instance, we saw how to simulate flipping a coin. We can use similar methods to simulate situations in which more than two events may occur.

Example 8-8. Write a program that simulates the sales of 20 cars sold by a used car dealership in a typical month. Each time a car is sold, there is a 50% chance that the car will be a Ford, there is a 35% chance that the car will be a Toyota, and a 15% chance that the car will be a BMW. (See Table 8-4.) These percentages were determined by examining all sales records of a large dealership for a two year period. The program should print the name of the car make for each car sold and how many of each car make were sold in a typical month.

Car Make	Percentage of Sales
Ford	50%
Toyota	35%
BMW	15%

Table 8-4: Car Makes and Percentage of Sales

To simulate the sales by this dealership, we must first construct a model showing how the numbers in [0, 1) will represent the sale of different kinds of cars. We divide the numbers between 0 and 1 into three intervals so that each interval represents the sale of a car of a particular make. Since the probability of each car being sold is different, the sizes of the intervals reflect this. We refer to each car's percentage of sales in order to make the proper divisions of the numbers between 0 and 1. See Figure 8-7 for a model of this simulation.

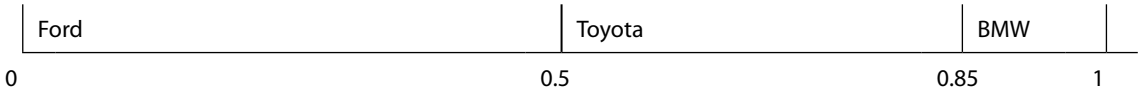


Figure 8-7: Number Representation of the Percentage Sales of Each Car Make

When a random number is in [0, 0.5), we say that a Ford has been sold. When a random number is in [0.5, 0.85), we say that a Toyota has been sold. When a random number is in [0.85, 1), we say that a BMW has been sold.

OUTPUT:

Out of 20 sales, the following
number of each make was sold.

Ford	11
Toyota	7
BMW	2

SOLUTION:

```
    Rem Simulate twenty car sales and display how many of each
    Rem car make were sold using a model based on previous sales:
    Rem 50% Ford, 35% Toyota, 15% BMW.
    Rem OUTPUT: How many cars of each model were sold
        Randomize
        Cls
    Rem Initialize sales of each make
A       ford = 0
        toyota = 0
        bmw = 0
    Rem Simulate 20 sales
B       For I = 1 To 20
C           x = Rnd
D           If x < 0.5 Then
              ford = ford + 1
            ElseIf x < 0.85 Then
              toyota = toyota + 1
            Else
              bmw = bmw + 1
            End If
E       Next I
    Rem Display output
F       Print "Out of 20 sales, the following"
        Print "number of each make was sold."
        Print
        Print "Ford", ford
        Print "Toyota", toyota
        Print "BMW", bmw
```

COMMENTS: Beginning in **A** the number of sales of each car make is initialized to zero. In **C**, the **Rnd** function generates a number between 0 and 1. Beginning in **D**, the **If** block associates the random number with a corresponding car make. When the interval for a random number is identified and the program increments the sales count for that make, the program continues at **E**. At **E** the increment, testing, and transfer are carried out until the body of the loop has been executed 20 times. Each case of the **If** block increments the number of sales of one of the car brands. The **For .. Next** loop (**B** through **E**) repeats twenty times to simulate twenty car sales. Finally, beginning in **F** outside the **For .. Next** loop, the results of the sales are displayed.

On Your Own

1. A computer store sells approximately 50 computers per week. A store clerk has estimated that each time a computer is sold, there is a 55% chance that the computer is an IBM, a 35% chance that the computer is a Macintosh and a 10% chance that the computer is a Sun. Write a program that simulates one week of sales at the store. Your program should display how many of each kind of computer was sold.
2. Redo Example 8-8 by first scaling [0, 1) to 1, 2, … , 20 and using the values 1, 2, … , 10 to represent selling a Ford; 11, 12, … , 17 to represent selling a Toyota; and 18, 19, 20 to represent selling a BMW.
3. Redo Example 8-8 by first scaling [0, 1) to 1, 2, … , 100 and using the values 1, 2, … , 50 to represent selling a Ford; 51, 52, … , 85 to represent selling a Toyota; and 86, 87, … , 100 to represent selling a BMW.

8.8 Putting It All Together

1. Smith has a .189 batting average, Jones has a .273 batting average, and Lewis has a .324 batting average. Write a program that simulates 27 at bats in which each player bats nine times in the order Smith, Jones, and Lewis. Observe that there are three models to use. Display the total number of hits by each player.
2. Write a program that simulates a dice rolling game. Roll a pair of dice ten times. If a 7 or an 11 is rolled, record a win. Your program should display the results of each roll and the total number of wins.
3. Suppose the color of a car is determined by the value of a random number. If a random number has a value of less than 0.5 color the car red. If the random number is between 0.5 and 0.75 color the car green. Otherwise, color the car blue. If the production line produces 25 cars a day, write a program that simulates choosing the colors of the cars produced on a typical day. Output the number of cars of each color.
4. Imagine a slot machine that has three equally likely outcomes: orange, lemon and bell. Write a program that simulates 10 plays.
   (a) Output the number of times the outcome was a bell.
   (b) Modify (a) so that each play of the game generates two outputs: orange-bell, bell-bell, lemon-bell, etc. Output the two symbols and indicate whether the two symbols match.
   (c) Modify (b) so that each play of the game generates three outputs. Output the three symbols and indicate whether or not all three symbols match.
   (d) Modify (c) so that the program adds 5 to a bankroll when the output is three oranges, 10 when the output is three lemons, and 25 when the output is three bells. Subtract 1 from the accumulator whenever the three symbols do not match. Start with a bankroll of 25. Output the value of the bankroll after 100 plays of the game. (Do not output the symbols.)

5. If a player's at bat in a baseball game takes three minutes 28% of the time, five minutes 40% of the time, and eight minutes the other times, simulate a game with 54 at bats and determine how long a game may take. Repeat the experiment 50 times and find the average game time for the 50 simulated games. Hint: All the code for the play of the game can be part of the true range of a condition that keeps track of whether 50 games have been completed. The true range of the condition should also sum the time for each experiment so the average can be computed when the loop finishes.
6. Simulate a slot machine with four outcomes. Two outcomes occur 25% of the time each, one occurs 35% of the time, and the fourth occurs 15% of the time. The outcomes that occur 25% of the time have a payoff of $3. The outcome that occurs 35% of the time has a payoff of $1. The option that occurs 15% of the time has a payoff of $5. If each play costs $2, compute the winnings for 20 plays of the slot machine. Add an option that will find the average winnings for any number of 20 plays.
7. For twenty customers simulate their shopping experiences. Suppose there are four different stores and the sale for each customer at a store is less than $20.00. Use two random numbers, one to simulate which store the customer visits assuming each store is equally likely to be chosen. The second random number indicates how much the customer spent at the store they visited. Compute how many customers went to each store and the total sales for each store.
8. Repeat problem (7) but now assume the purchase by a customer was in the range:

$5.00 < purchase amount < $25.00.

9 Graphics

Graphs and charts can show relationships that exist between and among data elements. Animations can be used to create films or commercials by putting together a sequence of graphical images. All these graphical objects are applications of computer graphics. This chapter gives an insight into how computer graphics are created using BASIC programming.

Spreadsheets make graphical display of data very easy. Underlying the ease spreadsheets provide for the user is the kind of programming that is introduced in this chapter. In Figure 9-1 a pie chart gives a visual representation of three categories of expenses and shows how the relative size of each class of expenses took up part of all expenses.

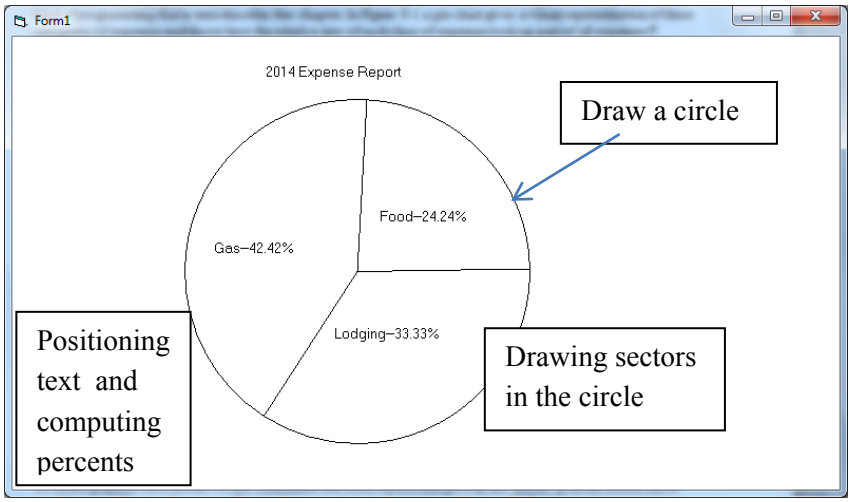


Figure 9-1: Pie Chart

9.1 Resolution and Color

Personal computers and laptops have color monitors and full color graphics display capabilities. The output form defined in the template code creates a graphics display area divided into 425 rows of 720 columns each. This form is organized as 306,000 **pixels**, small points on the screen that can be individually accessed and colored.

To create graphics on a screen, we use commands that refer to particular pixels on the screen. Each pixel has an address on the screen. The pixels are numbered using two coordinates, the first increasing as you move from the left margin of the form to the right (column numbers). The second coordinate increases as you move from the top of the form towards the bottom of the form (row numbers) on the screen. A pixel's address consists of its column number and its row number. Figure 9-2 shows the addresses of the pixels at the corners of the display area.

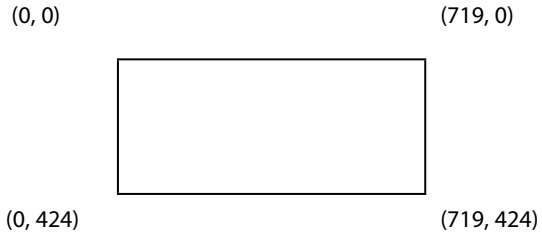


Figure 9-2: Addresses of Corner Pixels

Graphic images are produced by changing the colors of pixels. Each pixel can be separately assigned a color. BASIC has a number of predefined color constants that may be used that are listed in Table 9-1.

Color Constant	Color
vbBlack	Black
vbBlue	Blue
vbGreen	Green
vbRed	Red
vbCyan	Cyan
vbMagenta	Magenta
vbYellow	Yellow
vbWhite	White

Table 9-1: BASIC Color Constants

A color is composed of three components: red, green, and blue. The **RGB()** function built into BASIC may be used to specify a color in BASIC. The **RGB()** function takes three arguments. Each argument is a value between 0 and 255 indicating the strength of the component's color. The first argument specifies the red component (R), the second specifies the green component (G), and the third specifies the blue component (B). See Table 9-2 for a list of standard colors and their **RGB()** specifications. Note, for example, that yellow is produced by setting the red and green components to the maximum value and setting the blue component to the minimum value.

Color	RGB specification
Black	**RGB**(0, 0, 0)
Blue	**RGB**(0, 0, 255)
Green	**RGB**(0, 255, 0)
Red	**RGB**(255, 0, 0)
Cyan	**RGB**(0, 255, 255)
Magenta	**RGB**(255, 0, 255)
Yellow	**RGB**(255, 255, 0)
White	**RGB**(255, 255, 255)

Table 9-2: Standard Colors and their RGB Specifications

The **RGB()** function may be used whenever a color can be specified. For example, to change the default background color of the display, the built-in **BackColor** variable is assigned a different color. To change the background color to blue, use the command:

BackColor = **RGB**(0, 0, 255)

There is a similar variable **ForeColor** that can be assigned a vb-color or an RGB setting to indicate the color used to draw on the background color.

9.2 Coloring Pixels

Graphics in BASIC can be thought of as drawing pictures on the screen. Drawing pictures is the result of coloring pixels one at a time until the final picture appears because some set of pixels have been assigned a different color from the background color. The screen starts out with every pixel colored in the **BackColor**.

PSet() is the BASIC command to color pixels The **PSet()** command takes two parameters: a pixel address and a color. The first parameter, (*a, b*), is the location of the pixel to be colored. Remember (*a,b*) uses *a* to measure the distance from the left side of the screen and *b* to measure the distance from the top of the screen. The second parameter, **color**, is the color that the pixel will be colored. However, if the **color** parameter is omitted from **PSet()**, the pixel will be colored with the current value of **ForeColor**. BASIC starts with default values for both **BackColor** and **ForeColor**. Table 9-3 shows examples of valid **PSet()** commands.

PSet command	Result
PSet (90, 210)	Color pixel (90, 210) the foreground color
PSet (9, 11), vbYellow	Color pixel (9, 11) yellow
PSet (409, 102), RGB(98, 22, 144)	Color pixel (409, 102) purple

Table 9-3: Example PSet Commands

SYNTAX

Pset(*,*),color

PSet(over, down) changes the color of the pixel at (over, down) from the background color to the value of color. If no explicit value for color is used, the pixel is colored with **ForeColor**.

On Your Own

1. Explain why the following **PSet** statements are invalid.

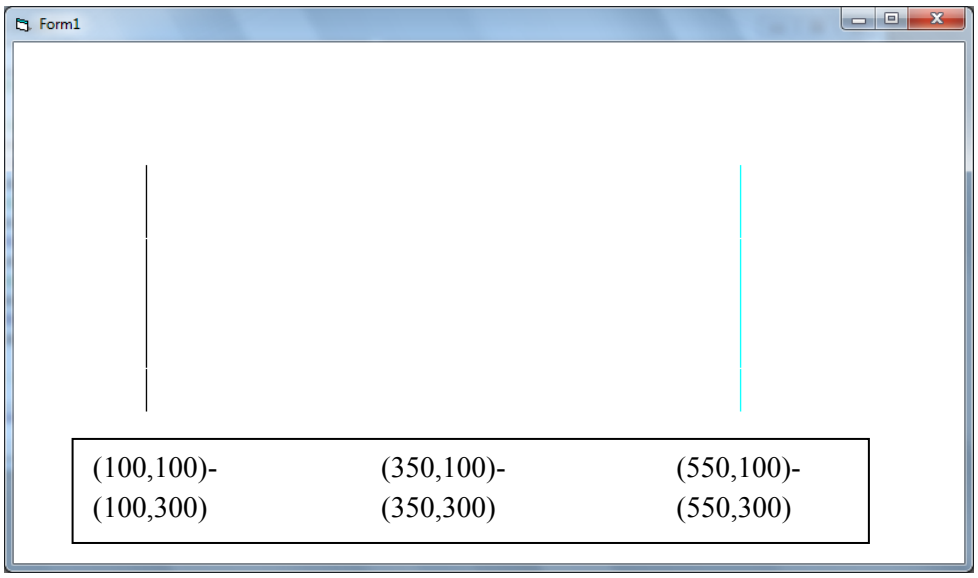
   PSet 89, 10

   PSet (90, 12) **vbRed**

   PSet (54, 39); **RGB**(33, 202, 188)

Example 9-1. Using the **PSet()** command, write a program that colors the pixels (100, 100), (100, 101), (100,102), … , (100, 300) in the foreground color. The program should then color the pixels (350, 100), (350, 101), (350, 102), … , (350, 300) with the background color. Finally, the pixels (550, 100), (550, 101), (550, 102), … , (550, 300) should be colored with cyan. Notice that the second line is "invisible."

OUTPUT:



The pixels that are arguments for **PSet()** change color. The machine is usually so fast that you cannot actually see the lines drawn pixel by pixel with the different colors.

SOLUTION:

```
      Rem Colors a series of pixels on the screen
      Rem in the foreground color. Then
      Rem colors a series of pixels in the
      Rem background color. Finally, it colors a series
      Rem of pixels with vbCyan.
      Rem OUTPUT: Points on the screen in three colors
          Cls
      Rem Draw the points in the foreground color
A         For down = 100 To 300
B             PSet (100, down)
C         Next down
      Rem Draw the points in the background color
D         For down = 100 To 300
E             PSet (350, down), BackColor
F         Next down
      Rem Draw the points in cyan
G         For down = 100 To 300
H             PSet (550, down), vbCyan
I         Next down
```

COMMENTS: The commands at **A, B,** and **C** draw the points (100, 100) to (100, 300) in the foreground color. The **For .. Next** counter variable, *down*, represents the numbers 100 to 300 and is used in the pixel address in **B**. The **For .. Next** loop beginning in **D** is the same as that beginning in **A**, except that the points are drawn in the background color at a location 350 pixels from the left edge of the window. Finally, the **For .. Next** loop beginning in **G** once again draws a line, but this time the pixels are drawn in cyan.

To cause the program to pause, we can use the **InputBox()** function to ask the user to press the **Enter** key:

pause = **InputBox**("Press Enter to continue.")

The effect of this command is to freeze the screen as it is until the user presses the **Enter** key. When the user presses the **Enter** key, the program will continue. If you include this command after **C** and **F**, you will see the lines drawn one at a time.

Example 9-2. Modify the previous example program to pause each time a line is completed.

SOLUTION:

```
    Rem Colors a series of pixels on the screen in the foreground color.
    Rem Then colors a series of pixels in the background color.
    Rem Finally, it colors a series of pixels cyan.
    Rem OUTPUT: Points on the screen in three colors
        Cls
    Rem Draw the points in the foreground color (black)
        For down = 100 To 300
            PSet (100, down)
        Next down
    Rem Pause for user
A       pause = InputBox("Press Enter to continue.")
    Rem Draw the points in the background color (white)
        For down = 100 To 300
            PSet (350, down), BackColor
        Next down
    Rem Pause for user
B       pause = InputBox("Press Enter to continue.")
    Rem Draw the points in cyan
        For down = 100 To 300
            PSet (550, down), vbCyan
        Next down
```

COMMENTS: In the solution program, the commands in **A** and **B** cause the program to pause so that the user can see the results of the **For .. Next** code above it. An execution of this program would clearly show the points along the line first in black and the third line in cyan. The second line was drawn in the background color so you do not "see" it.

A **For .. Next** command is a natural way to go about changing the color of the pixels in these two programs. The resulting figures were just vertical lines but the idea of how to draw any line should be intuitively clear. We must change the color of all the pixels on the line segment from one end point of the line to the other endpoint of the line segment. The problem is to figure out the addresses of just those pixels that lie on the line to be drawn.

On Your Own

1. Write a program that sets the screen background color to green. Draw the line segment with endpoints (0, 75) and (200, 75). The line segment should first appear in the color white and then, after a pause, change to the color blue.
2. Using the program in Example 9-2 as a model incorporate the **STEP** option with an increment of 3 to see how dashed lines can be drawn.

9.3 Drawing Lines

With the ability to draw lines comes the ability to create boxes, charts, graphs, parallelograms, and polygons of any size. The ideas we explore learning to draw lines with BASIC shows us how circles and any figure that can be described by a function can be drawn.

Drawing a line starts with the identification of the two endpoints of the line segment. Suppose we want to draw the line from the point (100, 75) on the output form to the point (250, 315). These two points are shown in Figure 9-3.

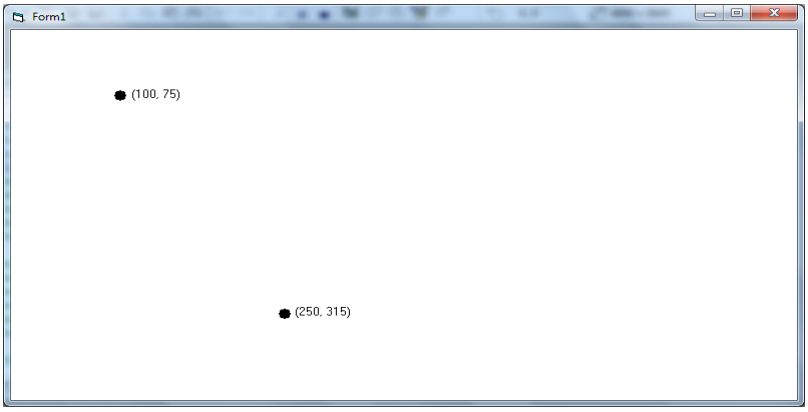


Figure 9-3: Two Points on the Output Form

In Figure 9-4 we see a dashed line joining (100, 75) to (250, 315). What we need to determine is the location of each pixel along the dashed line so we can color them differently from the background color.

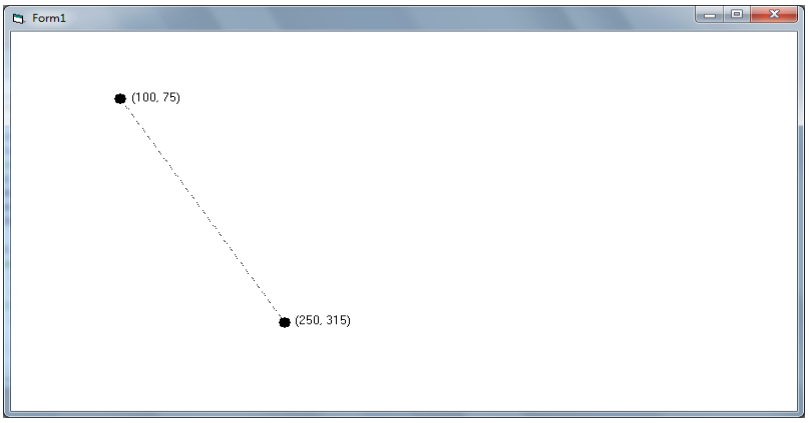


Figure 9-4: Dashed Line joining the Endpoints

The obvious answer is to find the equation of the line joining (100, 75) to (250, 315).

The fundamental property about lines is that the slope can be computed using any two points on the line and in fact no matter which two points are used, the value of the slope is always the same. The slope of a line joining two points (over1, down1) and (over2, down2) is simply

$$\frac{\text{down2 - down1}}{\text{over2 - over 1}}$$

If we let (over1, down1) = (100, 75) and (over2, down2) = (250, 315), the slope of the line joining them is

$$\frac{315 - 75}{250 - 100}$$

The next step in finding the equation of this line is to compute the slope again using one of the given points and an arbitrary point on the line (see Figure 9-5). Let us choose the point (over, down) as an arbitrary point on the line. Now when we compute the slope for the two points (over1, down1) = (250, 315) and (over2, down2) = (over, down) we get:

$$\frac{\text{down - 315}}{\text{over - 250}}$$

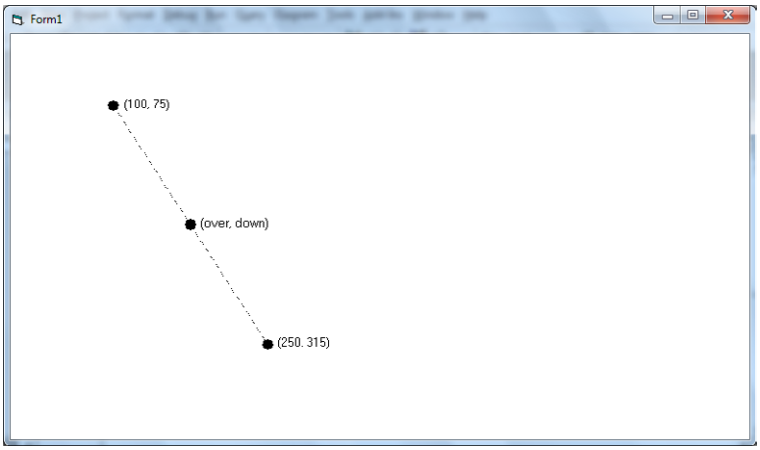


Figure 9-5: Three Points on the Line

We can now use the property of a line that says the slope is the same regardless of which two points on the line are used to compute it to set our two computations equal to each other:

$$\frac{\text{down - 315}}{\text{over - 250}} = \frac{\text{315 - 75}}{\text{250 - 100}}$$

Figure 9-6 summarizes the steps we must take to complete finding the equation of the line with endpoints (over1, down1) = (100, 75) and (over2, down2) = (250, 315) and an arbitrary third point on the line denoted as (over, down).

Step 1: Write the formula

$$\frac{\text{down - down2}}{\text{over - over2}} = \frac{\text{down2 - down1}}{\text{over2 - over1}}$$

Step 2: Substitute the known values

$$\frac{\text{down - 315}}{\text{over - 250}} = \frac{\text{315 - 75}}{\text{250 - 100}}$$

Step 3: Clear the left fraction of its denominator

down - 315 = (315 - 75)/(250 - 100) * (over - 250)

Step 4: Make the left side contain only the variable

down = (315 - 75)/(250 - 100) * (over - 250) + 315

Figure 9-6: Finding the Equation of the Line with Endpoints (100, 75) and (250, 315)

On Your Own

1. Find the equation of the line with endpoints (50, 100) and (125, 200). First represent the down coordinate as a function of the over coordinate.
2. Find the equation of the line with endpoints (50, 100) and (125, 200). First represent the over coordinate as a function of the down coordinate.

9.4 Using the PSet Command

To draw the line once the equation of the line is known involves changing the color of all the pixels with over coordinate:

$$over1, over1 + 1, over1 + 2 , \ldots , over2 - 2, over2 - 1, over2$$

where we assume over1 < over2. For the moment we will avoid dealing with the down coordinate of these points and just set up the code so that all these *over* values are used. The code is simply a **For** .. **Next** loop as shown in Table 9-4.

```
For over = over1 To over2

....

Next over
```

Table 9-4: Code to Visit All the over Values for Drawing a Line

In Figure 9-7 we see the *over* coordinate at 185 and 210 and how it needs to find the appropriate down coordinate for each step of its way.

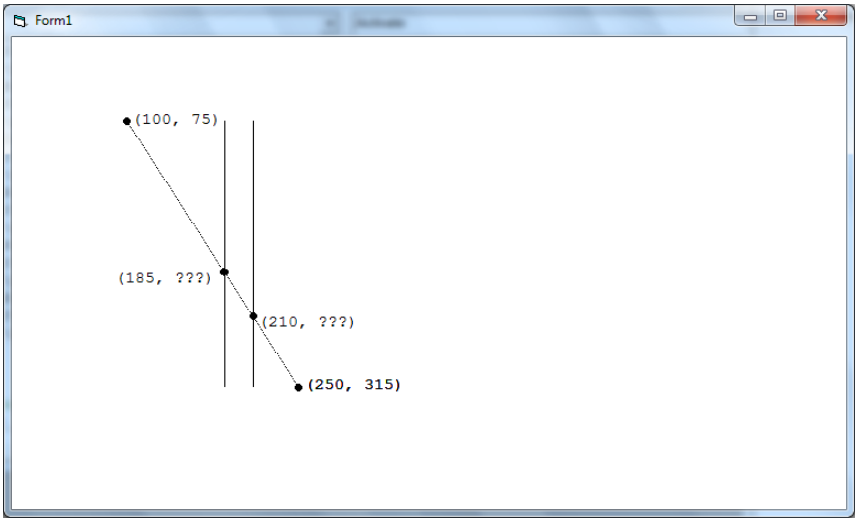


Figure 9-7: Relationship between over and down in Drawing a Line

The appropriate *down* is just the value of the equation of the line joining the two points evaluated at the current value of *over*. For a particular value of *over* its *down* coordinate on the line is just

$$down = (down2 - down1) / (over2 - over1) * (over - over2) + down2$$

The code to change the color of the pixel just for the point (over, down) is:

```
PSet(over, (315 - 75) / (250 - 100) * (over - 250) + 315)
```

Now to change the color for each pixel on the segment we just put this command into the **For** .. **Next** command that makes *over* take on all the required values. The code is shown in Table 9-5.

```
For over = 100 To 250
        PSet(over, (315 – 75) / (250 – 100) * (over – 250) + 315)
Next over
```

Table 9-5: Drawing a Line

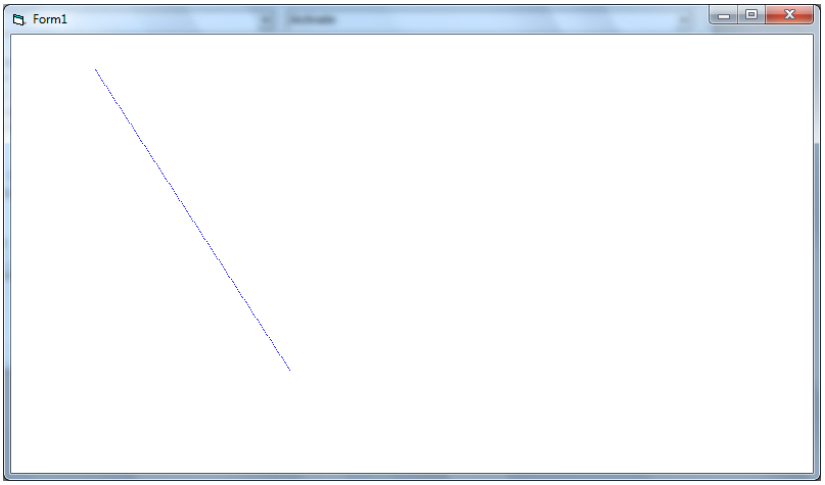
Another example of drawing a line is given in Example 9-3.

Example 9-3. Write a program that draws in blue the line segment with endpoints (75, 75) and (250, 375). Use the equation of the line:

$$down = (375 - 75) / (250 - 75) * (over - 250) + 375$$

where *over* ranges from 75 to 250.

OUTPUT:



SOLUTION:

```
    Rem Draw the line with endpoints (75, 75) and (250, 375)
    Rem OUTPUT: The specified line segment
        Cls
    Rem Draw the line in blue
A       For over = 75 To 250
B           down = (375 - 75) / (250 - 75) * (over - 250) + 375
C           PSet (over, down), vbBlue
        Next over
```

COMMENTS: The **For .. Next** loop beginning in **A** references each point with an over coordinate between 75 and 250. The assignment statement in **B** uses the equation of the line to calculate the corresponding value of *down*. The **PSet()** statement in **C** colors pixels in blue.

What we have really seen is that for any figure that can be described by an equation, we can draw the output of the function one pixel at a time. Unfortunately, most functions are not as easy to manipulate algebraically as a line. The principles are the same, however.

On Your Own

1. Write a program that draws a green line with endpoints (50, 100) and (125, 200).
2. Write a program that draws a line joining the points (55, 255) and (150, 320).

9.5 Using the Line Command

The line is a fundamental form that can be used to draw more complex figures, such as triangles and rectangles. The fact that a line can be part of so many figures would make it nice to have all the code encapsulated into a single BASIC command. We have seen effective encapsulation in the **For .. Next** command and something similar for drawing a line would be helpful. BASIC does, in fact, include a command to draw a line given only the two end points. The **SYNTAX** for this is:

SYNTAX

Line

Line (a1, b1) – (a2, b2), color

(a1, b1) and *(a2, b2)* are the starting and ending points of the line to be drawn. If (*a1*, *b1*) is missing, the line drawn starts at the pixel last accessed by the program but a hyphen must be included before *(a2, b2*).

color specifies the color in which the line is drawn. If *color* is not specified, the default foreground color is used.

Using the **Line** command saves us the trouble of computing the equation of the line we wish to draw. Table 9-6 shows the two different ways to draw the line (line segment) with endpoints (50, 25) and (80, 75).

Using PSet	**Using Line**
For over = 50 To 80 down = (75 - 25) / (80 - 50)*(over - 80) + 75 PSet (over, down) Next over	Line (50, 25)-(80, 75)

Table 9-6: Two Ways to Draw the Line with Endpoints (50, 25) and (80, 75)

In Table 9-6 the first way to draw the line is to use a **PSet()** command. The second way is to use the **Line** command. In the second way, the single **Line** command replaces the four lines which make up the **For .. Next** loop in the first way. It is obvious that using the **Line** command is much more convenient and intuitive. BASIC graphics commands are ways of allowing the programmer to think at a higher level instead of at the level of individual pixels.

We show the two lines:

a) **Line** (65,40) - (200, 30)
b) **Line** (150, 100) - (375, 230), **RGB**(200, 100, 100)

drawn with the **Line** command in Figure 9-8.

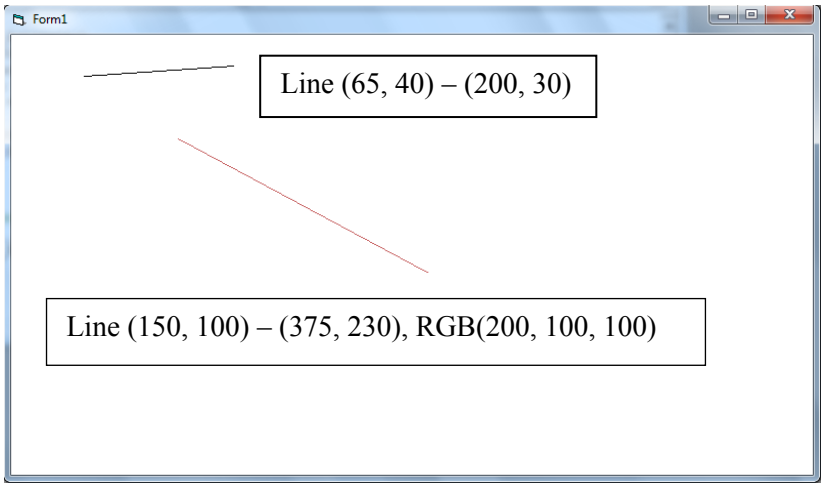


Figure 9-8: Drawing of Two Lines

9.6 Drawing Rectangles

Once we understand how lines are drawn by BASIC, we can ask if there are additional figures, such as rectangles, that can be as easily drawn. In Figure 9-9 we show the line joining (250, 140) and (375, 270).

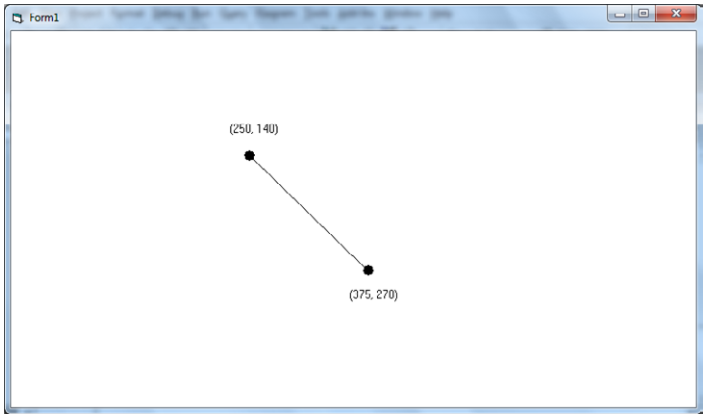


Figure 9-9: A Line and Its Endpoints

If the two endpoints in Figure 9-9 were opposite corners of a rectangle and not the endpoints of a line segment, we would need to find the coordinates of the other two corners to know how to draw the rectangle with these four corners. What we need to do is identify the corners labeled *B* and *C* in Figure 9-10.

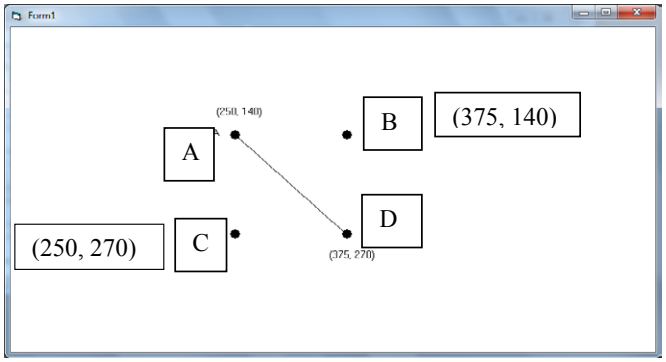


Figure 9-10: Corners of a Rectangle

We see that **C** is over the same amount as **A** so its over coordinate is 250. **C** is down the same amount as **D** so its down coordinate is 270. **B** is over the same amount as **D** so its over coordinate is 375, Finally, **B** is down the same amount as **A** so its down coordinate is 140. Now that we know the coordinates of these four points, we must get BASIC to draw the four lines that form the rectangle with these four points as corners. We draw

Line (250, 140) – (375, 140)
Line – (375, 270)
Line – (250, 270)
Line – (250, 140)

but not the line from (250, 140) to (375, 270). This is exactly what BASIC will do if we use an additional option in the line command.

SYNTAX

Line / Box Command

Line **(a1, b1) - (a2, b2), color,** B or BF

(a1, b1) and *(a2, b2)* are the starting and ending points on the display of the line to be drawn.

color specifies the color in which the line is drawn. If *color* is not specified, the default foreground color is used.

If **B** is specified, a box is drawn with *(a1, b1)* and *(a2, b2)* as opposite corners. **BF** produces a box filled in with the specified color or the default color if no color argument is used. You may use **B** without **F** but you may not use **F** without **B**.

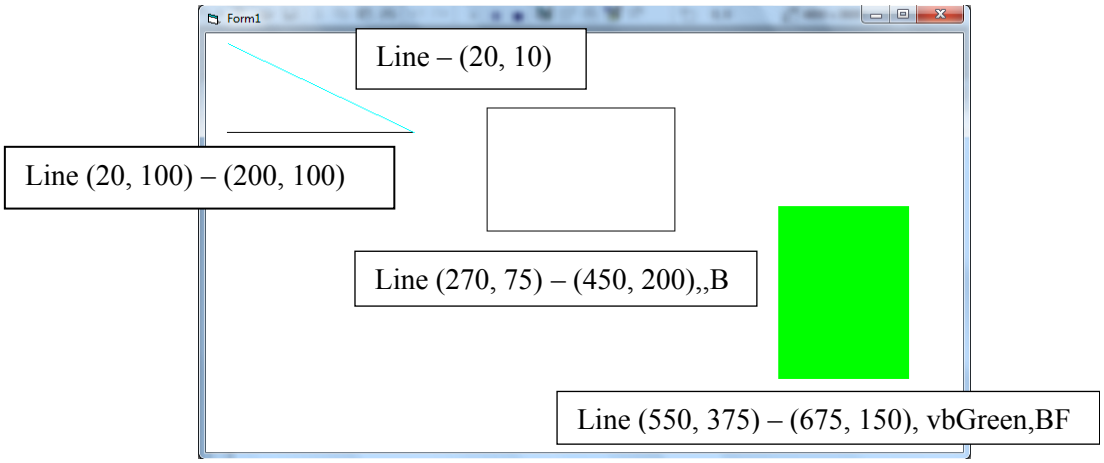
On Your Own

1. Explain why the following **Line** statements are invalid.
   a) Line
   b) Line (67, 89), **vbBlue**
   c) Line (78, 100)-(100, 99) **B**
   d) Line (24, 5)-(75, 150), **vbYellow**, F

2) What does the command **Line –(200,200)** mean?

Example 9-4. Using the variations of the **Line** command, write a program that draws two lines, one from (20, 100) to (200, 100) and the other from (200, 100) to (20, 10) in cyan. Also draw a box with corner points (270, 75) and (450, 200) and a filled box with corner points (550, 375) and (675, 150) having color green.

OUTPUT:



SOLUTION:

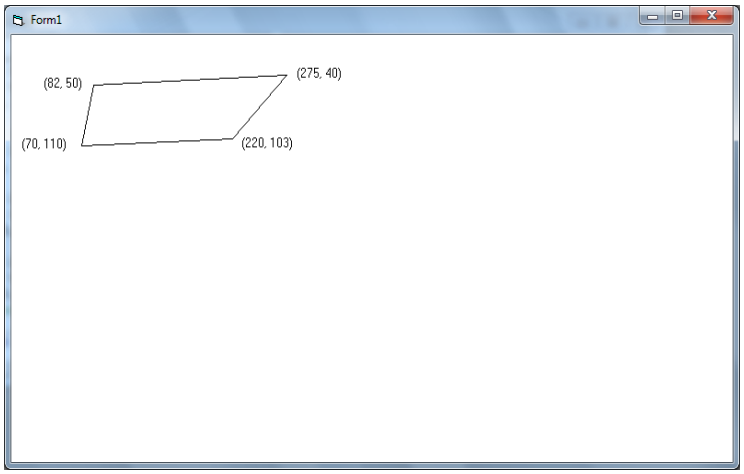
```
    Rem Draw two lines, a box
    Rem and a filled box.
    Rem Output: 2 lines, a box and a filled box colored green
      Cls
    Rem Draw a line using the foreground color
A     Line (20, 100) - (200, 100)
    Rem Draw a line in the color cyan
B     Line - (20, 10), vbCyan
    Rem Draw a box using the foreground color
C    Line (270, 75) - (450, 200), , B
    Rem Draw a filled box in the color green
D     Line (550, 375) - (675, 150), vbGreen, BF
```

COMMENTS: In the solution program, the **Line** statement in **A** draws a line from (20, 100) to (200, 100) in the default foreground color. **B** draws a line from the last referenced point, (200, 100), to (20, 10) in the color cyan. **C** draws the boundary edges of a box with (270, 75) and (350, 200) as opposite corners using the foreground color. **D** draws and fills-in with the color green a box with (550, 375) and (675, 150) as opposite corners.

Unfortunately, the **Line** command can only draw rectangles automatically but not parallelograms or trapezoids. Other four sided figures have to be drawn using the **Line** command four times.

Example 9-5. Use the **Line** command to draw the polygon shown below.

OUTPUT:

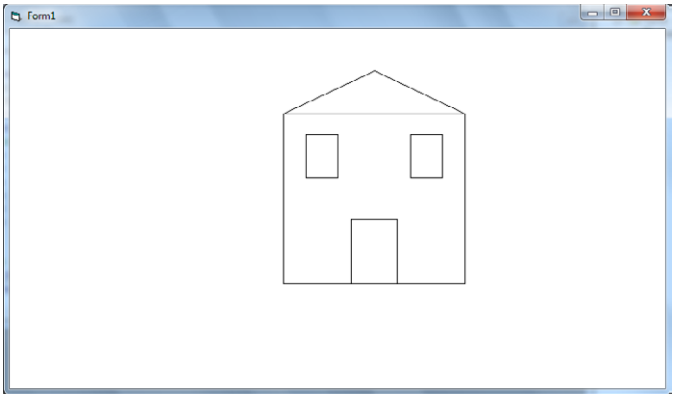


SOLUTION:

```
Line    (82, 50) - (275, 40)
Line             - (220, 103)
Line             - (70, 110)
Line             - (82, 50)
```

On Your Own

1. Write a program that draws a picture of a house. Your house should look like the figure below:



Notice that all the features are lines and boxes (the door is a box, the window frames are boxes, and a portion of the house without the roof is a box).

9.7 Drawing Circles

Now that we have seen how BASIC encapsulates drawing lines into a single command that can be extended to draw rectangles, we want to explore one other BASIC figure that is drawn with a single command. The figure is the circle. Since a circle can be drawn if we know its center and its radius, it is not surprising that BASIC's circle command asks for these two values. The syntax is described next.

The following are examples of valid **Circle** commands that draw the circles in Figure 9-11.

Circle (150, 200), 75

Circle (400, 200), 75, **vbRed**

Circle (400, 300), 75

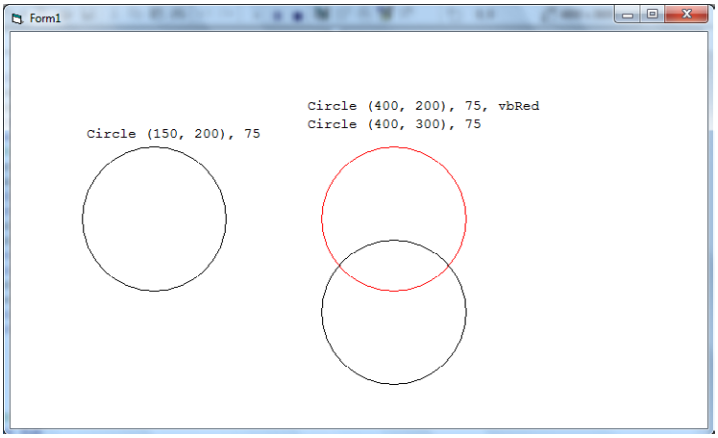


Figure 9-11: Three Circles

Filling in a circle with a color will be explained when we discuss **FillStyle** and **FillColor**.

On Your Own

1. Explain why the following Circle statements are invalid.

   Circle (90, 45)
   Circle (10, 100); 5
   Circle , 50, **vbCyan**

2. Write a program that draws a circle centered at (100, 75) with radius 30 pixels. Any color can be used for the boundary.

SYNTAX

Circle

Circle (over, down), rad, color

(over, down) is the pixel location of the center of the circle. rad is the radius of the circle and color indicates the color to be used to draw the circle boundary.

9.8 Drawing Arcs

Suppose there is a circle with center (a, b) and radius rad. The circumference of the circle, which measures the length of the boundary of the circle, is 2π rad. To draw an arc of the circle we need to identify the starting and ending points along the boundary of the circle. We do this by indicating how far around the circle from (a+rad, b), we traverse to get to the starting point of the arc. We then indicate how far the ending point is from the same starting point. Thus, if we include in the **Circle** command both the starting and ending points of an arc in terms of where these points are located as we traverse the circumference of the circle, BASIC can draw just that portion of the circle. To draw an arc that starts at (a+rad, b) we are talking about a starting point with zero distance from this point along the circumference of the circle. BASIC does not allow 0 as a starting point but requires we use some small value like 0.01. In Figure 9-12 we see a circle labeled with the measure for 1/4, 1/2, and 3/4 of the way along the circumference. If we define

pi=3.14159

these points are pi/2, pi, and 3 * pi /2 radians from the starting point.

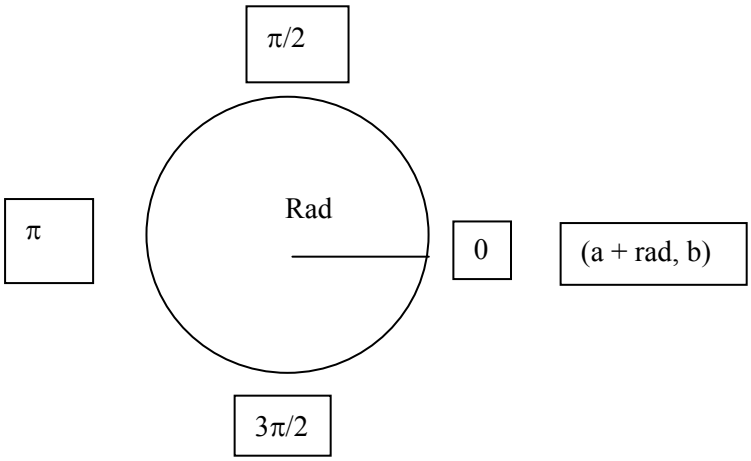


Figure 9-12: Distance Along the Circumference of a Circle

SYNTAX

Circle (Arc)

Circle (**over**, **down**), **rad**, **color**, **start**, **end**

(over, down) is the center of the circle, rad is the radius of the circle, and color indicates the color to be used to draw the circle. **Start** and **end** indicate the starting and ending points of an arc on this circle. **Start** and **end** are distances along the circle from the starting point (over +rad, down).

Arcs are always drawn in the counter-clockwise direction. Suppose that we want to draw the semi-circle that makes up the right half of a full circle. See Figure 9-13.

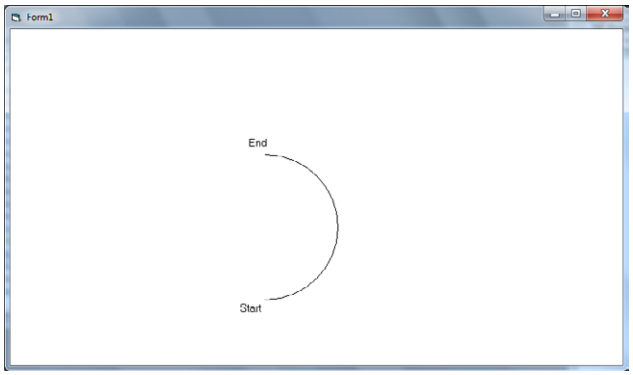


Figure 9-13: Semi-circle that Makes up the Right Half of a Full Circle

To draw this semi-circle, we would specify 4.71239 (3π/2) as the *start* parameter. We would then draw the arc in the counter-clockwise direction and specify 1.57079 (π/2) as the *end* parameter. Thus, the following statement would produce the semi-circle shown in Figure 9-6.

Circle (350, 250), 85, , 4.71239, 1.57079

The arc begins at 3π/2 and continues in the counter-clockwise direction until it reaches π/2. The pair of consecutive commas indicates the default color is used to draw the arc.

It is very important not to try to draw an arc in the clockwise direction. If you do so, your results will be very different from what you expect. For instance, suppose we tried to draw the arc in Figure 9-13 in a clockwise direction. We would have then specified the *start* parameter as 1.57079 and the *end* parameter as 4.71239 as in the following statement.

Circle (350, 250), 85, , 1.57079, 4.71239

Our coordinates would have been backwards, and we would see the results shown in Figure 9-14.

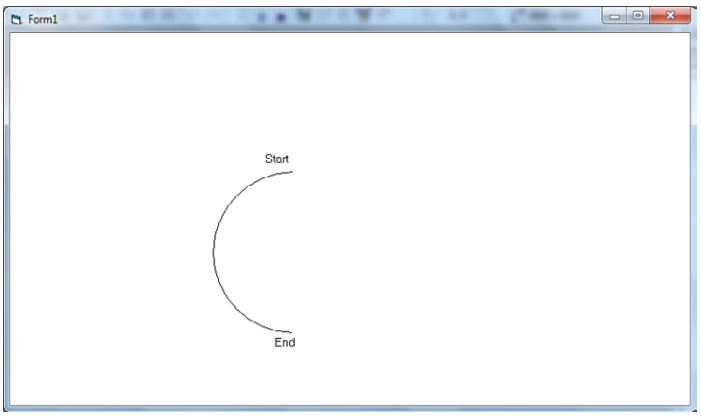


Figure 9-14: Result of Drawing an Arc with Backwards Parameters

When working with distance along the circumference of a circle, it is useful to create a variable called *pi* and use that variable in **Circle** commands rather than computing distances along the boundary of a circle directly. If you have the statement

pi = 3.14159

then you can use *pi* in **Circle** commands. Assuming the variable *pi* has been defined, Figure 9-15 shows the arcs drawn by the following commands:

a) **Circle** (160, 200), 70, , pi / 2, pi
b) **Circle** (370, 200), 80, , 7 * pi / 8, pi / 3

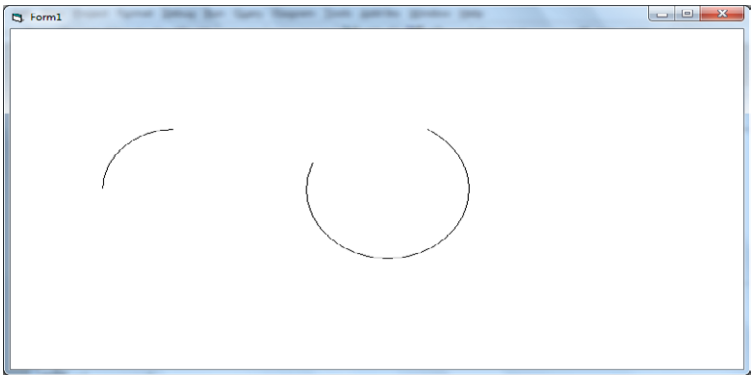


Figure 9-15: Example Arcs

On Your Own

1. Write a program to draw on the screen an arc centered at (200, 75) with radius 50 pixels. The arc should begin at 0 radians and end at 3π/4 radians. Any color can be used.

9.9 Drawing Sectors

A **sector** is a part of a circle that looks like a piece of pie. A sector has lines that extend from the ends of an arc to the center of the circle. Figure 9-16 shows a sector of a circle with radius 100 with its center at (250, 150) and its arc extending from 0 to 3π/2.

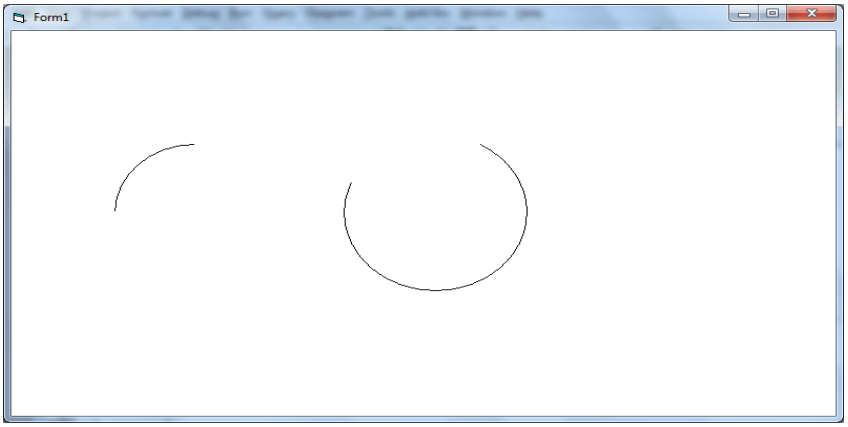


Figure 9-16: An Example Sector

Drawing a sector is easy. It is exactly like drawing an arc with the **Circle** command except that the *start* and *end* arguments are specified as negative values. Using a minus sign is a convention that BASIC uses as a signal telling it to draw a sector and not just an arc. Since the start and end points of an arc can never be negative, using the minus sign allows BASIC to know what to do. To draw the arc of the sector in Figure 9-16, we would use the statement:

Circle (250, 150), 100, , .01,3* pi /2

(This uses the variable *pi* defined above.) To draw the sector of this arc, we would simply place negative signs in front of the *start* and *end* parameters as follows:

Circle (250, 150), 100, , -.01,-3*pi / 2

SYNTAX

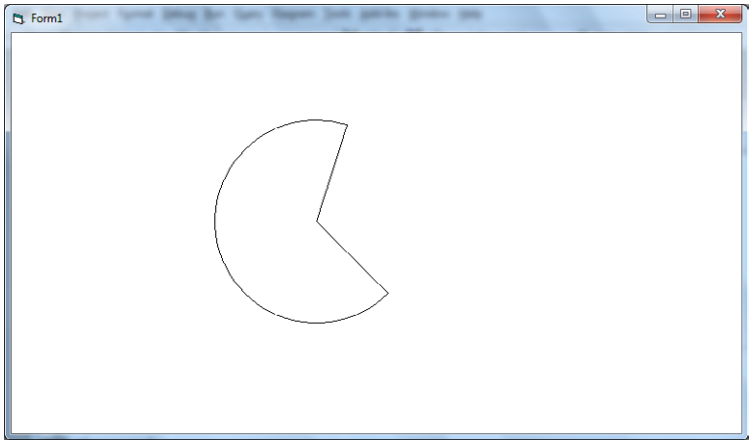
Circle (Sector)

Circle (**over**, **down**), **rad**, **color**, -**start**, -**end**

(over, down) is the center of the circle, rad is the radius of the circle, and color indicates the color to be used to draw the circle. Start and end indicated the starting and ending points of an arc on this circle that will be drawn. The minus sign on either or both of the **start** and **end** arguments will cause BASIC to draw a line from the center of the circle to the position on the circle where the arc starts and/or ends.

Example 9-6. Use the **Circle** with center (300,200) and radius equal to 100 to draw a sector that starts 1/5 of the way around the circle and ends 7/8 of the way around the circle.

SOLUTION: **Circle** (300,200), 100 , ,-2/5*pi, -7/4*pi



On Your Own

1. Write a program to draw a sector with its arc from π radians to 0 radians. The center of the sector should be at (100, 50), and its radius should be 30 pixels. Any colors may be used.

9.10 Drawing Ellipses

An ellipse is a circle that has been elongated or lengthened or flattened. The last parameter to the circle command, **aspect**, is used to specify the ratio of the vertical diameter to the horizontal diameter. If **aspect** is equal to 1 then a circle is drawn. The following command

Circle (245,185), 100, , , , 0.5

produces the ellipse shown in Figure 9-17.

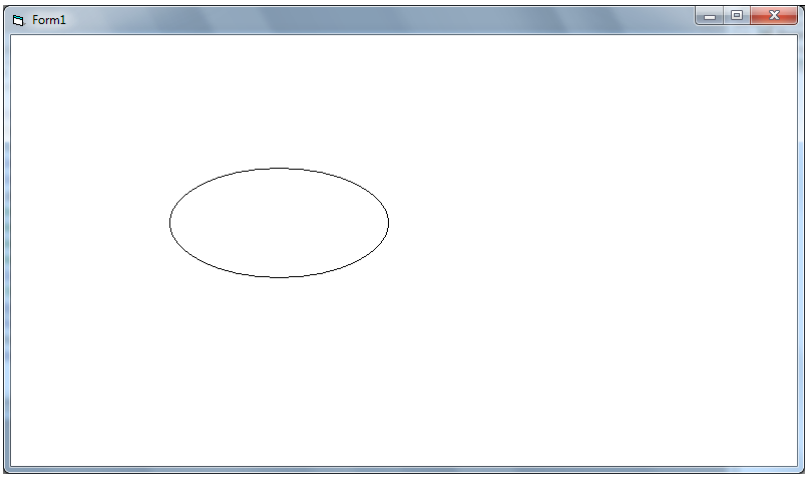


Figure 9-17: An Example Ellipse

SYNTAX

Circle (Aspect)

Circle (**over**, **down**), **rad**, , , , **aspect**
where aspect is a value between 0 and 1

If aspect is 1, the figure drawn will be a circle. If the aspect is less than 1, a flattened circle will be drawn.

For aspect equal to 1/2, an ellipse will be drawn. With the aspect equal to 0 a line will be drawn

9.11 Fill Styles

When a rectangle or a circle is drawn, it can be filled with a solid color or a pattern. The built-in variable **FillStyle** controls what pattern will be used to fill a figure. Initially **FillStyle** is set to "transparent," no filling is used and anything already inside the figure remains visible. To draw a figure filled with a solid color, use the command

FillStyle = 0

before drawing the figure. The command **FillStyle** needs a color to work with. If none is specified, **ForeColor** will be used with the particular style chosen. For example, the following commands draw a circle with a blue border and filled with yellow.

FillStyle = 0
FillColor = **vbYellow**
Circle (250, 150), 100, **vbBlue**

The BASIC **FillStyle** settings are listed in Table 9-7. Up to this point **FillStyle** has been assigned the default value of 1.

	FillStyle
Setting	**Description**
0	Solid
1	Transparent
2	Horizontal line
3	Vertical line
4	Upward diagonal
5	Downward diagonal
6	Cross
7	Diagonal cross

Table 9-7: FillStyle Options

SYNTAX

FillStyle and FillColor

FillStyle = value : FillColor = color

FillStyle indicates the pattern that will be used to fill in a closed curve such as a box, a circle, or a sector. **FillColor** allows user to choose a color to be used with **FillStyle**

The program in Figure 9-18 produces three shapes, each with a different **FillStyle** and a different color.

```
    Rem Draws three filled shapes on the display
    Rem OUTPUT: Shapes on the display
      Cls
      pi = 3.14159
    Rem Draw a filled sector
      FillStyle = 2 : FillColor = vbGreen
A     Circle (120, 100), 100, , -3 * pi / 4, -pi / 4
    Rem Draw a sector of an ellipse with a diagonal cross fill
      FillStyle = 7 : FillColor = vbCyan
B     Circle (350, 60), 100, , -7 * pi / 8, -pi / 8, 1 / 4
    Rem Draw a rectangle with a cross fill
      FillStyle = 6 : FillColor = vbYellow
C     Line (500, 180)-(650, 330), , B
```

Figure 9-18: Program that Draws Filled Shapes

The output of the program in Figure 9-18 is seen in Figure 9-19.

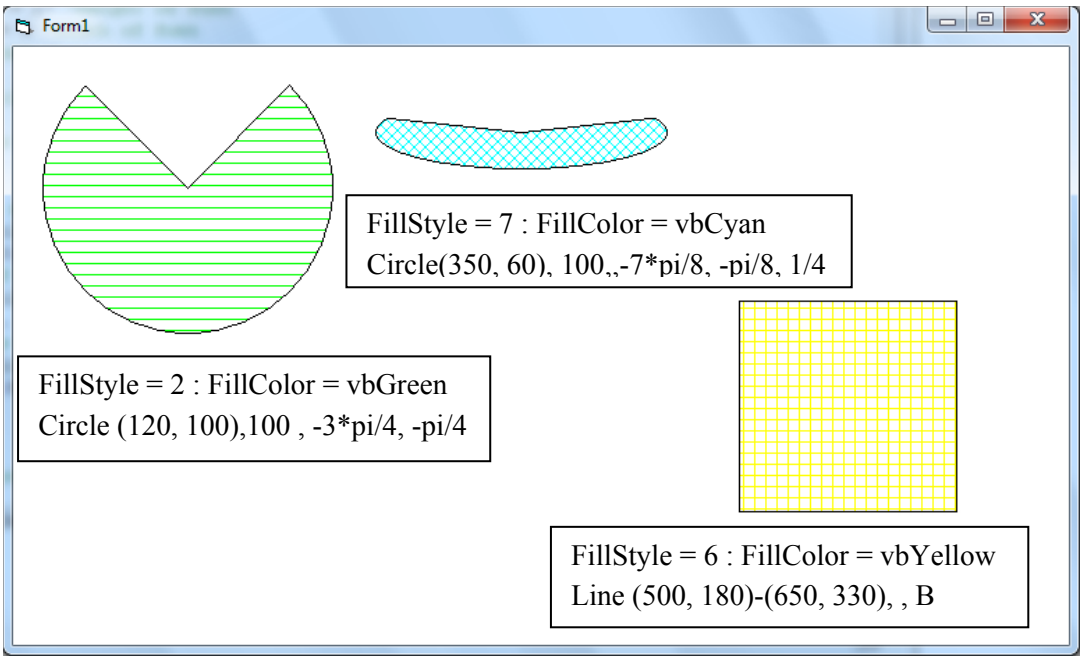


Figure 9-19: Output of Program in Figure 9-17

On Your Own

1. Write a program that draws two circles on a blue background. One of the circles should be outlined in cyan but filled in magenta. The other circle should be both outlined and filled in green. Use different fill styles for the two circles. The circles may appear anywhere on the screen.

9.12 A Pie Chart

A common application of graphics with spreadsheet programs is the creation of a pie chart. With the use of the **Circle** command and different styles, data can be displayed in pie chart form. The parts of a pie chart are just sectors of a circle usually filled with different colors and/or styles. An example of a pie chart appears in Figure 9-20. This pie chart has three sectors, each comprising 33.3% of the chart.

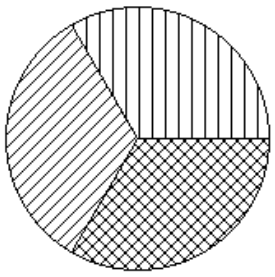


Figure 9-20: An Example Pie Chart with Three Equal Sectors

Example 9-7. Write a program that creates a pie chart with three sectors. The user enters the size of two sectors, and the chart is created based on that data.

SOLUTION:

```
    Rem Creates a pie chart with three sectors. The size of
    Rem the sectors are based on the user's data.
    Rem INPUT: Percent of circle representing size of each sector
    Rem OUTPUT: A pie chart with three sectors
      Cls
    Rem Get input-integer values for percentages
A     sector1 = Val(InputBox("Enter 1st sector percentage size."))
      sector2 = Val(InputBox("Enter 2nd sector percentage size."))
B     sector3 = 100 – sector1 – sector2
    Rem Initialize variables
C     pi = 3.14159
      radius = 80
      stopangle = .01
    Rem Draw each sector
D     For I = 1 To 3
E       If I = 1 Then
          sector = sector1
          FillStyle = 3
          FillColor = vbBlue
        ElseIf I = 2 Then
          sector = sector2
          FillStyle = 5
          FillColor = vbGreen
        Else
          sector = sector3
          FillStyle = 7
          FillColor = vbYellow
        End If
    Rem Calculate position and size of current sector
F     angle = (2 * pi) * sector / 100
G     startangle = stopangle
      stopangle = startangle + angle
H     If stopangle > 2 * pi Then
        stopangle = .01
      End If
    Rem Draw the sector
I     Circle (160, 100), radius, , -startangle, -stopangle
    Next I
```

COMMENTS: The program begins by allowing the user to input two sector sizes beginning in **A**. The third sector size is calculated (**B**) based on the first two. Beginning in **C**, variables that will be used later are initialized. The **For .. Next** loop beginning in **D** draws each sector individually. Within the loop, the **If** block beginning in **E** merely accesses the appropriate sector size to draw. Beginning in **F**, equations are used to calculate the size and position of the sector currently being drawn. For instance, in **G**, a reference is made to the ending position of the previously drawn sector. In **H**, the value of *stopangle* is checked to ensure that it is not greater than 2π. The **Circle** command at **I** actually draws the sector. The radius of the sector is the value which was initialized in **C**. **FillStyle** and **FillColor** are assigned appropriate values for each sector in the **If** block beginning at **E**.

On Your Own

1. Write a program that creates a pie chart with four sectors. Let three of the sector sizes be 20%, 25%, and 15%. The fourth sector is the remaining portion of the circle. Use different colors and fill style for each sector.
2. Calculate the starting and stopping points for the sectors of a pie chart where the three regions represent 50%, 25%, and 25% of the circle.

9.13 Histograms

When values need to be seen in comparison to other values as the sales for a sequence of months or the number of gyzmos sold each month, the chart called a histogram is effective. A histogram represents a set of values using a scale so that the relative difference in the magnitudes can be seen by observing the different heights of a set of rectangles that are proportional to the different values. The output form shown here displays a histogram for values collected over a period of four months denoted by J(anuary), F(ebruary), M(arch), and A(pril).

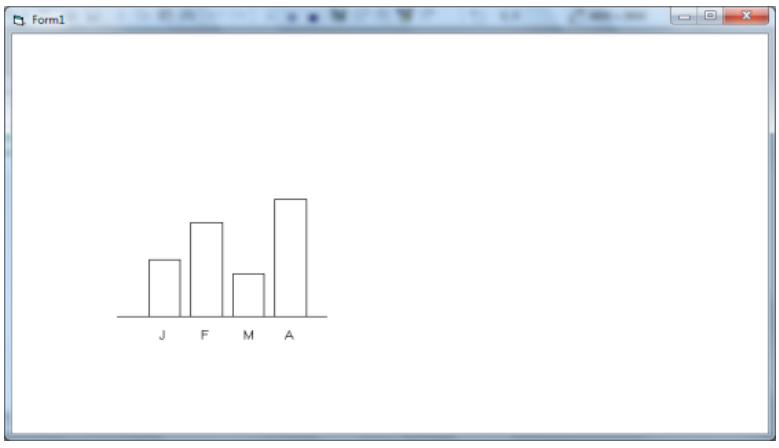


Figure 9-21: A Histogram

The code to draw the chart in Figure 9-21 is given in Example 9-8.

Example 9-8. Draw a histogram representing the four values 120, 200, 90, and 250 that represent sales in January, February, March, and April, respectively.

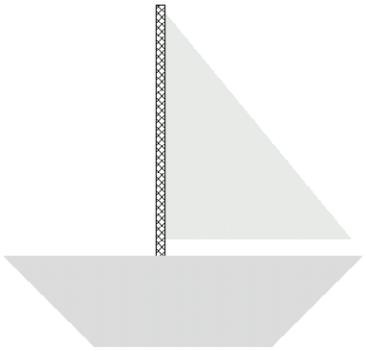
```
Rem: Draw a histogram with four values
Rem INPUT: Four values to be represented
Rem OUTPUT: Histogram representing the relative size of the input numbers
Line (110, 300)-(250, 300)
Line (120, 300)-(140, 300 – 3 * 12), , B
Line (150, 300)-(170, 300 – 3 * 20), , B
Line (180, 300)-(200, 300 – 3 * 9), , B
Line (210, 300)-(230, 300 – 3 * 25), , B
CurrentX = 125: CurrentY = 312
Print "J"
CurrentX = 155: CurrentY = 312
Print "F"
CurrentX = 185: CurrentY = 312
Print "M"
CurrentX = 215: CurrentY = 312
Print "A"
```

SOLUTION:

In Putting It All Together you will explore ways to make drawing a pie chart or a histogram much more flexible so that one program can serve a variety of requirement.

9.14 Putting It All Together

1. Write a program that sets background to blue and then draws the points (100, 10) to (100, 100) and (200, 10) to (200, 100). The points should first appear in the color green and then change to blue and then back to green. There should be pauses between the changes so that the user can see them occurring.
2. Write a program that draws a picture of a sailboat. Your sailboat may look something like the figure below:



3. Write a program that draws a green filled box on the screen. Within that box, draw another box with a red outline. Fill in the inner box with the color blue.
4. You can add labels to the sectors of a pie chart using code of the following sort when the circle is centered at (X, Y) and has a radius R.

```
alpha= (stopangle – startangle) / 2.0
beta = alpha + startangle
If beta < pi/2 Then
   currentX = X + cos(beta) * R/2
   currentY = Y – sin(beta) * R/2
   Print "Sector's label"
ElseIf beta < pi Then
   currentX = X – cos(beta) * R/2
   currentY = Y – sin(beta) * R/2
   Print "Sector's label"
ElseIf beta < pi * 3./2. Then
   currentX = X – cos(beta) * R/2
   currentY = Y + sin(beta) * R/2
   Print "Sector's label"
Else
   currentX = X + cos(beta) * R/2
   currentY = Y + sin(beta) * R/2
   Print "Sector's label"
End If
```

Example 9-8 uses three pixels of height to represent one unit in a category. The vertical axis should be labeled 3, 6, 9, … , 24, 27 units with the label 3 positioned 4*3 pixels about the horizontal axis.

a) Draw a line from (50,300) to (50,150) to represent the vertical axis.
b) For each of the points 300 – 4*4, 300 – 6*4, … , 200 -27*4 on the vertical axis draw a tic mark, a line with endpoints (300 – n*4, 50 – 5) and (300 – n*4, 50+5) for n = 1, 2, … , 10.
c) Use CurrentX and CurrentY to position the print head at

CurrentX = 50 – 5 – 20
CurrentY = 300 – n*4 -8

where n = 3, 6, 9, … , 24, 27. For CurrentX you want the print head to have room for two digits to the left of the start of the tic mark. For CurrentY you want the print head to print out the digit half above the tic mark and half below the tic mark.

5. Write a program that draws on the screen two overlapping circles. One circle should be solid red and the other should be solid blue
6. Write a program to draw a histogram with six vertical bars. The values represented are 25, 60, 85, 70, 90, and 55. Use two pixels of height to represent each unit. The categories should be labeled cars(25%), trucks (60%), SUV (80%), ATM (70%), Hogs (90%), and bikes (55%).
7. Repeat problem 6 but use horizontal bars for display.
8. Write a program that draws a pie chart with three regions. The regions represent 60%, 30%, and 10% of the circle. Draw the sector for the largest sector so that it pulled out of the circle. We call this an exploded sector.

10 Arrays and Tables

Programs to this point have input data for immediate use. We have not needed to use a piece of data entered a second time after some major computation step involving all the data values is completed. There are, however, instances such as finding all the scores greater than the average of the set of scores that cannot be solved by making a decision when a value is entered. In many problems that require finding how data values relate to some computed property, we need to process the data once to determine the property and then go through each data value again to compare it with the property. Certainly, we could reenter the data but that is both tedious and error prone. We could also use a different variable name for each data value so it would remain in memory after its use. Using different names for each data value leads to unnecessarily long programs that are also tedious to write and very error prone for the user. What we want is a convenient way to store any number of values of the same data type in memory and be able to reuse the values easily.

Figure 10-1 shows the output of a statistical analysis program.

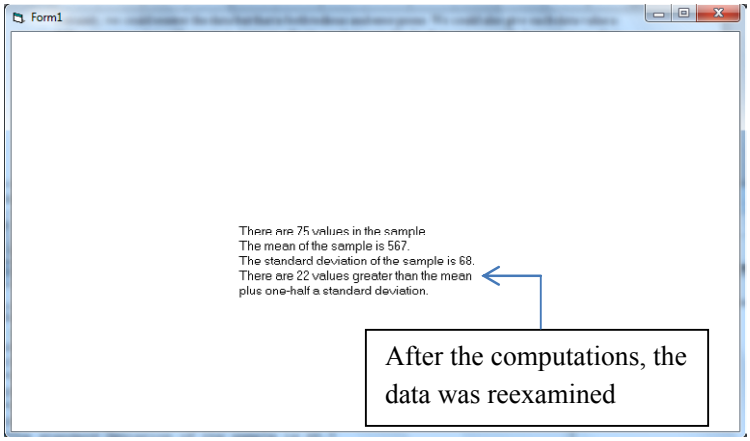


Figure 10-1: Statistical Analysis of Sample

A simple example of finding the values greater than the average using three values is shown in Figure 10-2. Just think of how this program would be written if there were thousands of data values.

	Rem INPUT: Three numbers
	Rem OUTPUT: The average of the numbers and
	Rem the numbers greater than the average
	sum = 0
A	grade1 = Val(InputBox("Enter first number:"))
	sum = sum + grade1
	grade2 = Val(InputBox("Enter second number:"))
	sum = sum + grade2
	grade3 = Val(InputBox("Enter third number:"))
	sum = sum + grade3
B	Rem Find average
	ave = sum / 3
	Print ave
C	If grade1 > ave Then
	Print grade1
	End If
	If grade2 > ave Then
	Print grade2
	End If
	If grade3 > ave Then
	Print grade3
	End If

Figure 10-2: Finding Values Greater Than the Average for Three Data Values

In Figure10-2 the code starting with **A** enters three data values and assigns each to a separate storage location. The storage locations are named *grade1*, *grade2*, and *grade3*. The data values are added together as they are entered as this is required to find the average. In **B** the summing is finished and the average is calculated. In **C** the program must revisit each of the data values to determine if it is greater than the average.

10.1 Defining an Array

In the example above we needed to store three values that each represented a grade. Each value was assigned to a separate storage location by means of the variable names used. For sets of data values that are of the same kind, such as, grades, names, tax rates, BASIC allows the programmer to name a storage area consisting of any number of storage locations each of which can be used for similar kinds of data. Such a storage area is called an **array**. For the example in Figure 10-2, we need a storage area with three storage locations as shown in Figure 10-3.

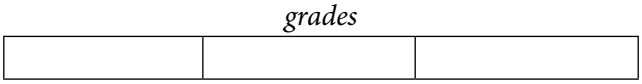


Figure 10-3: Storage Area Named grades

There is still one problem to solve. How do we reference a single one of these storage locations? Remember *grades* is the name of all this storage. To differentiate the individual storage locations, BASIC simply numbers the storage locations allocated when the storage area is defined. Normally, the numbering of the storage locations begins with 1 and proceeds from left to right giving each other storage location a number one more than its left neighbor. The number of the storage location is called a **subscript**. We now have a model for grades that looks like the one in Figure 10-4.

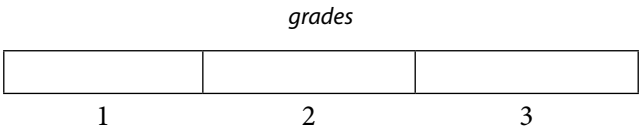


Figure 10-4: *grades* with Subscript Numbering

10.2 The Syntax of Defining Arrays

To use arrays and subscripts to reference a storage location and/or its contents BASIC uses a two part syntax. The first part of the syntax explained here is the statement used to allocate a storage area with properly numbered storage locations. In addition the programmer must tell what kind of information (**Integer**, **Single**, **Double**, or **String**) will be put in each location in the array. All locations in an array must be used for the same kind of data value. **Integer** values are whole numbers between -32768 and 32767. A **Double** value is any numeric value that is to be treated as if it contained a decimal point. Both 35 and 35.0 are examples of values of type **Double**. The value 35 could also be considered of type **Integer** but 35.0 could only be considered of type **Double**. **Single** is a less flexible data type than **Double** that also deals with numeric values containing a decimal point. The syntax for defining an array is:

SYNTAX

Array-Dim Statement

Dim arrayName (num1 **To** num2) **as** dataType

arrayName : valid variable name

num1, num2: range of array-subscript range with num1 < num2

dataType **Integer, Single, Double, or String**

EXAMPLE: Dim grades (1 **To** 10) **as Double**

x = grades(2)

Many times the program will allocate more storage locations than needed for a particular instance of a problem. It will make no difference to BASIC and the program will have the flexibility to be used for different number of data items without having to change the program. The program is responsible to keep track of how many storage locations are used regardless of how many are in the range of the array. Of course, the program cannot try to use more storage locations than are allocated.

10.3 Assigning and Using Values in an Array

In the last section we saw how to allocate a block of memory with a number of storage locations for holding values of the same type. There were two key parts to the definition. The first was the name of the whole area allocated. The name for an array is formed following the rules for making a variable name. We must now see how BASIC accesses a particular one of the storage locations in the area defined in a **Dim** statement. The second part of the name for a location in an array is called a subscript and indicates which location in the array is being referenced. A subscript is included in parentheses following the array name.

SYNTAX

Array Elements

Dim variable (num1 To num2) as dataType

Individual elements are accessed using a two-part name. The first is the name of the storage area. The second is the number of the particular location in the array being referenced written in parentheses following the array name. The result is a variable name.

EXAMPLE: Dim grades (1 To 10) as Double
grades (2)

An example using array elements is shown in Figure 10-5. BASIC automatically initializes storage locations in arrays containing numeric values to zero.

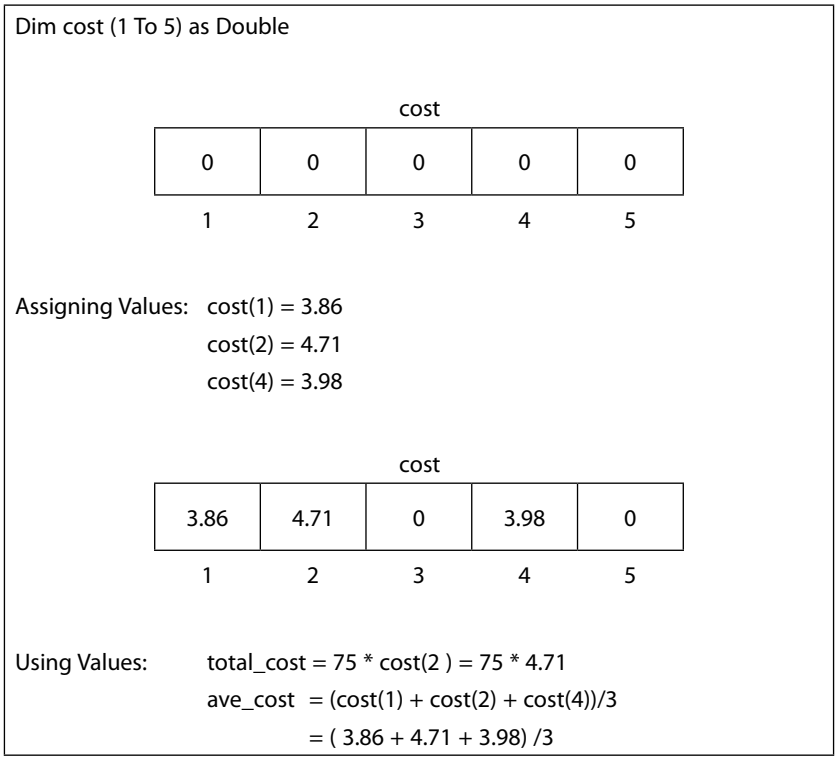


Figure 10-5: Using Array Elements

A variable name indicating a location in an array is really no more than a variable name with a slightly different format because of the subscript. The advantage to an array is that it is easy to access each of its values by merely changing the value of the subscript. The program in Figure 10-6 shows how arrays can be used with a **For** .. **Next** loop

```
Dim surName (1 To 10) as String
Dim carModels (1 To 20) as String
N1 = Val(InputBox("# surnames"))
For I = 1 To N1
        surName(I) = InputBox("Enter surname:")
Next I
N2 = Val(InputBox("# car models"))
For I = 1 To N2
        carModels (I) = InputBox("Enter car model:")
Next I
Rem Print selected car models & surNames
Print "Output car models"
For I = 1 To N2 Step 3
        Print carModels(I)
Next I
Print "Output surnames"
For I = 1 To N1 Step 2
        If surName(I) > "M" Then
                Print surName(I)
        End If
Next I
```

Figure 10-6: Arrays and For..Next

On Your Own

1. Allocate a storage area with seven locations numbered 10, 11, … , 16.
2. Allocate a storage area with ten locations numbered 0, 1, 2, … , 8, 9.
3. Allocate a storage area with eight locations numbered 2, 3, …, 9. Assign 1, 3, 5, 7, 9, 11, 13, and 15 to these locations starting with 1 in location 2.
4. Allocate a storage area with seven location numbered 1, 2, … , 7 with 3* location number as an initial value.

10.4 Finding the Average and the Standard Deviation

Now that we have the tools to use an array, it is instructive to see how the program we used to introduce the idea of an array can be written using this new feature. The program similar to the program shown in Figure 10-2 is shown in Figure 10-7. The difference is that this program uses both the average and the standard deviation to identify special values.

```
A   Dim grade (1 To 10) as Integer
    sum = 0 : sumSq = 0
    N = Val(InputBox("Enter Number of Grades"))
    For I = 1 to N
B           grade(I) = Val(InputBox("Enter a grade"))
            sum = sum + grade(I)
            sumSq=sumSq + grade(I)^2
    Next I
    ave = sum / N : stDev = sqr(1/N* sumSq – ave^2)
    Print "Grades > ";ave + Stdev
    For I = 1 To N
C           If grade(I) > ave + stDev Then
                    Print grade(I)
    End If
    Next I
```

Figure 10-7: Grade Program Using an Array

In **A** the array is defined. Notice the program could be used for larger data sets as the array is defined to contain ten locations numbered from 1 to 10. In **B** the program reads the data and stores each data value in a different location in the array *grade*. The code also accumulates the values needed to compute both the average and the standard deviation of the grades. Finally, in **C** the program revisits the elements of *grade* to determine which of the values are larger than the average plus one standard deviation. (The standard deviation measures how values cluster around the average.)

10.5 File Input

The model for using arrays to this point will normally require using **InputBox()** to assign values to array locations each time a program is run to enter the data. With applications using arrays it is often the case that there is a lot of data to process. Entering each data value using the **InputBox**() approach is very time consuming and can be quite prone to error. To make dealing with large amounts of data more convenient, a very restricted version of file input will be shown. The model given can be used with any program but is most useful with entering values into an array. The programs you write can take the block of code that is introduced and just copy it into a program whenever file input is needed. The only other problem to deal with is how to create a file for the data. A file can be created in Word if the result is saved as having type txt. You can choose this file format option in Word in the **Save As Type** pull down menu when you go to save the data file.

For use in this book we use a very special format for the data files so the input code template is simpler. The restriction on the format of a data file is that the first value must be the number of data items. The data items follow one after another after the initial number of data items entry with each entry in the file separated from the next (including the number of data items from the first data item) by a comma. Suppose the data consists of five values: 4, 7, 5, 2, 9. The file would be created as:

5,4,7,5,2,9

and saved as *filename.txt*. The syntax and code used to enter a data file so that each value is in a different location in an array has a different syntax from the **InputBox()** function.

SYNTAX

Read From A File

Open "filespec" for Input as #n

```
Input #n, N
For I = 1 To N
        Input #n, dataValue(I)
Next I
Close #n
```

filespec**:** path name to file
n: an integer value
dataValue(): arrary and subscript
Close: releases the link from BASIC to the file

EXAMPLE:

```
Open "PATH\dataFile.txt" for Input as #1
Input #1, noElements
For I = 1 To noElements
        Input #1, dataValue(I)
Next I
Close #1
```

The **Open** statement makes a connection between the data file and the BASIC program by telling the program how to access the data file in the local file space. For purposes of our use the filename will be of one of the forms:

\\netspace\students\init\logon\public\filename.txt

\\netspace\departments\computer_science\public\filename.txt

In the first path, *init* is the first letter of your surname and the *logon* is your user name. The file filename.txt will be whatever file is used in the program. Obviously, if you are not using this file system, you will have to determine the **filespec** in terms of your computing environment. In Windows you can use Windows Explorer to find the path you need BASIC to know. In general, the **filespec** will be shortened as **PATH***filename*.**txt** where you will need to replace **PATH** with what is appropriate for your file system. The code you will actually use to access the elements in a data file is shown in Figure 10-8.

```
Dim arrayName (1 To 100) as Double
Open "PATH\dataFile.txt" for Input as #1
Input #1, N
For I = 1 To N
        Input #1, arrayName(I)
Next I
Close #1
```

Figure 10-8: Code to Enter Data into an Array from a File

The block of code shown in Figure 10-8 can be used for any array input operation needed by just forming PATH and using the appropriate dataFile.txt name. Once the PATH is used in the **Open** statement, it is not referenced again in the program.

10.6 Using Arrays – Searching an Array

Once an array is assigned values, a common operation for a user to request is to find out if some value is in one of the array locations. This operation is called searching an array. The code to carry out the search of an array is shown in Figure 10-9.

```
    Rem Determine if a value is stored in an array and returns its location if found
    Rem INPUT: An array with N elements and and a value to search for
    Rem OUTPUT: The location of the element in the array or message "NOT FOUND"
A   Dim elements (1 To 100) as Double
    Open "PATH\dataFile.txt" for Input as #1
    Input #1, N
    For I = 1 To N
            Input #1, elements(I)
    Next I
B   Close #1
C     searchElement=Val(InputBox("Enter value to be found"))
      location = -1
      For I = 1 To N
D           If searchElement = elements(I) Then
                location = I
               GoTo L1
            End If
      Next I
E   L1: If location = -1 Then
            Print searchElement; " NOT FOUND."
     Else
            Print searchElement; " was found at location:";location
     End If
```

Figure 10-9: Search Code

The code from **A** to **B** dimensions an array and reads a file into the array. The element that is the object of the search is entered by the user at **C**. In **D** the program examines each entry in the array to see if any of the elements match the element sought. The transfer to **E** when the element is found lets the program avoid searching additional array locations when the answer is known. If the element is not found, after the whole array is searched, control passes to **E** with *location* still having its initial value of -1.

10.7 Using Arrays – Finding a Smallest Element in an Array

Putting the elements in an array in increasing or decreasing order is called sorting. To understand a simple sorting program, there are two parts of the process that need to be explained. In this section we examine the problem of finding the smallest element in some range of subscripts. In the next two sections we finish our exploration of sorting. In Figure 10-10 we show how to find a smallest element in an array.

```
    Rem Find the smallest element in an array
    Rem INPUT: The number of string elements and the strings
    Rem OUTPUT: The smallest value in the array

A   Dim stringElements (1 To 100) as String
B   Open "PATH\dataFile.txt" for Input as #1
    Input #1, N
    For I = 1 To N
            Input #1, stringElements(i)
    Next I
C   Close #1
D   smallest = 1 'first estimate of location of answer
    For I = 2 To N
            If stringElements(smallest) > stringElements(I) Then
                    smallest = I
            End If
E   Next I
F   Print smallest, stringElements(smallest)
```

Figure 10-10: Finding a Smallest Element

In **A** the program defines an array which can hold up to 100 elements of type **String**. A file of elements is identified and its elements are input into the array in the code from **B** to **C**. The code from **D** to **E** compares each element successively in the array with the smallest element encountered to that point. If an element is found that is smaller than the smallest to that point, we update the location of the smallest element encountered so far. When we get to **F**, the location in *smallest* gives the location of the smallest element in the array. The smallest means the least element in the array as determined by the ordering relation on that type of element. The ordering could be either the natural numeric ordering for numbers or the dictionary ordering for strings.

10.8 Using Arrays – Interchanging Two Elements in an Array

In the process of sorting we first find the smallest element in the array and then interchange it with the element in the first location. When we finish, the smallest element is in the first location of the range searched and the original element in that location is now in the location that originally contained the smallest element in the range. Figure 10-11 shows the general process of interchanging two elements in an array where one is located location J = 2 and the second at location K = 5. Interchanging two elements need not involve the smallest element in a range of subscripts as we see in this example.

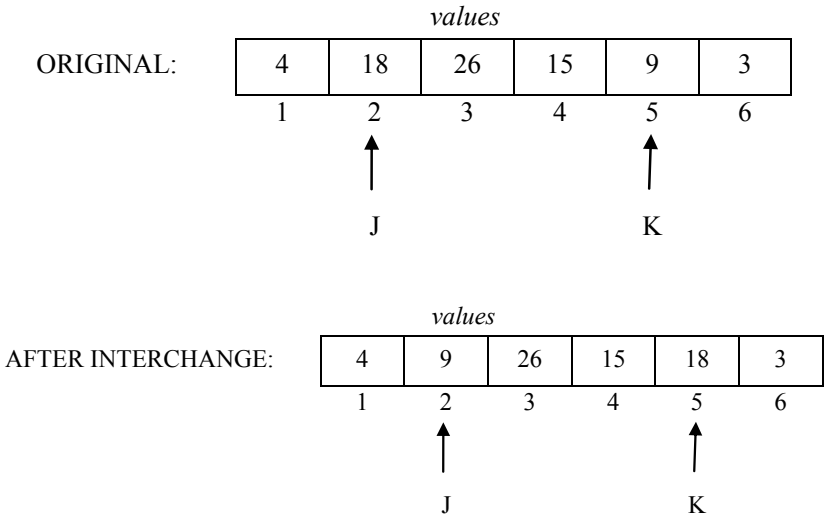


Figure 10-11: Interchanging Two Elements in an Array

To interchange two elements in an array, we must know the location of each of the values, say *J* and *K*. It is tempting to write the following code to interchange the elements at these two locations:

elements(J) = elements(K)
elements(K) = elements(J)

If we trace this code for the original values:

elements(J) = 18
elements(K) = 9

we first assign *elements (K)* to *elements (J*) giving

elements(J) = 9

If we now assign *elements(J)* to *elements(K)* we get

elements(K) = 9.

The difficulty arises because the value of *elements(J*) is changed to the original value in *elements(K)* before we assign the contents of *elements(J)* to *elements(K*). The solution to the problem of actually interchanging two elements is to assign the original value of *elements(J*) to a separate variable before we assign the value in *elements(K)* to *elements(J).* We can then assign the value in the separate variable to *elements(K*) and get the two elements interchanged as we intended. The code to solve this problem is shown in Figure 10-12.

```
     Rem Interchange two elements in an array. Print the array before
     Rem and after the interchange
     Rem INPUT: An N element array and two locations of elements
     Rem to interchange
     Rem:OUTPUT: The array before and after the interchange
A    Dim elements (1 To 100) as Integer
     Open "PATH\dataFile.txt" for Input as #1
     For I = 1 To N
          Input #1, elements(I)
     Next I
     Close #1

B    For I = 1 To N
          Print I, elements(I)
     Next I
C    J=Val(InputBox("First interchange location"))
     K=Val(InputBox("Second interchange location"))

     Print J,elements(J), K, elements(K) 'original places for elements
D    temp = elements(K)
     elements(K) = elements(J)
     elements(J) = temp

     Print J,elements(J), K, elements(K) 'see what has changed
E    For I = 1 To N
          Print elements(I)
     Next I
```

Figure 10-12: Interchanging Two Elements in an Array

Starting at **A** the array is defined and the file of elements is read into the program. The loop starting at **B** is simply a printing of the elements in the original order. The two values entered at **C** are the array locations whose elements need to be interchanged. In the code starting with **D** the actual interchange takes place. Notice the use of the variable *temp*. The loop starting at **E** prints the array with the two elements interchanged.

10.9 Using Arrays – Sorting an Array

For the sorting process we will use a simple algorithm that has two major steps that are repeated. The two steps are first to find the smallest element in a range of storage locations and then interchange that smallest value found with the first element in that range of locations. Since we build up a sorted array one element at a time, we will need to repeat this process with an increasingly smaller number of elements in the search area until all the elements are in sorted order. The coding tool we use is a nested **For** .. **Next** construct.

For the sorting process we first need to find the smallest element in the array and put that element in location 1. The second step is to find the smallest value remaining in locations 2, 3, … , *N* and put that element in location 2. The key is to have a smaller range of locations to search each step so previously found smallest elements are not found again! We show an example for an array with four elements in Figure 10-13.

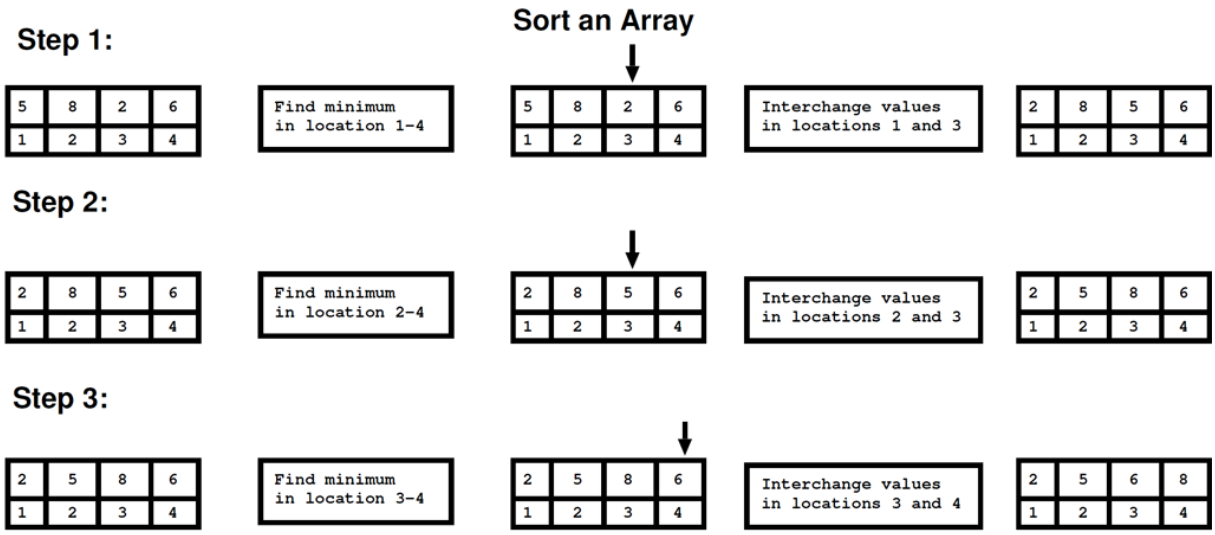


Figure 10-13: Minimum Selection Sorting Example

The code that actually carries out the sorting is shown in Figure 10-14.

```
    Rem Sort an array with N elements
    Rem INPUT: The number of elements in an array and the elements
    Rem OUTPUT: The N elements of the array in increasing order

    Dim values (1 To 100) as Double
    Open "PATH\dataFile.txt" for Input as #1
    Input #1, N
    For I = 1 To N
        Input #1, values(I)
    Next I
    Close #1

A   For I = 1 to N – 1
        smallLocation = I 'first estimate of location ofanswer
        For J = I + 1 to N
B           If values(J) < values(smallLocation) Then
                smallLocation = J
            End If
        Next J
        temp = values(I)
C       values(I) = values(smallLocation)
        values(smallLocation) = temp
    Next I
    For I = 1 To N
D       Print values(I)
    Next I
```

Figure 10-14: Minimal Selection Sort

In **A** we set the loop control so that the minimal search process is repeated $N - 1$ times. When $I = N$-1 we are comparing the elements at N-1 and N. When this loop is finished both of these elements are in increasing order. Consequently, we do not have to repeat the body of the loop when $I = N$. (There are no elements in locations past location N so the element in location N is the smallest element in the range of N to N!!) B is looking for the smallest element in the range $I + 1, \ldots, N$ as we start the process saying that the smallest element is in location I. We update out estimate of the location of the smallest element by comparing each element in the search range with the best estimate found to that point of the search. When $I = 1$, we compare elements at 2, 3, … , N. When $I = 2$, we compare elements at 3, 4, … , N. As I increases, the number of elements in the search range decreases. The interchange that puts the smallest element in location I is carried out at **C**. In **D** the array has its elements printed out after the sort is completed.

10.10 Using Arrays – Finding a Distribution of Elements

Since arrays are used to store many data values, it is often useful to find out how the data is distributed over the range of all the data. This idea is used to find out how many values fall in each bracket of a decomposition of the range of values. There are several ways we can define the brackets of a decomposition for integer values. In this example we are looking for how many values in an array with elements in the range 0-100 are in each of the following intervals:

Brackets

88–100	80–87	73–79	60–72	0–59
1	2	3	4	5

Table 10-1: Brackets for a Distribution

The brackets are given and can be quite different for different applications and different data sets. One method to find out how many values are in each bracket uses the Multi Case If . The selection code is shown in Figure 10-15 with the bottom value of *Brackets(I)* in elements where I ranges from 1 to 5. The number of elements in a bracket is counted in the array *categories*.

```
For I = 1 To N
If elements (I) >=88 Then
                    categories(1)=categories(1)+1
        ElseIf elements (I) >= 80 Then
                    categories(2)=categories(2)+1
        ElseIf elements (I) >= 73 Then
                    categories(3)=categories(3)+1
        ElseIf elements (I) >= 60 Then
                    categories(4)=categories(4)+1
        Else
                    categories(5)=categories(5)+1
        End If
Next I
```

Figure 10-15: Multi Case If Categorization of Data

The **If .. ElseIf** statement is really quite involved. If a value is found to be less than 88, it will be compared with 80. The interesting aspect to this syntax is that if the comparison with 88 is false, we know that the element is not in the interval 88–100. This means the next condition is true if the element is in the interval 80–87 even though it looks like we are asking if the value is in the range 80–100. Checking on the upper bound for an interval is unnecessary but is implied since the conditions ask about the bottom value in a bracket having eliminated all brackets above the one with this bottom value.

A second alternative used for finding a distribution uses a sequence of compound conditions to determine the categories. The code for this approach is shown in Figure 10-16.

```
For I = 1 To N
        If elements (I) >=88 Then
                categories(1)=categories(1)+1
        End If
        If elements (I) >= 80 And elements (I) <= 87Then
                categories(2)=categories(2)+1
        End If
        If elements (I) >= 73 And elements(I) <=79Then
                categories(3)=categories(3)+1
        End If
        If elements (I) >= 60 And elements(I) <=72Then
                categories(4)=categories(4)+1
        End If
        If elements(I) <= 59 Then
                categories(5)=categories(5)+1
        End If
Next I
```

Figure 10-16: Selection Using Compound Conditions

10.11 Using Arrays – Parallel Arrays

Problems using arrays have focused on computing with a single set of related values. Even with the problem of finding the average of a set of numbers, there is often other information related to each value that must be used in the problem solution. For example, suppose a set of grades is stored in an array named *grades* and the corresponding set of student names is to be stored in an array named *student*. To find the name of the student with the highest grade requires we know which grade belongs to which student. It is certainly possible to have one array store the grades and another array store the names. The issue is that if location *I* in *grades* contains a grade that it should be easy to find the name of the student whose grade it is. The simplest solution is to enter a grade and a name so that the subscript used to identify the location in *grades* is also used to identify the location in *student* where that student's name will be stored. The process of coordinating the use of storage locations in two or more arrays is known as using parallel arrays.

The problem solved in Figure 10-2 can be expanded using parallel arrays to identify not only the grades greater than the average but also the names of the students with a grade greater than the average. See Figure 10-17.

```
Dim student (1 To 50) as String
Dim grade (1 To 50) as Integer
sum = 0 'accumulator
N = Val(InputBox("Enter number of values."))
For I = 1 to N
        student(I) = InputBox("Enter student's name:")
        grade(I) = Val(InputBox("Enter a grade"))
        sum = sum + grade(I)
Next I
ave = sum / N
For I = 1 To N
        If grade(I) > ave Then
           Print student(I),grade(I)
        End If
Next I
```

Figure 10-17: Using Parallel Arrays for Grades and Names

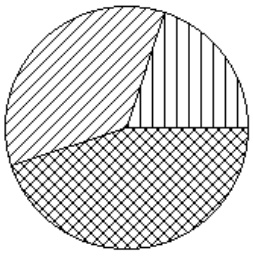
10.12 Using Arrays – Drawing A Pie Chart

The tools we learned in Chapter 9 about drawing capabilities in BASIC can be used in more complex problems when we use arrays. The problem of displaying data using a pie chart is modified here by using an array to keep track of the size of each sector in terms of the percentage of the total area of the circle.

Example 10-1. Write a program that creates a pie chart with three sectors. The user enters the size of two sectors, and the chart is created based on that data.

SOLUTION:

The sector sizes have been input to be 20% and 35%. The third sector will have size 100 – 20 – 35%.



```
    Rem Create a pie chart with three sectors. The size of
    Rem the sectors are based on the user's data.
    Rem INPUT: Percent of circle representing size of some sectors
    Rem OUTPUT: A pie chart with three sectors
    Dim sector (1 To 10) as Double
    Cls
    Rem Get input
A   For I = 1 To 2
        sector(I) = Val(InputBox("Enter sector size:"))
    Next I

    sector(3) = 100 – sector(1) – sector(2)
    Rem Initialize variables
B       pi = 3.14159
        radius = 80
        stopangle = .001
    Rem Draw each sector
C   For I = 1 To 3
        If I = 1 Then
          FillStyle = 3
          FillColor = vbBlue
        ElseIf I = 2 Then
          FillStyle = 5
          FillColor = vbGreen
        Else
          FillStyle = 7
          FillColor = vbYellow
        End If
    Rem Calculate position and size of current sector
D       angle = 2 * pi * sector(I)/100
E       startangle = stopangle
        stopangle = startangle + angle
        If stopangle > 2 * pi Then
        stopangle = .001

    End If
    Rem Draw the sector
F       Circle (160, 100), radius,,-startangle,-stopangle
    Next I
```

COMMENTS: The program begins by allowing the user to input two of the *sector* sizes at **A**. Beginning in **B**, variables that will be used later are initialized. The **For .. Next** loop beginning in **C** draws each sector individually. Within the loop, **D** merely computes the appropriate sector size to draw in radian measure. Beginning in **E**, equations are used to calculate the size and position of the sector currently being drawn. The **Circle** command at **F** actually draws the sector. The radius of the sector is the radius that was initialized at **B**. The color and style of the sector is given by the values assigned to **FillColor** and **FillStyle** in the **If** block beginning at **C**.

On Your Own

1. Write a program that creates a pie chart with four sectors. Let three of the sector sizes be 20%, 25%, and 15%. The fourth sector is the remaining portion of the circle. Use any colors and fill styles.

10.13 Using Arrays – Drawing a Histogram

When values need to be seen in comparison to other values as the sales for a sequence of months or the number of gyzmos sold each month, the chart often used is a histogram. A histogram represents a set of values using as a scale the relative difference in the magnitudes. The output form shown here displays a histogram for values collected over a period of four months denoted by J(anuary), F(ebruary), M(arch), and A(pril). We see the output for this problem in Figure 10-18.

Figure 10-18: A Histogram

The code to draw this chart is given in Example 10-2

Example 10-2. Draw a histogram representing the four values 120, 200, 90, and 250 that represent sales in January, February, March, and April, respectively. Scale the values so that 120 is represented by 24, 200 by 40, 90 by 18, and 250 by 50. Use the labels J, F, M, A for the four months.

SOLUTION:

```
Rem: Draw a histogram with four values
Rem INPUT: Four values to be represented
Rem OUTPUT: Histogram representing the relative size of the input numbers
Dim heightValue (1 To 10) as Integer
Dim label (1 To 10) as String

Line (100, 300) – (100, 100)
Line (100, 300) – (250, 300)
Rem Tic mark on the down axis-each tic represents 4 units
For I = 1 to 20
        Line (98, 300 – 10 * I)-(102, 300 – 10 * I)
Next I
Rem Enter heights and labels
For I = 1 to 4
        heightValue(I) = Val(InputBox("Enter  height"))
        label(I) = InputBox("Box label")
Next I
Rem Draw and label rectangles
For I = 1 to 4
        Line (120 + 30 * (I – 1), 300) – (140 + 30 * (I – 1), 300 – 3 * heightValue(I)), ,B
        CurrentX = 125 + 30 * (I – 1) : CurrentY = 312
        Print label(I)
Next I
```

10.14 Putting It All Together

1. Input five numbers into an array using a **For** .. **Next** loop. Then using another loop, print the values in the array.
2. Read ten values into an array using a **For** .. **Next** loop. In a second **For** .. **Next** loop add these numbers and finally print the average of the ten numbers.
3. Read *N* numbers from a file and store them in an array. Count the number of values that are bigger than the average value.
4. Count the number of values that are less than the average minus 8, the number between (inclusive) the average plus 8 and the average minus 8, and the number greater than the average plus 8. Read the values from a file into an array. Use variables ct1, ct2, and ct3 to store the counts. Print the number of values in each category.
5. Repeat problem 4 but store the counts in an array *ave* with three storage locations. Print the values in ave using a **For** .. **Next** loop. Print the values on a single line.
6. Modify the code in Figure 10-17 so that there are three parallel arrays. The first array holds the size of the sectors. The second array at the same index holds the **FillStyle** value for this sector. The third array at the same index holds the **FillColor** value for this sector. Run the program for a five sector pie chart.

7. Modify the code in Example 10-2 so that up to seven bars can be included in the histogram. Use parallel arrays for the bar heights and the bar labels. Use files for the input into the arrays. Experimental data is often "smoothed" before it is used. The "smoothing" process replaces each value by the average of the three points that precede it. For example, for a data file with N elements in an array *rawData* a new array *smoothData* is defined and given values as follows:

```
smoothData(1) = 0
smoothData(2) = 0
smoothData(3) = 9
smoothData(4) = (rawData(1) + rawData(2) + rawData(3)) / 3
```

   In general,for 4 < I < N define

```
smoothData(I) = (rawData(I) + rawData(I – 1) + rawData(I – 2) / 3.
```

   Write a program that smoothes data in a file with 25 random numbers in the range 0-18. Output the results in a table with three columns with titles I, rawData(I), smoothData(I).

8. Experimental data is often "smoothed" before it is used. The "smoothing" process replaces each value by the weighted average of the three points that precede it. For example, for a data file with N elements in an array *rawData* a new array *smoothData* is defined and given values as follows:

```
smoothData(1) = 0
smoothData(2) = 0
smoothData(3) = 9
smoothData(4) = (rawData(1) + rawData(2) * 2 + rawData(3) *3) / 3
```

   In general, for 4 < I < N define

```
smoothData(I) = (rawData(I – 1) + rawData(I – 2) * 2 + rawData(I – 3) * 3) / 3.
```

   Write a program that smoothes data in a file with 25 random numbers in the range 0-18. Output the results in a table with three columns with titles I, rawData(I), smoothData(I).

Index

Endnotes

1. http://en.wikipedia.org/wiki/Teletype_Model_33