So You Want to Learn to Program?

Third Edition

James M. Reneau, Ph.D.

Associate Professor

Shawnee State University

Portsmouth Ohio USA

http://www.basicbook.org

James M. Reneau

P.O. Box 278

Russell, Kentucky 41169-2078 USA

For BASIC-256 Version 2.0.0.0 or later

So You Want to Learn to Program?

James M. Reneau, Ph.D. - jim@renejm.com

P.O. Box 278 – Russell KY 41169-0278 USA

Create Space Print ISBN: XXXXX

Revision Date: 2019-06-29

Table of Contents

Index of Programs

Index of Illustrations

Acknowledgments:

A big thanks go to all the people who have worked on the BASIC-256 project, at Sourceforge. Most especially, Ian Larsen (aka: DrBlast) for creating the BASIC-256 computer language and his original vision.

I also feel the need to thank the Sumer 2010 programming kids at the Russell Middle School and Julia Moore. Also a shout to my peeps Sergey Lupin, Joel Kahn, Ugly Mike, Florian Opera, and all who have contributed to the BASIC-256 project.

Dedications:

To my wife Nancy and my daughter Anna.

Credits:

Some public domain clip art from http://www.openclipart.com.

Preface

The first edition of this book was created as an introduction to programming in the BASIC language for middle to high school students who wanted to create code on their own. Over the last couple of years the text has evolved to be used in secondary and post-secondary education.

This second edition keeps most of the material in the first edition and includes the modernization of BASIC-256 to include Subroutines, Functions, and better error handling. In addition to updating the language and cleaning up the text and programs, exercises have been added to the end of each chapter to reinforce the techniques discussed and to give the readers/students an additional challenge.

The third edition is updated to include new features of the BASIC-256 language. These include: the new dynamic nature of arrays and variables being of a variant type.

This book chapters can be structured for use in a variety of ways:

1. a 9 or 18 week introduction to programming
    - chapters 1, 2, 3, 4[*], 5, 6, 7, 8, and 9 for the first 9 week term
    - chapters 10, 11, 12, 13[*], 14, 15, 16, 17, 18 and 19[*] for the second 9 week term
2. a brief introduction to the concepts of programming
    - chapters 1, 3[**], 4[*] , 5, 6, 7, 9, and 14
3. an introduction to data structures for non-programmers
    - chapters 1, 3[**], 4[*] , 5, 6, 7, 9, 14, 15[*], 16[*], 17 and 20
4. a brief programming project for a database system course
    - chapters 1, 3[**], 4[*] , 5, 6, 7, 9, 14, 15[*], 16[*], 19 and 21
5. and a brief programming project for a networking course.
    - chapters 1, 3[**], 4[*] , 5, 6, 7, 9, 14, 15[*], 16[*], 19 and 23

The most important part of this book is the ability to mix and re-mix the material to fit your very specific needs.

I wish you nothing but success.
-Jim

---

* Denotes Optional Chapter
** Numeric Variables Section Only

Chapter 1: Meeting BASIC-256 – Say Hello.

This chapter will introduce the BASIC-256 environment using the **print** and **say** statements. You will see the difference between commands you send to the computer, strings of text, and numbers that will be used by the program. We will also explore simple mathematics to show off just how talented your computer is. Lastly you will learn what a syntax-error is and how to fix them.

The BASIC-256 Window:

The BASIC-256 window also known as an Integrated Development Environment (IDE) is divided into five sections: the Menu Bar, Tool Bar, Program Area, Text Output Area, and Graphics Output Area (see Illustration 1: The BASIC-256 Integrated Development Environment (IDE) below).

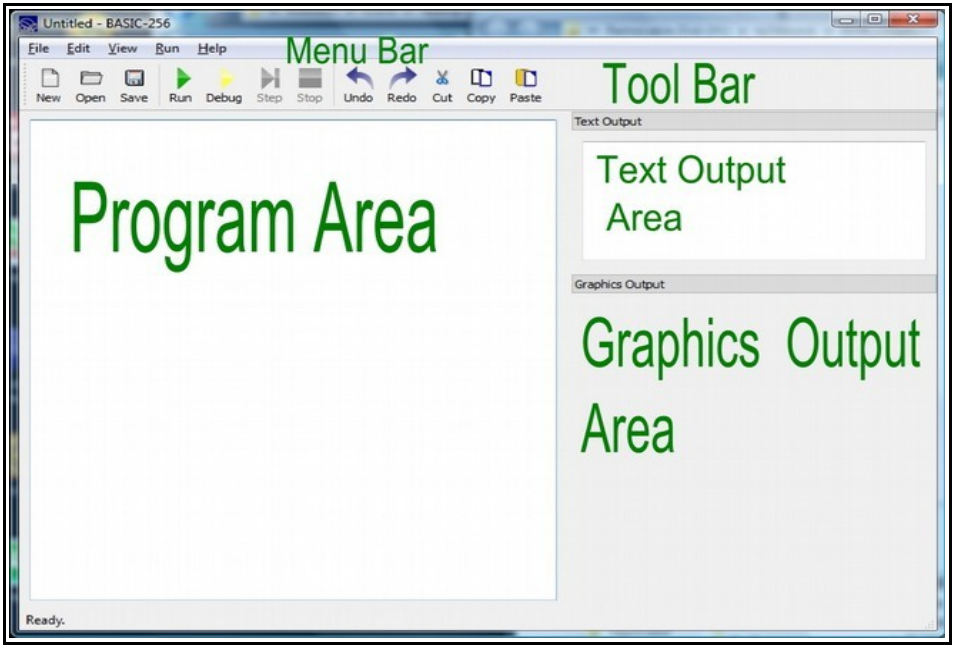


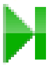
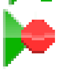
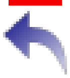
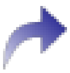
Illustration 1: The BASIC-256 Integrated Development Environment (IDE)

Menu Bar:

The menu bar contains several different drop down menus. These menus include: "File", "Edit", "View", "Run", and "About". The "File" menu allows you to save, reload saved programs, print and exit. The "Edit" menu allows you to cut, copy and paste text and images from the program, text output, and graphics output areas. The "View" menu will allow you to show or hide various parts of the BASIC-256 window. The "Run" menu will allow you to execute and debug your programs. The "About" menu option will display a pop-up dialog with information about BASIC-256 and the version you are using.

Tool Bar:

The menu options that you will use the most are also available on the tool bar.

-  New – Start a new program
-  Open – Open a saved program
-  Save – Save the current program to the computer's hard disk drive or your USB pen drive
-  Run – Execute the currently displayed program
- Debug – Start executing program one line at a time
-  Step – When debugging – go to next line
-  Run to Break Point – When debugging – run to the next line marked as a break point
-  Stop – Quit executing the current program
-  Undo – Undo last change to the program.
-  Redo – Redo last change that was undone.

- Cut – Move highlighted program text to the clipboard
- Copy – Place a copy of the highlighted program text on the clipboard
- Paste – Insert text from the clipboard into program at current insertion point

Program Area:

Programs are made up of instructions to tell the computer exactly what to do and how to do it. You will type your programs, modify and fix your code, and load saved programs into this area of the screen.

Text Output Area:

This area will display the output of your programs. This may include words and numbers. If the program needs to ask you a question, the question (and what you type) will be displayed here.

Graphics Output Area:

BASIC-256 is a graphical language (as you will see). Pictures, shapes, and graphics you will create will be displayed here.

Your first program – The saystatement:

Let's actually write a computer program. Let us see if BASIC-256 will say hello to us. In the Program Area type the following one-line program (you will see the line number in BASIC256 but you should not type it):

```
say "hello"
```

Program 1: Say Hello

Once you have this program typed in, use the mouse, and click on "Run" in the tool bar. 

Did BASIC-256 say hello to you through the computer's speakers?


```
say expression
```

The **say** statement is used to make BASIC-256 read an expression aloud, to the computer's speakers.

BASIC-256 treats letters, numbers, and punctuation that are inside a set of quotation marks as a block. This block is called a string.

"letters, numbers 9988, and symbols &%"
'another string with a "quote" inside.'

A string may begin with either a single quote mark (') or a double quote mark (") and ends the same as it began. A string surrounded with single quotes may contain double quotes and a string surrounded by double quotes may contain single quotes.

"Run" on the tool bar - or - "Run" then "Run" on the menu

You must tell BASIC-256 when you want it to start executing a program. It doesn't automatically know when you are done typing your programming code in. You do this by clicking on the "Run" icon on the tool bar or by clicking on "Run" from the menu bar then selecting "Run" from the drop down menu.

To clear out the program you are working on and completely start a new program we use the "New" button on the tool bar. The new button will display the following dialog box:

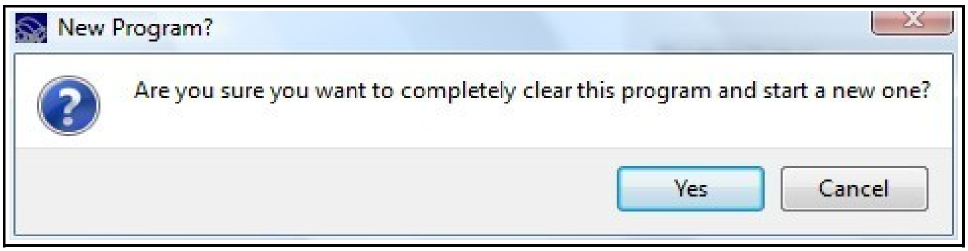


Illustration 2: BASIC-256 - New Dialog

If you are fine with clearing your program from the screen then click on the Yes "Yes" button. If you accidentally hit "New" and do not want to start a new program then click on the Cancel "Cancel" button.

New Concept	"New" on the tool bar - or - "File" then "New" on the menu The "New" command tells BASIC-256 that you want to clear the current statements from the program area and start a totally new program. If you have not saved your program to the computer (Chapter 2) then you will lose all changes you have made to the program.

Your Second Program – Saying Something Else

You can also have the **say** statement speak out numbers. Try the following program:

```
say 123456789
```

Program 2: Say a Number

Once you have this program typed in, use the mouse, and click on "Run" in the tool bar.

Did BASIC-256 say what you were expecting?

New Concept	numbers BASIC-256 allows you to enter numbers in decimal format. Do not use commas when you are entering large numbers. If you need a number less than zero just place the negative sign before the number. Examples include: 1.56, 23456, -6.45 and .5

BASIC-256 is really good with numbers – Simple Arithmetic:

The brain of the computer (called the Central Processing Unit or CPU for short) works exclusively with numbers. Everything it does from graphics, sound, and all the rest is done by manipulating numbers.

The four basic operations of addition, subtraction, multiplication, and division are carried out using the operators show in Table 1.

Operator	Operation	Example
+	Addition	`expression1 + expression2`
-	Subtraction	`expression1 - expression2`
*	Multiplication	`expression1 * expression2`
/	Division	`expression1 / expression2`

Table 1: Basic Mathematical Operators

Try this program and listen to the talking super calculator.

```
say 12 * (2 + 10)
```

Program 3: Say the Answer

The computer should have said "144" to you.

```
say 5 / 2
```

Program 4: Say another Answer

Did the computer say "2.5"?


```
+
-
*
/
()
```

The four basic mathematical operations: addition (+), subtraction (-), division (/), and multiplication(*) work with numbers to perform calculations. A numeric value is required on both sides of these operators. You may also use parenthesis to group operations together.

Examples include: 1 + 1, 5 * 7, 3.14 * 6 + 2, (1 + 2) * 3 and 5 - 5

Concatenation:

Concatenation is the operation that joins two strings together to make a longer string. If the strings “abcd” and “xyz” and concatenated together the string “abcdxyz” would be the result. This operation is called concatenation, or "cat" for short.

BASIC-256 has three different operators that will concatenate strings, but they perform differently when the expressions are numbers. The **;** operator will convert expressions to strings and always concatenate, the **+** operator will numerically add two numbers but concatenate if either are strings, and the **&** operator will perform a 'bit-wise and' if both are numbers but will otherwise concatenate.

Let's try it out:

```
say "Hello " ; "Mary."
```

Program 5: Say Hello to Mary

The computer should have said hello to Mary.

Try another.

```
say 1 ; " more time"
```

Program 6: Say it One More Time

In the last example concatenation was performed with a number and a string. The number was first converted to a string "1" and then BASIC-256 was able to concatenate.

```
1 Say 1 + 2
2 say '1' + 2
3 say 1 ; 2
```

The computer should have said "three", "twelve", and "twelve". In the first line, the plus operator adds the numbers 1 and two. In line 2, the plus operator concatenates the string 1 to the string 2 (the number is converted). In the last line the semicolon operator converted both numbers to strings and concatenates.


```
; (concatenate)
+ (concatenate)
& (concatenate)
```

The semicolon (**;**) is used to tell the computer to concatenate (join) strings together. If one or both operands are numeric they will be changed to strings before concatenation.

The **+** and **&** operators perform concatenation if either or both expressions are strings. If both are numbers then they perform other actions.

The text output area - The print statement:

Programs that use the Text to Speech (TTS) **say** statement can be very useful and fun but it is also often necessary to write information (strings and numbers) to the screen so that the output can be read. The **print** statement does just that. In the Program Area type the following two-line program:

```
print "hello"
print "there"
```

Program 7: Print Hello There

Once you have this program typed in, use the mouse, and click on  "Run" in the tool bar. The text output area should now show "hello" on the first line and "there" on the second line.


```
print expression
print expression;
```

The **print** statement is used to display text and numbers on the text output area of the BASIC-256 window.

The print statement, by default, advances the text area so that the next print is on the next line. If you place a ; (semicolon) on the end of the expression being printed, it will suppress the line advance so that the next print will be on the same line.

The **print** statement, by default, advances the text area so that the next **print** is on the next line. If you place a ; (semicolon) on the end of the expressionbeing printed, it will suppress the line advance so that the next **print** will be on the same line.

```
cls
print "Hello ";
print "there, ";
print "my friend."
```

Program 8: Many Prints One Line


```
cls
```

The clsstatement clears all the old displayed information from the text output area.

What is a "Syntax error":

Programmers are human and occasionally make mistakes. "Syntax errors" are one of the types of errors that we may encounter. A "Syntax error" is generated by BASIC-256 when it does not understand the program you have typed in. Usually syntax errors are caused by misspellings, missing commas, incorrect spaces, unclosed quotations, or unbalanced parenthesis. BASIC-256 will tell you what line your error is on and will even attempt to tell you where on the line the error is.

Exercises:

Word Search

```
z a h d g p b a n n q m c
j g j r o i q l o c q o x
r u n t u u n i l c n s z
v w s y o b s s k c y l l
e n a t i s s p a n p a x
r s e p e q r t t f r p t
r b k r y o e a r m m r a
o r p i g n x d o i f n i
r x n r a y t i h l n a f
e g a t m d w n v e d g i
t m i a c v c e i j f d n
b o t c c a u s o r c i s
n a m z i z i g n c p r u
```

cls, concatenation, error, expression, print, program, quote, run, say, stop, string, syntax

Problems

1. Write a one line program to say the tongue twister 'Peter Piper picked a peck of pickled peppers."

2. Add a second line to Problem 1 to also display that sentence on the screen.

3. Use the computer as a talking calculator to solve the following problem and to say the answer: Bob has 5 pieces of candy and Jim has 9. If they were to share the candy evenly between them, how many would they each have (average).

4. Use the computer as a talking calculator to solve the following problem and to say the answer: You want 5 model cars that each cost $1.25 and one model boat that costs $3.50. How much

	money to you need to make these purchases. 5. Write a one line program to say “one plus two equals three” without using the word three or the number 3.

Chapter 2: Drawing Basic Shapes.

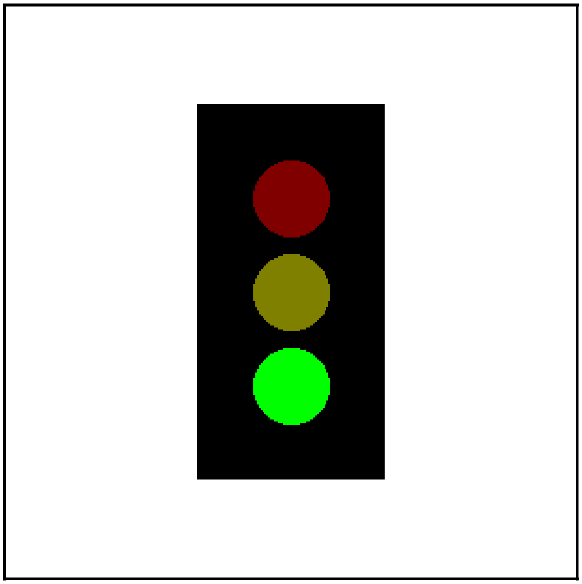
In this chapter we will be getting graphical. You will learn how to draw rectangles, circles, lines and points of various colors. These programs will get more and more complex, so you will also learn how to save your programs to long term storage and how to load them back in so you can run them again or change them.

Drawing Rectangles and Circles:

Let's start the graphics off by writing a graphical program that will draw a traffic light, specifically a green light.

```
# traffic_light.kbs
# Show a traffic light and say a message.

clg

color black
rect 100,50,100,200

color darkred
circle 150,100,20

color darkyellow
circle 150,150,20

color green
circle 150,200,20

say "Green light. You may go."
```

Program 9: Traffic Light



Sample Output 9: Traffic Light

Let's go line by line through the program above. The first and second lines are called remark or comment statements. A remark is a place for the programmer to place comments in their computer code that are ignored by the BASIC-256. They are a good place to describe what complex blocks of code is doing, the program's name, why we wrote a program, or who the programmer was.


```
#
rem
```

The **#** and **rem** statements are called remarks. A remark statement allows the programmer to put comments about the code they are working on into the program. The computer sees the **#** or **rem** statement and will ignore all of the rest of the text on the line.

On line four you see the **clg** statement. It is much like the **cls** statement from Chapter 1, except that the **clg** statement will clear the graphic output area of the screen.

New Concept

```
clg
clg color_name
clg rgb( red, green, blue )
```

The **clg** statement erases the graphics output area so that we have a clean place to do our drawings.

You may optionally define a color after the **clg** statement and it will set the entire graphics output window to that color.

Lines six, nine, twelve, and fifteen contain the simple form of the **color** statement. It tells BASIC-256 what color to use for the next drawing action. You may define colors either by using one of the eighteen standard color names or you may create one of over 16 million different colors by mixing the primary colors of light (red, green, and blue) together.

When you are using the numeric method to define your custom color be sure to limit the values from 0 to 255. Zero (0) represents no light of that component color and 255 means to shine the maximum. Bright white is represented by 255, 255, 255 (all colors of light) where black is represented by 0, 0, 0 (no colors at all). This numeric representation is known as the RGB triplet. Illustration 3 shows the named colors and their RGB values.


```
color color_name
color rgb( red, green, blue )
```

color can also be spelled **colour**.

The **color** statement allows you to set the color that will be drawn next. You may follow the **color** statement with a color name (black, white, red, darkred, green, darkgreen, blue, darkblue, cyan, darkcyan, purple, darkpurple, yellow, darkyellow, orange, darkorange, grey/gray, darkgrey/darkgray). You may also specify over 16 million different colors using the RGB() function by specifying how much red, blue, and green should be used.

Color Name and RGB Values	Color Name and RGB Values
black (0,0,0)	white (255,255,255)
red (255,0,0)	darkred (128,0,0)
Green (0,255,0)	darkgreen (0,128,0)
blue (0,0,255)	darkblue (0,0,128)
cyan (0,255,255)	darkcyan (0,128,128)
purple (255,0,255)	darkpurple (128,0,128)
yellow (255,255,0)	darkyellow (128,128,0)
orange (255,102,0)	darkorange (170,51,0)
grey/gray (164,164,164)	darkgrey/darkgray (128,128,128)

Illustration 3: Color Names

The graphics display area, by default is 300 pixels wide (x) by 300 pixels high (y). A pixel is the smallest dot that can be displayed on your computer monitor. The top left corner is the origin (0,0) and the bottom right is (299,299). Each pixel can be represented by two numbers, the first (x) is how far over it is and the second (y) represents how far down. This way of marking points is known as the Cartesian Coordinate System to mathematicians.

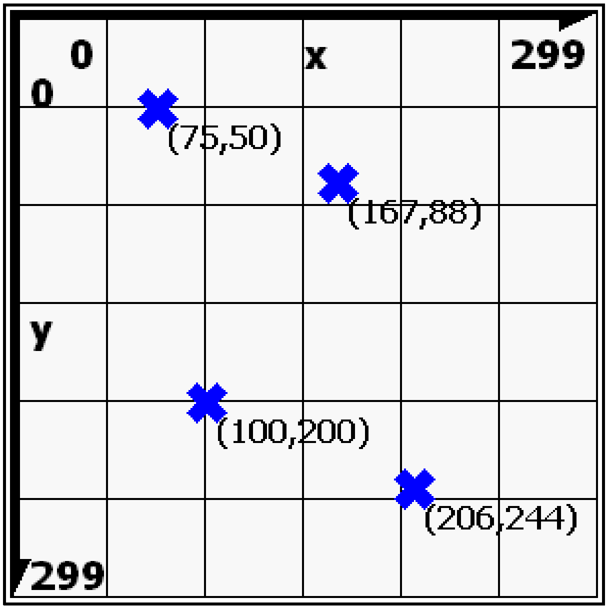


Illustration 4: The Cartesian Coordinate System of the Graphics Output Area

Explore

You can display grid lines on the Graphics Output Area of the screen by checking the "Graphics Window Grid Lines" option on the View menu.

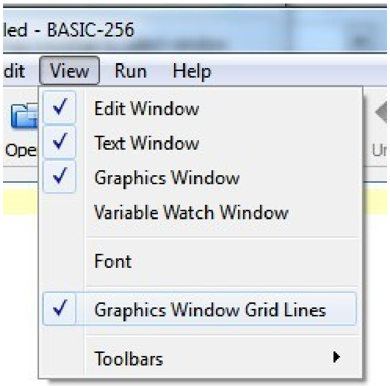


Illustration 5: Grid Lines Menu Option

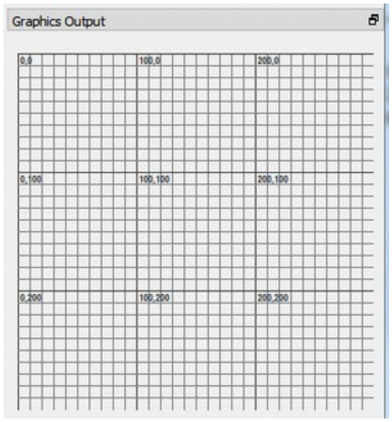


Illustration 6: Graphics Output Grid Lines

The next statement we will discuss (line 7) is **rect**. It is used to draw rectangles on the screen. It takes four numbers separated by commas; (1) how far over the left side of the rectangle is from the left edge of the graphics area, (2) how far down the top edge is, (3) how wide and (4) how tall. All four numbers are expressed in pixels (the size of the smallest dot that can be displayed).

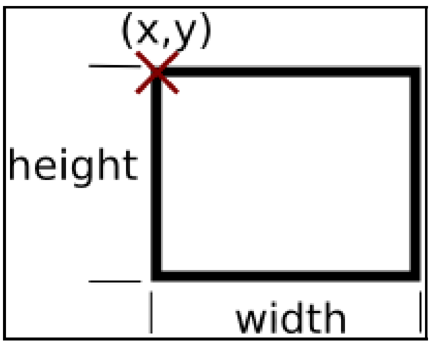


Illustration 7: Rectangle

You can see that the rectangle in the program starts at the point (100,50), is 100 pixels wide and 200 pixels tall.

New Concept	***rect x, y, width, height*** The **rect** statement uses the current drawing color and places a rectangle on the graphics output window. The top left corner of the rectangle is specified by the first two numbers and the width and height is specified by the other two arguments.

Lines 10, 13 and 16 of Program 9 introduce the **circle** statement to draw a circle. It takes three numeric arguments, the first two represent the Cartesian coordinates for the center of the circle and the third the radius in pixels.

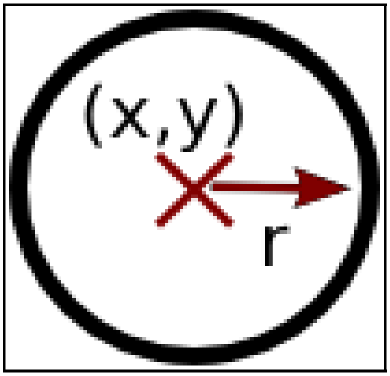


Illustration 8: Circle


```
circle x, y, radius
```

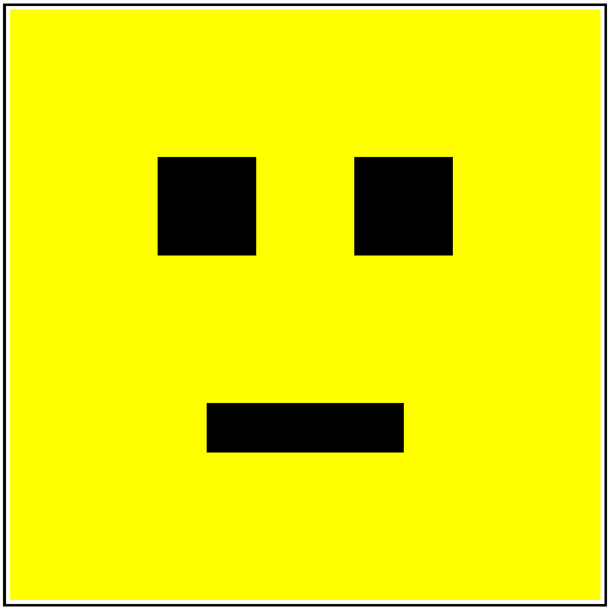
The **circle** statement uses the current drawing color and draws a filled circle with its center at (x, y) with the specified radius.

Some Other Programs Using Circles and Rectangles

Here are a couple of sample programs that use the new statements **clg**, **color**, **rect** and **circle**. Type the programs in and modify them. Make them a frowning face, alien face, or look like somebody you know.

```
# rectanglesmile.kbs

# make the screen yellow
clg yellow

# draw the mouth
color black
rect 100,200,100,25

# put on the eyes
color black
rect 75,75,50,50
rect 175,75,50,50

say "Hello."
```

Program 10: Face with Rectangles



Sample Output 10: Face with Rectangles

```
# circlesmile.kbs

# clear the screen
clg white

# draw the face
color yellow
circle 150,150,150

# draw the mouth by drawing a big black circle
# and then covering up the to part to leave
# a smile
color black
circle 150,200,70
color yellow
circle 150,150,70

```

```
# draw the eyes
color black
circle 100,100,30
circle 200,100,30
```

Program 11: Smiling Face with Circles

Sample Output 11: Smiling Face with Circles

Saving Your Program and Loading it Back:

Now that the programs are getting more complex, you may want to save them so that you can load them back in the future.

You may store a program by using the Save button  on the tool bar or Save option on the File menu. A dialog will display asking you for a file name, if it is a new program, or will save the changes you have made (replacing the old file).

If you do not want to replace the old version of the program and you want to store it using a new name you may use the Save As option on the File menu to save a copy with a different name.

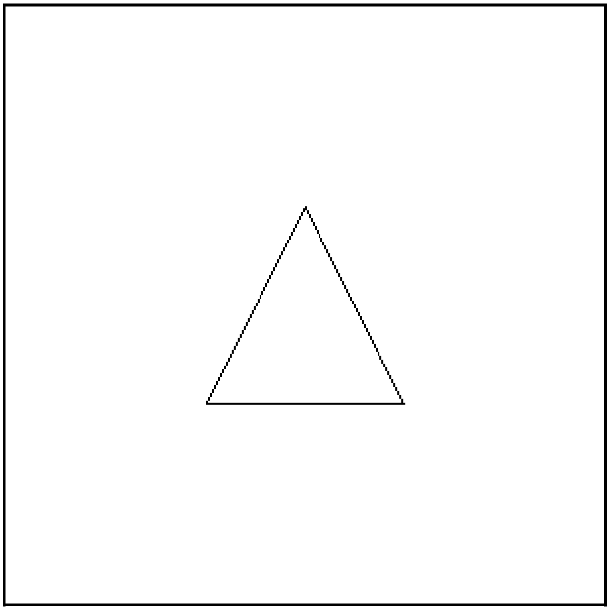
To load a previously saved program you would use the Open button  on the tool bar or the Open option on the File menu.

Drawing with Lines:

The next drawing statement is **line**. It will draw a line one pixel wide, of the current color, from one point to another point. Program 12 shows an example of how to use the **line** statement.

```
# triangle.kbs - draw a triangle

clg

color black
line 150, 100, 100, 200
line 100, 200, 200, 200
line 200, 200, 150, 100
```

Program 12: Draw a Triangle



Sample Output 12: Draw a Triangle

`line start_x, start_y, finish_x, finish_y`

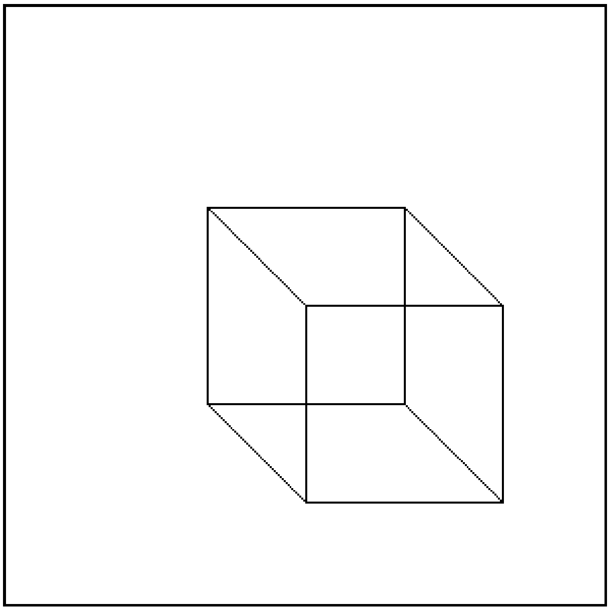
Draw a line one pixel wide from the starting point to the ending point, using the current color.

The next program is a sample of what you can do with many lines. It draws a cube on the screen.

```
# cube.kbs - draw a cube

clg
color black
```

```

# draw back square
line 150, 150, 150, 250
line 150, 250, 250, 250
line 250, 250, 250, 150
line 250, 150, 150, 150

# draw front square
line 100, 100, 100, 200
line 100, 200, 200, 200
line 200, 200, 200, 100
line 200, 100, 100, 100

# connect the corners
line 100, 100, 150, 150
line 100, 200, 150, 250
line 200, 200, 250, 250
```

Program 13: Draw a Cube



Sample Output 13: Draw a Cube

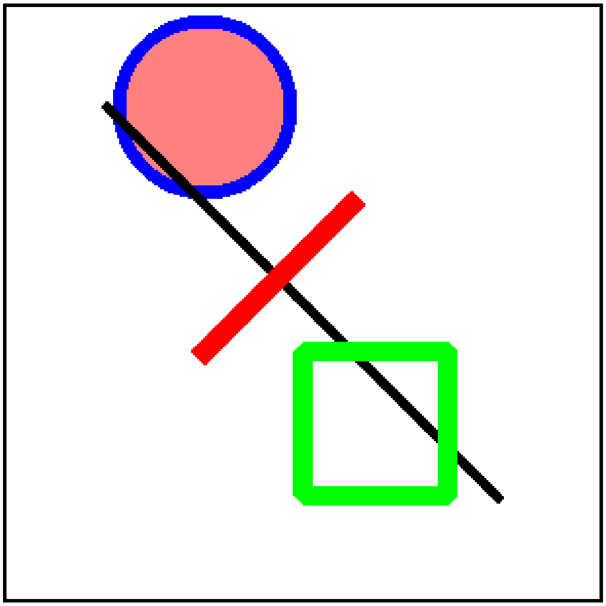
Setting Line Width and Drawing Shape Borders:

By default the width of a line drawn in BASIC256 is one pixel (dot) wide. The **penwidth** statement can be used to change the way lines (and borders around shapes) are drawn.

The following program will illustrate the **penwidth** statement, a more complex use of the **color** statement and an example of the special color **clear**.

```
# shapeoutline.kbs
# draw shapes with an outline

clg

# darw a pink circle with blue background
penwidth 7
color blue, rgb(255,128,128)
circle 100,50,44

# draw a thick black line
color black
penwidth 5
line 50,50,250,250

# draw another thick red line
color red
penwidth 10
line 175,100,100,175

# draw a green square that is not filled
color green, clear
penwidth 10
rect 150,175,75,75
```

Program 14: Penwidth and Shape Outline



Sample Output 14: Penwidth and Shape Outline

`penwidth n`

Changes the width of the drawing pen. The pen represents the width of a line being drawn and also the width of the outline of a shape.

New Concept

`color pen_color, fill_color`

Earlier in this chapter we saw the color statement with a single color. When only a single color is specified then both the pen and the fill color are set to the same value. You may define the pen and fill colors to be different colors by using the color statement with two colors.

New Concept

`clear`

The special color **clear** may be used in the color statement to tell BASIC256 to only draw the border of a shape. Just set the fill color to clear.

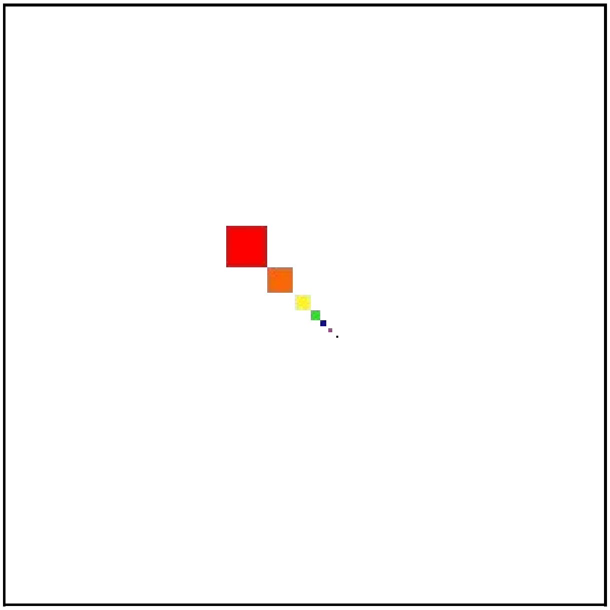
Setting Individual Points on the Screen:

The last graphics statement covered in this chapter is **plot**. The **plot** statement sets a single pixel (dot) on the screen. For most of us these are so small, they are hard to see. Later we will write programs that will draw groups of pixels to make very detailed images.

```
# pointplot.kbs - use plot to draw points

clg

color red
penwidth 21
plot 120,120

```

```
color orange
penwidth 13
plot 137,137

color yellow
penwidth 8
plot 149,149

color green
penwidth 5
plot 155,155

color blue
penwidth 3
plot 159,159

color purple
penwidth 2
plot 163,163

color black
penwidth 1
plot 166,166
```

Program 15: Use Plot to Draw Points



Sample Output 15: Use Plot to Draw Points

New Concept

```
plot x, y
```

Draws a point on the screen in the current pen color with the current pen width.

Big Program

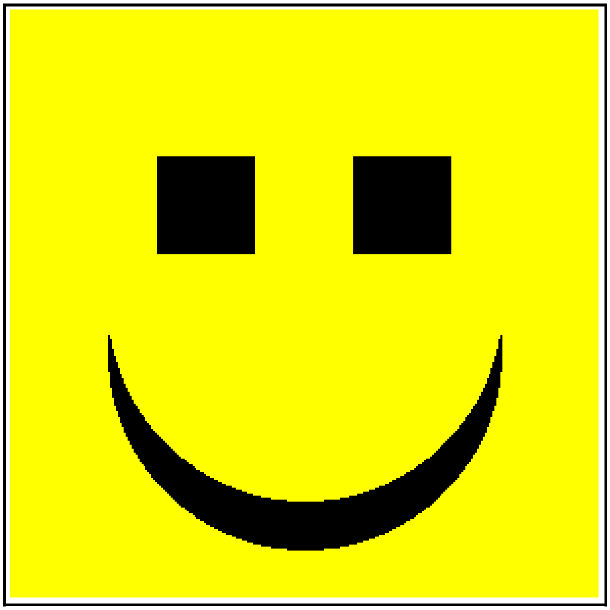
At the end of each chapter there will be one or more big programs for you to look at, type in, and experiment with. These programs will contain only topics that we have covered so far in the book.

This "Big Program" takes the idea of a face and makes it talk. Before the program will say each word the lower half of the face is redrawn with a different mouth shape. This creates a rough animation and makes the face more fun.

```
# talkingface.kbs
color yellow
rect 0,0,300,300
color black
rect 75,75,50,50
rect 175,75,50,50

#erase old mouth
color yellow
rect 0,150,300,150
# draw new mouth
color black
rect 125,175,50,100
# say word
say "i"

color yellow
rect 0,150,300,150
color black
rect 100,200,100,50
say "am"

color yellow
rect 0,150,300,150
color black
rect 125,175,50,100
say "glad"

```

```
color yellow
rect 0,150,300,150
color black
rect 125,200,50,50
say "you"

color yellow
rect 0,150,300,150
color black
rect 100,200,100,50
say "are"

color yellow
rect 0,150,300,150
color black
rect 125,200,50,50
say "my"

# draw whole new face with round smile.
color yellow
rect 0,0,300,300
color black
circle 150,175,100
color yellow
circle 150,150,100
color black
rect 75,75,50,50
rect 175,75,50,50
say "friend"
```

Program 16: Big Program - Talking Face



Sample Output 16: Big Program - Talking Face

Exercises:

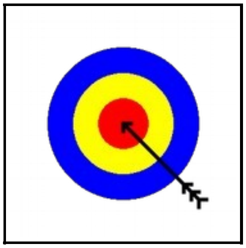
Word Search	r e t a n i d r o o c e e a r a e l c r u m m e l c r i c e s s r a c k v c e c c u y o r y j l n t i i t p l k a g t a h d h w l o q n e n p a g i q o c y r g a r i d p j t e c l r e e t s a v e h e g p h h u e n i l d j r x p e n w i d t h center, circle, clear, clg, color, coordinate, cyan, graphics, height, line, penwidth, plot, radius, rectangle, remark, save, width

Problems

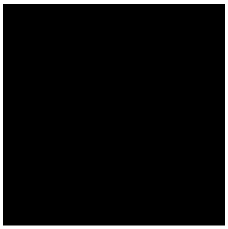
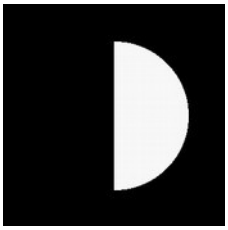
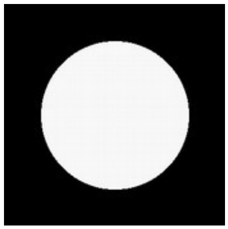
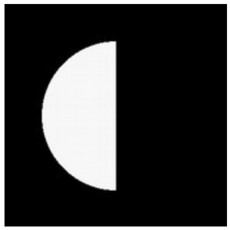
1. Type in the code for Program 11: Smiling Face with Circles (on page 24) and modify it to display Mr. Yuck. You may need to use the **penwidth** statement to make the lines you draw thicker.

2. Write a program to draw a square and then say "square". Clear the graphics screen, draw a circle, and say "circle". Then clear the graphics screen draw several lines (in any pattern you would like) and say "lines".

3. Use colors, lines, and circles to draw an archery target with an arrow in the center. Once the arrow is drawn make the computer say “Bullseye!”.



4. Write a program that draws each of the quarters of the moon (new moon, first quarter, full moon, and third quarter) and speaks the name for the quarter. Hint: Draw the moon as a circle and then draw a rectangle over the part you do not want.

			
New Moon	First Quarter	Full Moon	Third Quarter

Chapter 3 – Variables

This chapter is a new chapter in this edition that will introduce you to the concept and basic use of a variable.

What is a Variable

In computer program a variable is "a quantity or function that may assume any given value or set of values."[1] To describe it another way, we can think of a variable as name for a reserved location in the computer's temporary memory. We may store, change, and retrieve values from this location as our program runs by using the variable name.

In BASIC-256 a variable may be used to store integers (whole numbers), decimal numbers, and strings.

`Variable`

A variable allows you to assign a name to a block of storage in the computer's short-term memory. You may store, change and retrieve values from these variables in your program.

A variable's name must begin with a letter and may contain letters, numbers, and dollar signs. Variable names are case-sensitive and you may not use words reserved by the BASIC-256 language when naming your variables (see Appendix I).

Examples of valid variable names include: a, b6, reader, x, f$4, and zoo.

1 http://dictionary.reference.com/browse/variable

Variable names are case-sensitive. This means that an upper case variable and a lowercase variable with the same letters do not represent the same location in the computer's memory.

Assigning Values to Variables

In this introduction we will use the optional **let** statement to assign values to variables. **Let** calculates the expression on the right sign of the equals sign and then assign that new value to the variable on the left-hand side.

```
1   # letsimple.kbs - use variables to store numbers
2
3   let numerator = 30
4   let denominator = 4
5   let result = numerator / denominator
6
7   print numerator + " / " + denominator + " is " + result
8
9   let result = result * 2
10
11  print "result doubled is " + result
```

Program 17: Use Variables to Store Numbers

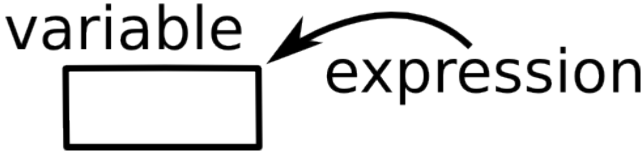
```
30 / 4 is 7.5
result doubled is 15.0
```

Sample Output: 17: Use Variables to Store Numbers

The program above uses three variables. On line three it stores the value 30 into the location named "numerator". Line four stores the value 4 in the variable "denominator". Line five takes the value from "numerator" divides it by the value in the "denominator" variable and stores the value in the variable named "result". Another thing to watch is on line nine, you can see the statement **let result = result * 2** takes the value in result multiplies it by two and then save the value back into the variable result.


```
let variable = expression
variable = expression
```

The let statement will calculate an expression (if necessary) on the right hand side of the equals sign and saves the value into the variable on the left. We call this process assignment or assigning a variable.



The actual **let** statement is optional. You can just assign a variable using the equal sign.

In the first example you saw whole numbers and floating-point numbers stored into variables. In the next example you will see that a variable may contain a string value, just as easily.

```
# letstring.kbs = assign a variable a string

let word = "hello"
let rhyme = "yellow"

stuff = word + " and " + rhyme + " are words that rhyme."
```

```
print stuff
say stuff
```

Program 18: Use Variables to Store Strings

```
hello and yellow are words that rhyme.
```

Sample Output 18: Use Variables to Store Strings

Variable Assignment Shortcuts

Another thing you will learn about computer programming is that there are often more than one way to do a task. BASIC-256 and most computer programming languages allow for a shortcut form of addition and subtraction when working with a variable. In the programs of future chapters you will see these shortcuts.

Shortcut Assignment	Description
`variable += expression`	Add expression to a variable
`variable -= expression`	Subtract expression from a variable
`variable++`	Add one to a variable using old value
`variable--`	Subtract one from a variable using old value
`++variable`	Add one to a variable using new value
`--variable`	Subtract one from a variable using new value

Table 2: Shortcut Variable Assignment

```
a = 9
a += 10
print a ## print 19 - 9 + 10
```

```
a++ ## a = a + 1 (20)
print a

print a++ ## print a then add 1 (20)

print a ## 21

print ++a ## add 1 then print a
```

Program 19: Variable Shortcuts

```
19
20
20
21
22
```

Sample Output 19: Variable Shortcuts

Variable and Data Types

It has been mentioned in prior sections that BASIC-256 understands numbers and strings. Actually, it has four standard types of values: 1) unassigned, 2) integers, 3) floating-point numbers, and 4) strings. We call these data types. In most programming languages there are many more types, but just about all languages have these.

Unassigned

If you attempt to use a variable before it has been assigned a value, it will cause an error or warning to be displayed and the value of "" will be returned to your program.

```
print '/' + x + '/'
```

Program 20: Unassigned Variable

```
WARNING on line 1: Variable has not been assigned a value.
//
```

Sample Output 20: Unassigned Variable

In the Preferences settings screen you may choose the "Runtime handling of unassigned variables" option. It has three settings: 1) Ignore – return a value of "" and do not print a warning, 2) Warn – return the value of "" and display a warning message, or 3) Error – Display an error and stop the program.

Integers

An integer is "a whole number (not a fractional number) that can be positive, negative, or zero"[2]. We use integers to count things or to hold exact values. On most computers integers have a range from –2,147,483,648 to 2,147,483,647. The range is limited because the number is stored in 32 bits (4 bytes) of the computer's memory.

Floating-Point Numbers

Numbers with decimal points are also allowed in BASIC-256 but they are stored in the computer's memory as floating-point numbers. Floating-point is "a mathematical notation in which a number is represented by an integer or a decimal fraction multiplied by a power of the number base indicated by an exponent"[3] Using this method of storage we can typically represent any number from 1.7×10^{-308} to 1.7×10^{308}. The computer actually stores an approximation of a decimal number by only keeping track of the 15 most significant digits.

Floating-point numbers may be entered as decimal numbers (1.45, -0.998, 12345.678) or entered in scientific notation using an "E" to mark the base 10 exponent (3.24e-1 = .324, 1.456e10 = 14560000000.0). You must not use a

2 http://whatis.techtarget.com/definition/integer
3 http://www.merriam-webster.com/dictionary/floating%E2%80%93point

thousand's separator when putting the numbers in your program.

When floating-point numbers are printed on the screen or the printed page, they will be shown with a thousand's separator, a decimal point, and a trailing zero if needed. This way, when you see one displayed, you will know it is a float and not an integer.

Strings

A string is "finite sequence of characters (i.e., letters, numerals, symbols and punctuation marks)"[4] In BASIC-256 a string is a bunch of letters, numbers, and other things surrounded by quotation marks. A string may be surrounded by single quotes (') or double quotes("). Be careful to always close your string with the same type of quote that you started with.

Examples include: "candy bar", "Say 'hi' to her for me.", and 'Why not?'.

Determining the Type of a Value or Variable

The **typeof** function in BASIC-256 that will tell you the type of the data stored in a variable or the type returned by an expression. **Typeof** returns an integer:

Typeof Value	Constant	Description
0	TYPE_UNASSIGNED	unassigned variable
1	TYPE_INT	integer
2	TYPE_FLOAT	floating-point
3	TYPE_STRING	string

Table 7: The typeof Function

4 http://www.linfo.org/string.html

typeof(expression or variable)

This function will return the type of an expression's result or the contents of a variable. If a variable had not been assigned a value the type will be 0. Expressions will return 1 for integers, 2 for floating-point numbers, and 3 for strings.

```
# types.kbs
print "integer 67 is type " + typeof(67)
print "floating-point 2.718 is type " + typeof(2.718)
print "string 'abcd' is type " + typeof('abcd')

print "variable a unassigned is type " + typeof(a)

a = 9
print "variable a containing " + a + " is type " + typeof(a)

a = 74.98
print "variable a containing " + a + " is type " + typeof(a)

a = "nine"
print "variable a containing " + a + " is type " + typeof(a)
```

Program 21: Data Types

```
integer 67 is type 1
floating-point 2.718 is type 2
string 'abcd' is type 3
variable a unassigned is type 0
variable a containing 9 is type 1
variable a containing 74.98 is type 2
variable a containing nine is type 3
```

Sample Output 21: Data Types

Converting Values from One Type to Another

BASIC-256 includes three functions that will convert values from one type to another. They are: **int()**, **float()**, and **string()**.

int(*expression*)

Return an integer value.

If the expression is floating-point number the decimal portion will be removed and just the whole part will be returned. No rounding will occur.

If the expression is a string, BASIC-256 will attempt to convert it to an integer (whole number). If the string does not contain a number then an error or warning will be displayed and zero will be returned.

float(*expression*)

Return a floating-point value.

If the expression is an integer, a floating-point number with the same value will be returned.

If the expression is a string, BASIC-256 will attempt to convert it to a floating-point number. If the string does not contain a number then an error or warning will be displayed and zero will be returned.

`string(`***`expression`***`)`

Return a string value.

If the expression is a numeric type (integer or float) then this function will return a string containing that number.

```
# intandstring.kbs

a = 9/2

# convert a to a string and concatenate
print "a is " + string(a)

# convert a to an integer
print "int(a) is " + int(a)

# round a to an integer
print "a rounded is " + int(a + .5)
```

Program 22: Converting Data Types

```
a is 4.5
int(a) is 4
a rounded is 5
```

Sample Output 22: Converting Data Types

Exercises:

Word Search

```
d s u h l s b h k s f m
a s s i g n m e n t f s
u n a s s i g n e d n t
s f m x y i l s v m m r
w h l o f n n y a f g i
i m o o b h u t r b t n
n x c r a z s y i t y g
t l v i t t n e a p p u
e e j v m c f v b p e r
g z f q w a u u l x o j
e c a o d b j t e z f d
r c r j z s n j n p d a
```

assignment, float, int, integer, shortcut, string, typeof, unassigned, variable

Problems

1. Create a program with two variables 'a' and 'b' that you will assign to two numbers. Print the sum of a and b, the difference of a and b, the difference of b and a, the product of a and b, the quotient of a divided by b, and the quotient of b divided by a. Run the program with several values of a and b. What happens when a or b are set to the value of zero?

Chapter 4: Sound and Music.

Now that we have color, graphics, and an understanding of variables, let's add sound and make some music. Basic concepts of the physics of sound and musical notation will be introduced. You will be able to translate a tune into frequencies and durations to have the computer synthesize a voice.

Sound Basics – Things you need to know about sound:

Sound is created by vibrating air striking your ear-drum. These vibrations are known as sound waves. When the air is vibrating quickly you will hear a high note and when the air is vibrating slowly you will hear a low note. The rate of the vibration is called frequency.

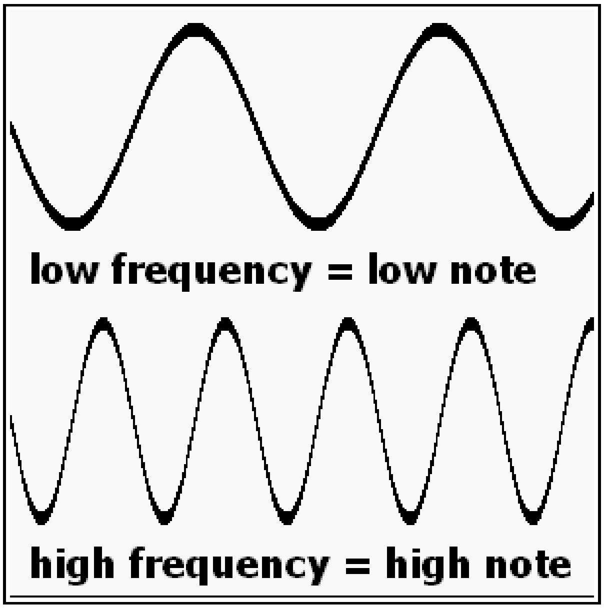


Illustration 9: Sound Waves

Frequency is measured in a unit called hertz (Hz). It represents how many cycles (ups and downs) a wave vibrates through in a second. A normal person can hear very low sounds at 20 Hz and very high sounds at 20,000 Hz. BASIC-256 can produce tones in the range of 50Hz to 7000Hz.

Another property of a sound is its length. Computers are very fast and can measure times accurately to a millisecond (ms). A millisecond (ms) is 1/1000 (one thousandths) of a second.

Let's make some sounds.

```
# sounds.kbs
sound 233, 1000
sound 466, 500
sound 233, 1000
```

Program 23: Play Three Individual Notes

You may have heard a clicking noise in your speakers between the notes played in the last example. This is caused by the computer creating the sound and needing to stop and think a millisecond or so. The sound statement also can be written using a list of frequencies and durations to smooth out the transition from one note to another.

In the program below, the first two values represent the frequency and duration of the first note. Once that is played the next two values are used to play the next note.

```
# soundslist.kbs
sound {233, 1000, 466, 500, 233, 1000}
```

Program 24: List of Sounds

This second sound program plays the same three tones for the same duration but the computer creates and plays all the sounds at once, making them smoother.

New Concept

```
sound frequency, duration
sound {frequency1, duration1, frequency2,
      duration2 …}
sound numeric_array[]
```

The basic soundstatement takes two arguments; (1) the frequency of the sound in Hz (cycles per second) and (2) the length of the tone in milliseconds (ms).

The second form of the sound statement uses a single list with curly braces to define the frequency and duration. This form can be confusing, be careful.

The third form of the sound statement uses an array containing frequencies and durations. Arrays are covered in a later chapter.

How do we get BASIC-256 to play a tune? The first thing we need to do is to convert the notes on a music staff to frequencies. Illustration 9 shows two octaves of music notes, their names, and the approximate frequency the note makes. In music you will also find a special mark called the rest. The rest means not to play anything for a certain duration. If you are using a list of sounds you can insert a rest by specifying a frequency of zero (0) and the needed duration for the silence.

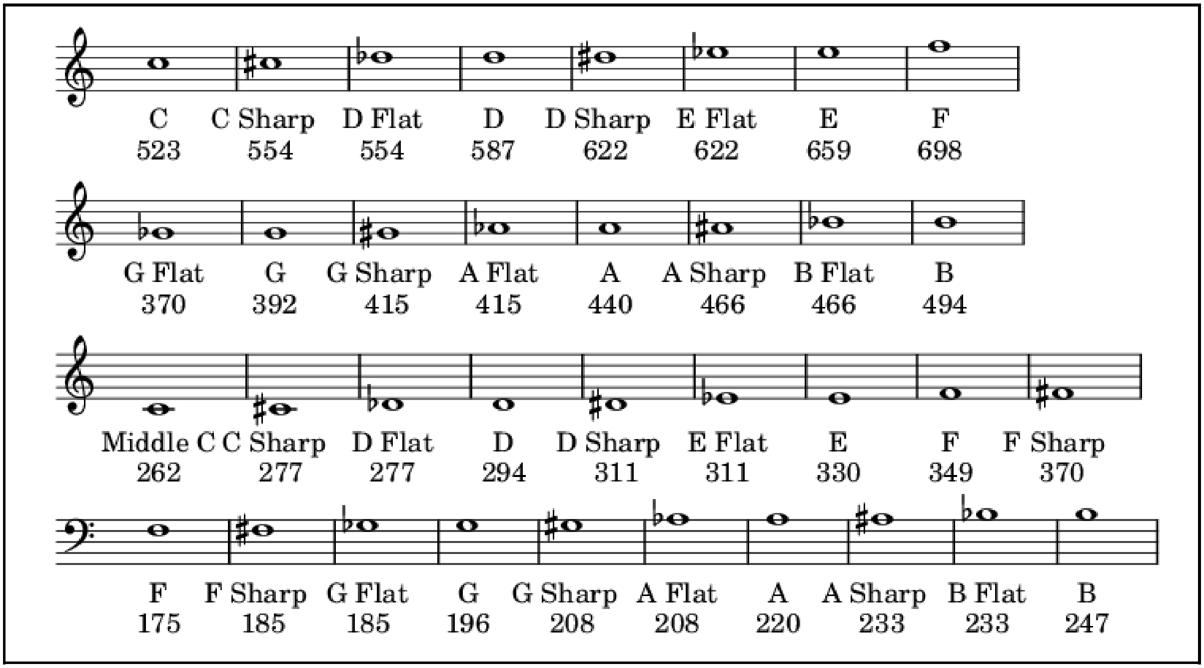


Illustration 10: Musical Notes

Take a little piece of music and then look up the frequency values for each of the notes. Why don't we have the computer play "Charge!". The music is in Illustration 11. You might notice that the high G in the music is not on the musical notes; if a note is not on the chart you can double (to make higher) or half (to make lower) the same note from one octave away.

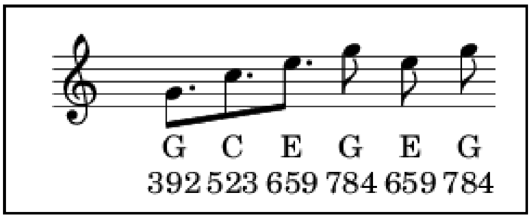



Illustration 11: Charge!

Now that we have the frequencies we need the duration for each of the notes. Table 3 shows most of the common note and rest symbols, how long they are when compared to each other, and a few typical durations.

Duration in milliseconds (ms) can be calculated if you know the speed if the music in beats per minute (BPM) using Formula 1.

$$Note\ Duration = 1000 * 60 / Beats\ Per\ Minute * Relative\ Length$$

Formula 1: Calculating Note Duration

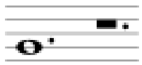
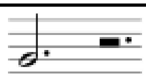
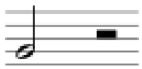
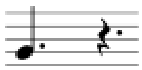
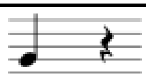
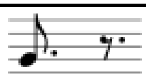
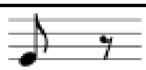
Note Name	Symbols for Note - Rest	Length in Beats	At 100 BPM	At 120 BPM	At 140 BPM
Dotted Whole		6.000	3600 ms	3000 ms	2571 ms
Whole		4.000	2400 ms	2000 ms	1714 ms
Dotted Half		3.000	1800 ms	1500 ms	1285 ms
Half		2.000	1200 ms	1000 ms	857 ms
Dotted Quarter		1.500	900 ms	750 ms	642 ms
Quarter		1.000	600 ms	500 ms	428 ms
Dotted Eighth		0.750	450 ms	375 ms	321 ms
Eighth		0.500	300 ms	250 ms	214 ms
Dotted Sixteenth		0.375	225 ms	187 ms	160 ms
Sixteenth		0.250	150 ms	125 ms	107 ms

Table 3: Musical Notes and Typical Durations

Now with the formula and table to calculate note durations, we can write the program to play "Charge!".

```
# charge.kbs - play charge
sound { 392, 375, 523, 375, 659, 375, 784, 250, 659,
250, 784, 250}
say "Charge!"
```

Program 25: Charge!

Instead of manually calculating the note durations, let's use a few variables to calculate and store the lengths for us. Using variables we could re-write the "Charge!" program using them to store the results of formulas to calculate note durations (Formula 1).

```
# charge2.kbs
# play charge - use variables
bpm = 120              # beats per minute
bms = 1000 * 60 / bpm          # ms per beat
dottedeighth = bms * .75
eighth = bms * .5
sound { 392, dottedeighth, 523, dottedeighth, 659,
dottedeighth, 784, eighth, 659, eighth, 784, eighth }
say "Charge!"
```

Program 26: Charge! with Variables

Big Program

For this chapter's big program let's take a piece of music by J.S. Bach and write a program to play it.

The musical score is a part of J.S. Bach's Little Fuge in G.

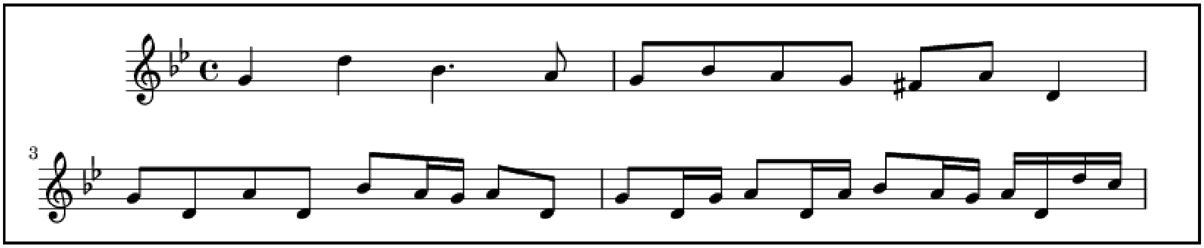


Illustration 12: First Four Measures of J.S. Bach's Little Fuge in G

```
# littlefuge.kbs
# Music by J.S.Bach - XVIII Fuge in G moll.

tempo = 100 # beats per minute
milimin = 1000 * 60 # miliseconds in a minute
q = milimin / tempo # quarter note is a beat
h = q * 2 # half note (2 quarters)
e = q / 2 # eight note (1/2 quarter)
s = q / 4 # sixteenth note (1/4 quarter)
de = e + s # dotted eight - eight + 16th
dq = q + e # doted quarter - quarter + eight

sound {392, q, 587, q, 466, dq, 440, e, 392, e, 466, e, 440, e, 392, e, 370, e, 440, e, 294, q, 392, e, 294, e, 440, e, 294, e, 466, e, 440, s, 392, s, 440, e, 294, e, 392, e, 294, s, 392, s, 440, e, 294, s, 440, s, 466, e, 440, s, 392, s, 440, s, 294, s}
```

Program 27: Big Program - Little Fuge in G

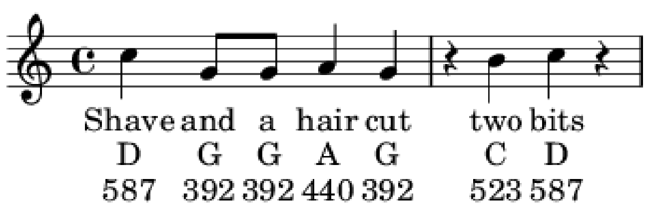
Exercises:

Word Search

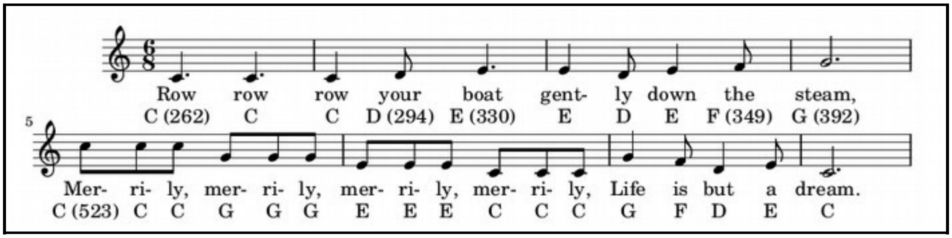
```
d j r a h e r t z q y t x
n a v a r i a b l e l z s
o s h a l f n g k j u e x
c s s h o r t c u t c g j
e i e h t h g i e a h i n
s g t u r l s l r t b k x
i n a t y f i b n d e d t
l m r s a i x e n e x l u
l e b y c n e u q e r f i
i n i b q t o e v a t c o
m t v z x s j w h o l e b
m u s i c r e t r a u q a
i j s q s e y t e t o n t
```

braces, eighth, frequency, half, hertz, millisecond, music, note, octave, quarter, shortcut, sixteenth, sound, vibrate, whole

Problems

1. Write a program using a single sound statement to play "Shave and a Hair Cut". Remember you must include the quarter rests in the second measure in your sound with a frequency of zero and the duration of a quarter note.

2. Type the sound statement below and insert the variable assignments before it to play “Row Row Row your Boat”. The variables c, d, e, f, g, and cc should contain the frequency of the notes of the tune. The variable n4 should contain the length in milliseconds of a quarter note; n2 twice n4, and n8 one half of n4.

```
sound {c,n4+n8, c,n4+n8, c,n4, d,n8, e,n4+n8,
e,n4, d,n8, e,n4, f,n8, g,n2+n4, cc,n8, cc,n8,
cc,n8, g,n8, g,n8, g,n8, e,n8, e,n8, e,n8, c,n8,
c,n8, c,n8, g,n4, f,n8, d,n4, e,n8, c,n2+n4}
```



Chapter 5: Thinking Like a Programmer

One of the hardest things to learn is how to think like a programmer. A programmer is not created by simple books or classes but grows from within an individual. To become a "good" programmer takes passion for technology, self learning, basic intelligence, and a drive to create and explore.

You are like the great explorers Christopher Columbus, Neil Armstrong, and Yuri Gagarin (the first human in space). You have an unlimited universe to explore and to create within the computer. The only restrictions on where you can go will be your creativity and willingness to learn.

A program to develop a game or interesting application can often exceed several thousand lines of computer code. This can very quickly become overwhelming, even to the most experienced programmer. Often we programmers will approach a complex problem using a three step process, like:

1. Think about the problem.
2. Break the problem up into pieces and write them down formally.
3. Convert the pieces into the computer language you are using.

Pseudocode:

Pseudocode is a fancy word for writing out, step by step, what your program needs to be doing. The word pseudocode comes from the Greek prefix "pseudo-" meaning fake and "code" for the actual computer programming statements. It is not created for the computer to use directly but it is made to help you understand the complexity of a problem and to break it down into meaningful pieces.

There is no single best way to write pseudocode. Dozens of standards exist and each one of them is very suited for a particular type of problem. In this introduction we will use simple English statements to understand our problems.

How would you go about writing a simple program to draw a school bus (like in Illustration 13)?

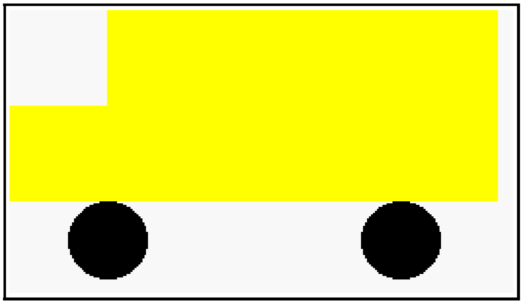


Illustration 13: School Bus

Let's break this problem into two steps:

- draw the wheels
- draw the body

Now let's break the initial steps into smaller pieces and write our pseudocode:

Set color to black. Draw both wheels. Set color to yellow. Draw body of bus. Draw the front of bus.

Table 4: School Bus - Pseudocode

Now that we have our program worked out, all we need to do is write it:

Set color to black.	`color black`
Draw both wheels.	`circle 50,120,20`
	`circle 200,120,20`
Set color to yellow.	`color yellow`
Draw body of bus.	`rect 50,0,200,100`
Draw the front of bus.	`rect 0,50,50,50`

Table 5: School Bus - Pseudocode with BASIC-256 Statements

The completed school bus program (Program 28) is listed below. Look at the finished program and you will see comment statements used in the program to help the programmer remember the steps that they used to initially solve the problem.

```
# schoolbus.kbs
# draw a school bus

clg

# draw wheels
color black
circle 50,120,20
circle 200,120,20

# draw bus body
color yellow
rect 50,0,200,100
rect 0,50,50,50
```

Program 28: School Bus

In the school bus example we have just seen there were many ways to break up the problem. You could have drawn the bus first and the wheels last, you could have drawn the front before the back,... We could list dozens of different ways this simple problem could have been tackled.

One very important thing to remember, THERE IS NO WRONG WAY to approach a problem. Some ways are better than others (fewer instructions, easier to read, …), but the important thing is that you solved the problem.

Flowcharting:

Another technique that programmers use to understand a problem is called flowcharting. Following the old adage of "a picture is worth a thousand words", programmers will sometimes draw a diagram representing the logic of a program. Flowcharting is one of the oldest and commonly used methods of drawing what a program is supposed to do.

This brief introduction to flowcharts will only cover a small part of what can be done with them, but with a few simple symbols and connectors you will be able to model very complex processes. This technique will serve you well not only in programming but in solving many problems thet you will come across. Here are a few of the basic symbols:

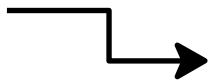
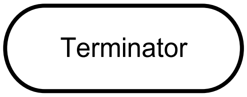
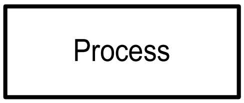
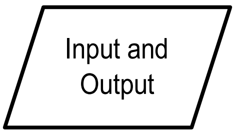
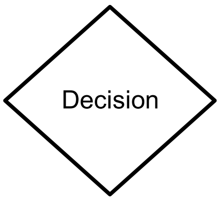
Symbol	Name and Description
	Flow – An arrow represents moving from one symbol or step in the process to another. You must follow the direction of the arrowhead.
	Terminator – This symbol tells us where to start and finish the flowchart. Each flowchart should have two of these: a start and a finish.
	Process – This symbol represents activities or actions that the program will need to take. There should be only one arrow leaving a process.
	Input and Output (I/O) – This symbol represents data or items being read by the system or being written out of the system. An example would be saving or loading files.
	Decision – The decision diamond asks a simple yes/no or true/false question. There should be two arrows that leave a decision. Depending on the result of the question we will follow one path out of the diamond.

Table 6: Essential Flowcharting Symbols

The best way to learn to flowchart is to look at some examples and to try your own hand it it.

Flowcharting Example One:

You just rolled out of bed and your mom has given you two choices for breakfast. You can have your favorite cold cereal or a scrambled egg. If you do not choose one of those options you can go to school hungry.

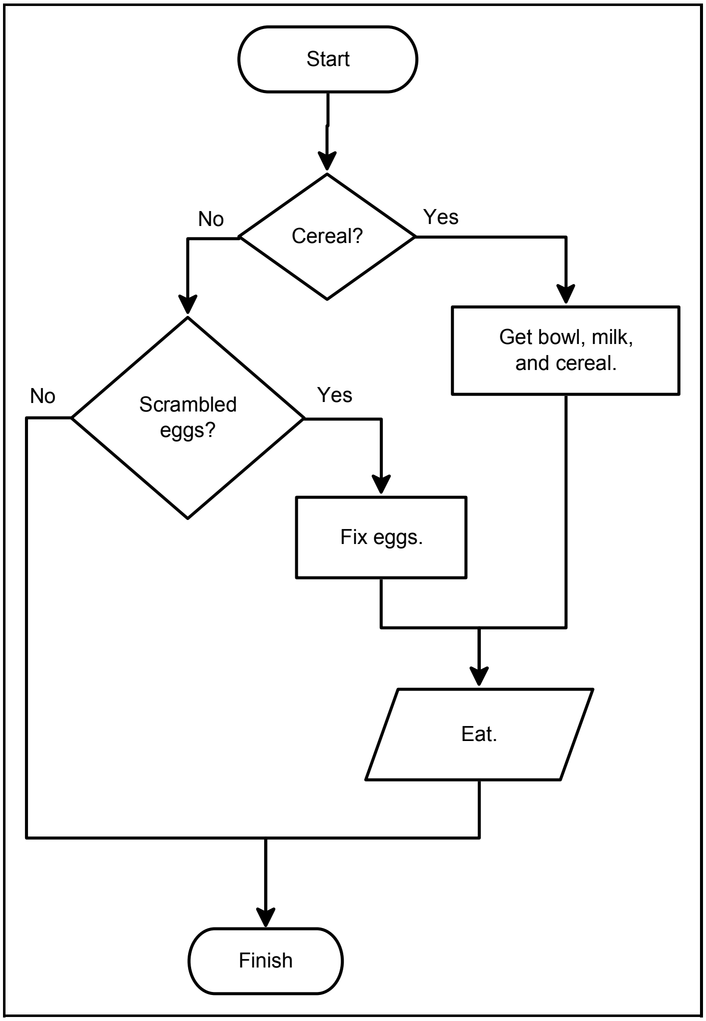


Illustration 14: Breakfast - Flowchart

Take a look at Illustration 14 (above) and follow all the arrows. Do you see how that picture represents the scenario?

Flowcharting Example Two:

Another food example. You are thirsty and want a soda from the machine. Take a look at Illustration 15 (below).

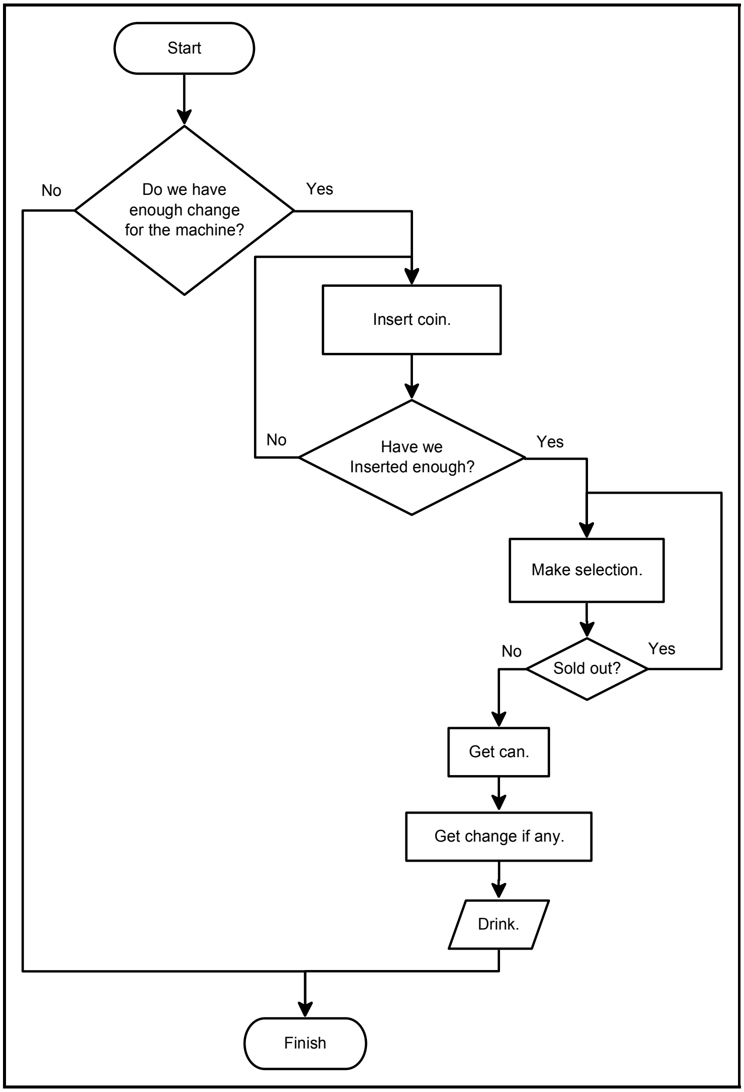


Illustration 15: Soda Machine - Flowchart

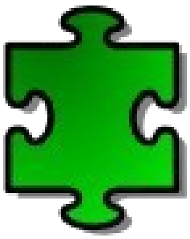
Notice in the second flowchart that there are a couple of times that we may need to repeat a process. You have not seen how to do that in BASIC-256, but it will be covered in the next few chapters.

Exercises:

Word Search

```
z d s y m b o l t r p
e m e w t a f r m r t
y d k c l u a v o s e
p q o z i h p g r p r
x r i c c s r n r e m
z f o w o a i e i t i
a u o c m d x o u s n
q l h m e p u p n q a
f o i q m s t e d u t
b n m h r u s w s b o
g e p r o b l e m p r
```

decision, flowchart, input, output, problem, process, programming, pseudocode, steps, symbol, terminator



Problems

1. In complete sentences can you write out the steps to make a peanut butter and jelly sandwich. Assume that the peanut butter jar, jelly jar, loaf of bread, place, and silverware are on the table in front of you. Can another person, who has never seen a PBJ, successfully make one using your directions?

2. In a flow chart (or in a similar diagram) diagram the process you go through to open the front door of your hours or apartment. Do you have your keys? Is the door locked? Is it already open?

3. In pseudocode (short statements) can you write out directions from your school or work to the nearest restaurant or gas station. Don't cheat and look the directions up on-line. Will the same directions get you back the same way or do the instructions need to be changed?

Chapter 6: Your Program Asks for Advice.

This chapter shows how BASIC-256 asks the user to enter strings and numbers, and how to use this in a program.

InputString – Getting Text From the User:

So far we have told the program everything it needs to know in the programming code. The next statement to introduce is inputstring The inputstringstatement captures a string that the user types into the text area and stores that value in a variable.

Let's take Program 22 and modify it so that it will ask you for a name and then say hello to that person.

```
# ilike.kbs
# using input to ask for a name

inputstring "enter your name>", name
message1 = name + " is my friend."
message2 = "I like " + name + "."

print message1
say message1
print message2
say message2
```

Program 29: I Like fill in the blank

```
enter your name>Vance
Vance is my friend.
I like Vance.
```

Sample Output 29: I Like fill in the blank


```
inputstring "prompt", variable
inputstring variable
```

The **inputstring** statement will retrieve a string that the user types into the text output area of the screen. The result will be stored in a variable that may be used later in the program.

A prompt message, if specified, will display on the text output area and the cursor will directly follow the prompt.

InputInteger and InputFloat – Getting Numbers

The "Math-wiz" program shows an example of input with numbers.

```
# mathwiz.kbs
# show several mathematical operations

inputfloat "a? ", a
inputfloat "b? ", b

print a + "+" + b + "=" + (a+b)
print a + "-" + b + "=" + (a-b)
print b + "-" + a + "=" + (b-a)
print a + "*" + b + "=" + (a*b)
print a + "/" + b + "=" + (a/b)
print b + "/" + a + "=" + (b/a)
```

Program 30: Math-wiz

```
a? 7.9
b? 6
7.9+6.0=13.9
```

```
7.9-6.0=1.9
6.0-7.9=-1.9
7.9*6.0=47.4
7.9/6.0=1.31666666667
6.0/7.9=0.759493670886
```

Sample Output 30: Math-wiz

.

New Concept

```
inputinteger "prompt", variable
inputinteger variable
inputfloat "prompt", variable
inputfloat variable
```

The **inputinteger** and **inputfloat** statements will allow a user to enter either an integer or float value and store that into a variable.

If the user enters a value that is not numeric, an error or warning will be displayed. If the "Runtime handling of bad type conversions" in the Preferences is set to either "warn" or "ignore" a zero (0) will be assigned to the variable.

The **inputfloat** statement will allow for a user to enter a number with a thousands separator (1,234,567.89) and will accept the number. The **inputinteger** statement only allows the numbers 0-9 and an optional leading minus sign.

A prompt message, if specified, will display on the text output area and the cursor will directly follow the prompt.

Here is another example using **inputinteger** and **inputstring**.

```
# sayname.kbs

inputstring "What is your name?", name
inputinteger "How old are you?", age

```

```
greeting =  "It is nice to meet you, " + name + "."
print greeting
say greeting

greeting =  "In 8 years you will be " + (age + 8) + " years old. Wow, that's old!"
print greeting
say greeting
```

Program 31: Fancy – Say Name

```
What is your name?Jo
How old are you?13
It is nice to meet you, Jo.
In 8 years you will be 21 years old. Wow,
that's old!
```

Sample Output 31: Fancy – Say Name

Input – Automatic Type Conversion

The last style of the input statement we will discuss is the plain input This statement will ask the user for something and automatically convert it to either a string, integer or floating-point value. This may be the behavior you wish but may cause problems in other places

New Concept

```
input "prompt", variable
input variable
```

The **input** statement will allow a user to enter a string, integer, or a floating-point number. After the input is complete, if the entry can be converted to an integer or a floating-point number it will and be stored that way. If the user enters a value that is not numeric, it will be stored as a string.

This automatic type assignment may cause some confusion as spaces, leading zeros, and trailing zeros after a decimal point will be stripped from numbers and they will be stored as integer or float values.

A prompt message, if specified, will display on the text output area and the cursor will directly follow the prompt.

Big Program

This chapter's "Big Program" is a silly story generator. Answer the questions with words and the computer will tell you a story.

```
# sillystory.kbs

print "A Silly Story."

inputstring "Enter a noun? ", noun1
inputstring "Enter a verb? ", verb1
inputstring "Enter a room in your house? ", room1
inputstring "Enter a verb? ", verb2
inputinteger "Enter an integer 2 or larger?", howmany
```

```
inputstring "Enter a plural noun? ", noun2
inputstring "Enter an adjective? ", adj1
inputstring "Enter a verb? ", verb3
inputstring "Enter a noun? ", noun3
inputstring "Enter Your Name? ", name

sentence = "A silly story, by " + name + "."
print sentence
say sentence

sentence = "One day, not so long ago, I saw a " + noun1 + " " + verb1 + " down the stairs."
print sentence
say sentence

sentence = "It was going to my " + room1 + " to " + verb2 + " " + string(howmany) + " " + noun2
print sentence
say sentence

sentence = "The " + noun1 + " became " + adj1 + " when I " + verb3 + " a " + noun3 + "."
print sentence
say sentence

sentence = "The End."
print sentence
say sentence
```

Program 32: Big Program - Silly Story Generator

```
A Silly Story.
Enter a noun? car
Enter a verb? drive
Enter a room in your house? bathroom
Enter a verb? walk
Enter an integer 2 or larger?5
Enter a plural noun? cows
```

```
Enter an adjective? big
Enter a verb? lifted
Enter a noun? hippo
Enter Your Name? Mary
A silly story, by Mary.
One day, not so long ago, I saw a car drive
down the stairs.
It was going to my bathroom to walk 5 cows
The car became big when I lifted a hippo.
The End.
```

Sample Output 32: Big Program - Silly Story Generator

Exercises:

Word Search

f	r	s	a	i	m	m	k	o	g	w	x
i	l	s	w	n	f	e	a	a	l	i	v
n	q	o	w	p	g	o	c	e	h	n	p
u	j	n	a	u	r	i	n	y	k	p	u
t	j	p	n	t	f	y	h	a	g	u	i
i	s	t	i	n	t	e	g	e	r	t	f
n	x	z	s	s	b	a	b	v	n	s	d
t	i	n	p	u	t	f	l	o	a	t	o
e	g	e	n	h	x	w	o	a	a	r	d
g	z	f	p	r	o	m	p	t	b	i	z
e	m	q	d	r	l	r	e	p	l	n	m
r	q	b	i	o	n	f	s	n	u	g	r

float, input, inputfloat, inputstring, integer, inutinteger, prompt, string

Problems

1. Write a program to ask for three names. Store them in string variables. Once the user enters the third name have the computer recite the classic playground song using the names:

```
[Name One] and [Name Two]
sitting in a tree,
K-I-S-S-I-N-G.
First comes love,
then comes marriage,
then comes [Name Three]
in a baby carriage!
```

2. Write a program to ask for an adjective, noun, animal, and a

	sound. Once the use enters the last one, build a single string variable (using concatenation) to say a verse of Old MacDonald. Print the result out with a single statement and say it with a single statement. (Adapted from The Old Macdonald Mad Lib from http://www.madglibs.com)
	
	`[Adjective] MacDonald had a`
	`[Noun], E-I-E-I-O and on that`
	`[Noun] he had an animal, E-I-E-I-O`
	`with a [Sound] [Sound] here and a`
	`[Sound] [Sound] there,`
	`here a [Sound], there a [Sound],`
	`everywhere a [Sound] [Sound],`
	`[Adjective] MacDonald had a`
	`[Noun], E-I-E-I-O.`

Chapter 7: Decisions, Decisions, Decisions.

The computer is also a whiz at comparing things. In this chapter we will explore how to compare two expressions, how to work with complex comparisons, and how to optionally execute statements depending on the results of our comparisons. We will also look at how to generate random numbers.

True and False:

The BASIC-256 language has one more special type of data, it is the Boolean data type. Boolean values are either true or false and are usually the result of comparisons and logical operations. Also to make them easier to work with there are two Boolean constants that you can use in expressions, they are: true and false.

true
false

The two Boolean constants true and false can be used in any numeric or logical expression but are usually the result of a comparison or logical operator. Actually, the constant true is stored as the number one (1) and false is stored as the number zero (0).

Comparison Operators:

Previously we have discussed the basic arithmetic operators, it is now time to look at some additional operators. We often need to compare two values in a program to help us decide what to do. A comparison operator works with two values and returns true or false based on the result of the comparison.

Operator	**Operation**
<	Less Than expression1 < expression2 Expression is true(1) if expression1 is less than expression2, otherwise it is false(0).
<=	Less Than or Equal expression1 <= expression2 Expression is true(1) if expression1 is less than or equal to expression2, otherwise it is false(0).
>	Greater Than expression1 > expression2 Expression is true(1) if expression1 is greater than expression2, otherwise it is false(0).
>=	Greater Than or Equal expression1 >= expression2 Expression is true(1) if expression1 is greater than or equal to expression2, otherwise it is false(0).
=	Equal expression1 = expression2 Expression is true(1) if expression1 is equal to expression2, otherwise it is false(0).
<>	Not Equal Expression1 <> expression2 Expression is true(1) if expression1 is not equal to expression2, otherwise it is false(0).

Table 7: Comparison Operators

< <= > >= = <>

The six comparison operations are: less than (<), less than or equal (<=), greater than (>), greater than or equal (>=), equal (=), and not equal (<>). They are used to compare numbers and strings.

Strings are compared alphabetically left to right.

Making Simple Decisions – The If Statement:

The ifstatement can use the result of a comparison to optionally execute a statement or block of statements. This first program (Program 33) uses three ifstatements to display whether your friend is older, the same age, or younger.

```
# compareages.kbs
# compare two ages

inputinteger "how old are you?", yourage
inputinteger "how old is your friend?", friendage

print "You are ";
if yourage < friendage then print "younger than";
if yourage = friendage then print "the same age as";
if yourage > friendage then print "older than";
print " your friend"
```

Program 33: Compare Two Ages

```
how old are you?13
how old is your friend?12
You are older than your friend
```

Sample Output 33: Compare Two Ages

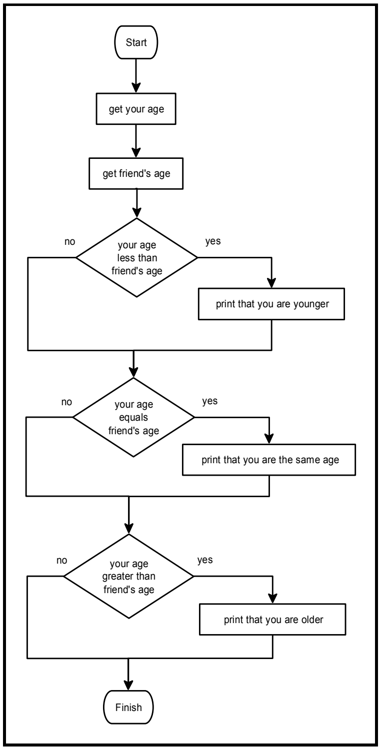


Illustration 16: Compare Two Ages - Flowchart

`if condition then statement`

If the condition evaluates to truethen execute the statement following the thenclause.

Random Numbers:

When we are developing games and simulations it may become necessary for us to simulate dice rolls, spinners, and other random happenings. BASIC-256 has a built in random number generator to do these things for us.


```
rand
rand()
```

A random number is returned when rand is used in an expression. The returned number ranges from zero to one, but will never be one ($0 \leq n < 1.0$).

Often you will want to generate an integer from 1 to r, the following statement can be used **n = int(rand * r) + 1**

```
# coinflip.kbs

coin = rand
if coin < .5 then print "Heads."
if coin >= .5 then print "Tails."
```

Program 34: Coin Flip

```
Tails.
```

Sample Output 34: Coin Flip

Warning	In program 34 you may have been tempted to use the rand expression twice, once in each if statement. This would have created what we call a "Logical Error". Remember, each time the randexpression is executed it returns a different random number.

Logical Operators:

Sometimes it is necessary to join simple comparisons together. This can be done with the four logical operators: and, or, xor, and not. The logical operators work very similarly to the way conjunctions work in the English language, except that "or" is used as one or the other or both.

<table>
<tr><th>Operator</th><th>Operation</th></tr>
<tr><td>AND</td><td>Logical And
expression1 AND expression2
If both expression1 and experssion2 are true then return a true value, else return false.

<table>
<tr><th colspan="2" rowspan="2">AND</th><th colspan="2">expression1</th></tr>
<tr><th>TRUE</th><th>FALSE</th></tr>
<tr><th rowspan="2">expression2</th><th>TRUE</th><td>TRUE</td><td>FALSE</td></tr>
<tr><th>FALSE</th><td>FALSE</td><td>FALSE</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td>OR</td><td>Logical Or
expression1 OR expression2
If either expression1 or experssion2 are true then return a true value, else return false.

<table>
<tr><th colspan="2" rowspan="2">OR</th><th colspan="2">expression1</th></tr>
<tr><th>TRUE</th><th>FALSE</th></tr>
<tr><th rowspan="2">expression2</th><th>TRUE</th><td>TRUE</td><td>TRUE</td></tr>
<tr><th>FALSE</th><td>TRUE</td><td>FALSE</td></tr>
</table>
</td></tr>
<tr><td>XOR</td><td>Logical Exclusive Or
expression1 XOR expression2
If only one of the two expressions is true then return a true value, else return false. The XOR operator works like "or" often does in the English language - "You can have your cake xor you can eat it".

<table>
<tr><th colspan="2" rowspan="2">OR</th><th colspan="2">expression1</th></tr>
<tr><th>TRUE</th><th>FALSE</th></tr>
<tr><th rowspan="2">expression2</th><th>TRUE</th><td>FALSE</td><td>TRUE</td></tr>
<tr><th>FALSE</th><td>TRUE</td><td>FALSE</td></tr>
</table>
</td></tr>
<tr><td>NOT</td><td>Logical Negation (Not)
NOT expression1
Return the opposite of expression1. If expression 1 was true then return false. If experssion1 was false then return a true.

<table>
<tr><th colspan="2">NOT</th><th></th></tr>
<tr><th rowspan="2">expression 1</th><th>TRUE</th><td>FALSE</td></tr>
<tr><th>FALSE</th><td>TRUE</td></tr>
</table>
</td></tr>
</table>

New Concept

```
and  or  xor  not
```

The four logical operations: logical and, logical or, logical exclusive or, and logical negation (not) join or modify comparisons.

You may also use parenthesis to group operations together.

Making Decisions with Complex Results – If/End If:

When we are writing programs it sometimes becomes necessary to do multiple statements when a condition is true. This is done with the alternate format of the **if** statement. With this statement you do not place a statement on the same line as the **if**, but you place multiple (one or more) statements on lines following the **if** statement and then close the block of statements with the **end if** statement.

New Concept

```
if condition then
    statement(s) to execute when true
end if
```

The **if**/**end if** statements allow you to create a block of programming code to execute when a condition is true. It is customary to indent the statements with in the **if/end if** statements so they are not confusing to read.

In the following example you will see **if** statements nested inside another **if** statement. It is important that you remember that the inner **if**s will only be tested when the outer **if** ia true.

```
# dice.kbs - roll 2 6-sided dice

die1 = int(rand * 6) + 1
die2 = int(rand * 6) + 1
total = die1 + die2

print "die 1 = " + die1
print "die 2 = " + die2
message = "You rolled " + total + "."

if die1 = die2 then
     message += " Doubles."
     if total = 2 then
          message += " Snake eyes."
     end if
     if total = 12 then
          message += " Box Cars."
     end if
end if

print message
```

Program 35: Rolling Dice

```
die 1 = 1
die 2 = 1
You rolled 2. Doubles. Snake eyes.
```

Sample Output 35: Rolling Dice

"Edit" then "Beautify" on the menu

The "Beautify" option on the "Edit" menu will clean up the format of your program to make it easier to read. It will remove extra spaces from the beginning and ending of lines and will indent blocks of code (like in the **if/end if** statements).

Deciding Both Ways – If/Else/End If:

The third and last form of the **if** statement is the **if/else/end if**. This extends the **if/end if** statements by allowing you to create a block of code to execute if the condition is trueand another block to execute when the condition is false.

```
if condition then
    statement(s) to execute when true
else
    statement(s) to execute when false
end if
```

The **if**, **else**, and **end if** statements allow you to define two blocks of programming code. The first block, after the **then** clause, executes if the condition is trueand the second block, after the **else** clause, will execute when the condition is false.

Program 36 re-writes Program 34 using the elsestatement.

```
# coinflip2.kbs
# coin flip with else

coin = rand
```

```
if coin < .5 then
   print "Heads."
   say "Heads."
else
   print "Tails."
   say "Tails."
end if
```

Program 36: Coin Flip – With Else

```
Heads.
```

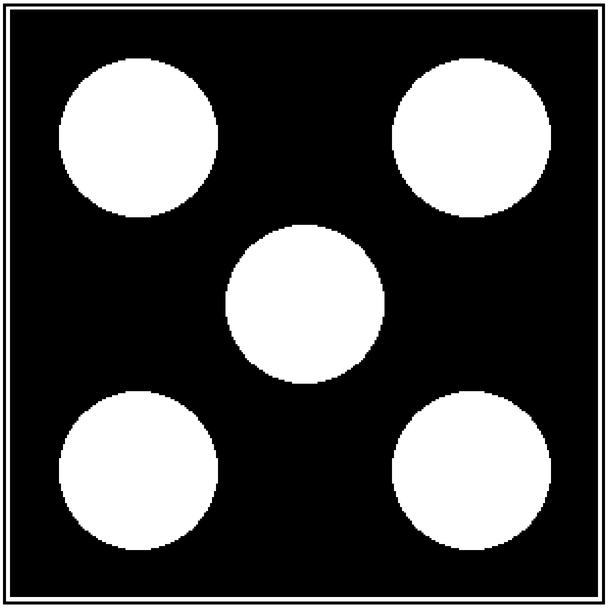
Sample Output 36: Coin Flip – With Else

Big Program	This chapter's big program is a program to roll a single 6-sided die and then draw on the graphics display the number of dots.

```
# dieroll.kbs - roll a 6-sided die on the screen

# radius of the dots
r = 40
# z1, z2, and z3 contain the center if the dots in each row and column
z1 = 65
z2 = 150
z3 = 235

# get roll
roll = int(rand * 6) + 1

```

```
clg black

color white
# top row
if roll <> 1 then circle z1,z1,r
if roll = 6 then circle z2,z1,r
if roll >= 4 and roll <= 6 then circle z3,z1,r
# middle row
if roll = 1 or roll = 3 or roll = 5 then circle z2,z2,r
# bottom row
if roll >= 4 and roll <= 6 then circle z1,z3,r
if roll = 6 then circle z2,z3,r
if roll <> 1 then circle z3,z3,r

message = "You rolled a " + roll + "."
print message
say message
```

Program 37: Big Program - Roll a Die and Draw It



Sample Output 37: Big Program - Roll a Die and Draw It

Exercises:


```
b t t h e n m r n s
i o r w l f o r z e
e d o u l d d o d s
r n r l e w n t j l
a a e u e t a a r e
p n t l n a r r o o
m o a a d s n e p l
o t e u i h l p t e
c i r q f f s o h s
w f g e e s l a f s
```

and, boolean, compare, else, endif, equal, false, greater, if, less, not, operator, or, random, then, true

Problems

1. Write a program that will toss a coin and tell you if your guess was correct. Assign a variable with a random number. Ask the user to enter the letter 'h' or 't' (for heads or tails). If the number is less than .5 and the user entered 'h' or the number was greater than or equal .5 and the user chose 't' then tell them they won the toss.

2. Modify program #1 in this chapter to also tell the user that they did not win the toss.

3. Write a simple program to draw a round of rock, paper, scissors. Use two numeric variables and assign a draw (random number) to each one. If a variable is less than 1/3 then it will be rock, greater than or equal to 1/3 and less than 2/3 it will be paper, and 2/3 or greater it will be scissors. Display what the two draws are.

4. Take the simple rock, paper, scissors draw program from #3 in this chapter and add rules to say who won. Remember “paper covers rock”, “rock smashes scissors”, and “scissors cut paper”. If both players draw the same thing then declare the round a “draw”.

5. Take the rock paper scissors game from #4 and add graphics and sound. Draw paper as a white rectangle, rock as a darkorange circle, and scissors as a red X. Have the computer announce the winner.

Chapter 8: Looping and Counting - Do it Again and Again.

So far our program has started, gone step by step through our instructions, and quit. While this is OK for simple programs, most programs will have tasks that need to be repeated, things counted, or both. This chapter will show you the three looping statements, how to speed up your graphics, and how to slow the program down.

The For Loop:

The most common loop is the **for** loop. The **for** loop repeatedly executes a block of statements a specified number of times, and keeps track of the count. The count can begin at any number, end at any number, and can step by any increment. Program 38 shows a simple **for** statement used to say the numbers 1 to 10 (inclusively). Program 39 will count by 2 starting at zero and ending at 10.

```
# for.kbs
for t = 1 to 10
   print t
   say t
next t
```

Program 38: For Statement

```
1
2
3
4
5
6
7
```

```
8
9
10
```

Sample Output 38: For Statement

```
# forstep2.kbs
for t = 0 to 10 step 2
   print t
   say t
next t
```

Program 39: For Statement – With Step

```
0
2
4
6
8
10
```

Sample Output 39: For Statement – With Step

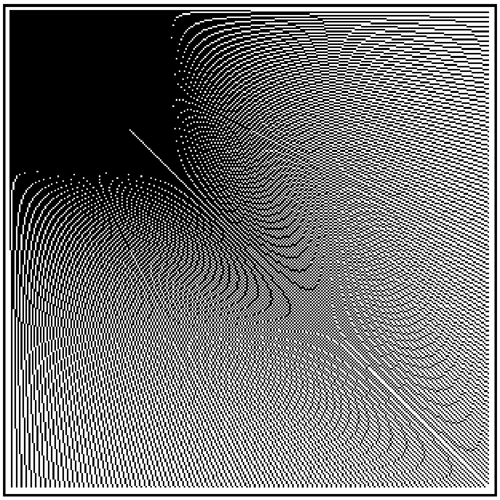

```
for variable = expr1 to expr2 [step expr3]
    statement(s)
next variable
```

Execute a specified block of code a specified number of times. The variablewill begin with the value of expr1 The variablewill be incremented by expr3(or one if step is not specified) the second and subsequent time through the loop. Loop terminates if variable exceeds expr2

Using a loop we can easily draw very interesting graphics. Program 40 will draw a Moiré Pattern. This really interesting graphic effect is caused by the computer being unable to draw perfectly straight lines. What is actually drawn are pixels in a stair step fashion to approximate a straight line. If you look closely at the lines we have drawn you can see that they actually are jagged.

```
# moire.kbs - draw a moire pattern

clg white
color black
for t = 1 to 300 step 3
   line 0,0,300,t
   line 0,0,t,300
next t
```

Program 40: Moiré Pattern



Sample Output 40: Moiré Pattern

Explore

What kind of Moiré Patterns can you draw? Start in the center, use different step values, overlay one on top of another, try different colors, go crazy.

For statements can even be used to count backwards. To do this set the **step** to a negative number.

```
# stepneg1.kbs

for t = 10 to 0 step -1
   print t
   pause 1.0
next t
```

Program 41: For Statement – Countdown

```
10
9
8
7
6
5
4
3
2
1
0
```

Sample Output 41: For Statement – Countdown


```
pause seconds
```

The **pause** statement tells BASIC-256 to stop executing the current program for a specified number of seconds. The number of seconds may be a decimal number if a fractional second pause is required.

Do Something Until I Tell You To Stop:

The next type of loop is the **do/until**. The **do/until** repeats a block of code one or more times. At the end of each iteration a logical condition is tested. The loop repeats as long as the condition is false. Program 42 uses the **do/until** loop to repeat until the user enters a number from 1 to 10.

```
# dountil.kbs

do
   inputinteger "enter an integer from 1 to 10?",n
until n>=1 and n<=10
print "you entered " + n
```

Program 42: Get a Number from 1 to 10

```
enter an integer from 1 to 10?66
enter an integer from 1 to 10?-56
enter an integer from 1 to 10?3
you entered 3
```

Sample Output 42: Get a Number from 1 to 10


```
do
    statement(s)
until condition
```

Do the statements in the block over and over again while the condition is false.

The statements will be executed **one or more times**.

Do Something While I Tell You To Do It:

The third type of loop is the **while/end while**. It tests a condition before executing each iteration and if it evaluates to true then executes the code in the loop. The **while/end while** loop may execute the code inside the loop zero or more times.

Sometimes we will want a program to loop forever, until the user stops the program. This can easily be accomplished using the Boolean trueconstant (see Program 43).

```
# whiletrue.kbs

while true
   print "nevermore ";
end while
```

Program 43: Loop Forever

```
nevermore.
nevermore.
nevermore.
nevermore.
nevermore.
```

```
… runs until you stop it
```

Sample Output 43: Loop Forever


```
while condition
    statement(s)
end while
```

Do the statements in the block over and over again while the condition is true.

The statements will be executed **zero or more times**.

Program 44 uses a while loop to count from 1 to 10 like Program 38 did with a **for** statement.

```
# whilefor.kbs

t = 1
while t <= 10
   print t
   t = t + 1
end while
```

Program 44: While Count to 10

```
1
2
3
4
5
6
7
```

```
8
9
10
```

Sample Output 44: While Count to 10

Continuing and Exiting Loops

Sometimes it becomes necessary for a programmer to jump out of a loop before it would normally terminate (exit) or to start the next loop (continue) without executing all of the code.

```
# exitwhile.kbs - adding machine

total = 0
while true
   inputfloat "Enter Value (-999 to exit) > ", v
   if v = -999 then exit while
   total = total + v
end while

print "Your total was " + total
```

Program 45: Adding Machine - Using Exit While

```
Enter Value (-999 to exit) > 34
Enter Value (-999 to exit) > -34
Enter Value (-999 to exit) > 234
Enter Value (-999 to exit) > 44
Enter Value (-999 to exit) > -999
Your total was 278.0
```

Sample Output 45: Adding Machine - Using Exit While


```
exit do
exit for
exit while
```

Jump out of the current loop and skip the remaining code in the loop.


```
continue do
continue for
continue while
```

Do not execute the rest of the code in this loop but loop again like normal.

Fast Graphics:

When we need to execute many graphics quickly, like with animations or games, BASIC-256 offers us a fast graphics system. To turn on this mode you execute the **fastgraphics** statement. Once **fastgraphics** mode is started the graphics output will only be updated once you execute the **refresh** statement.

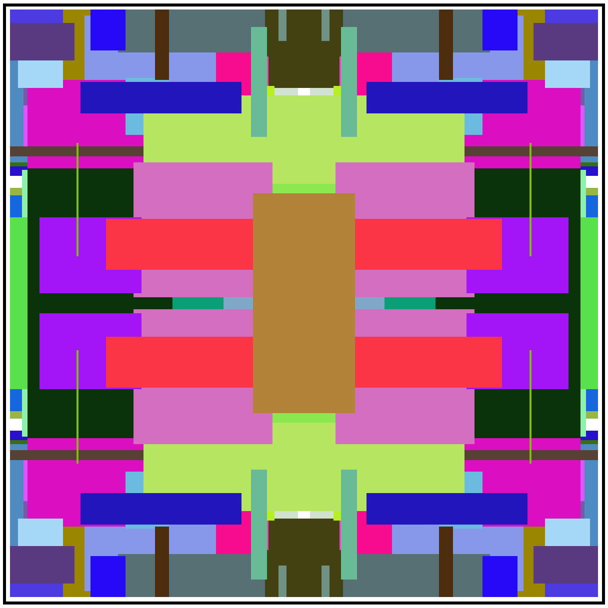

```
fastgraphics
refresh
```

Start the **fastgraphics** mode. In fast graphics the screen will only be updated when the **refresh** statement is executed.

Once a program executes the **fastgraphics** statement it can not return to the standard graphics (slow) mode.

```
# kaleidoscope.kbs

clg
fastgraphics
while true
     for t = 1 to 100
          r = int(rand * 256)
          g = int(rand * 256)
          b = int(rand * 256)
          x = int(rand * 300)
          y = int(rand * 300)
          h = int(rand * 100)
          w = int(rand * 100)
          color rgb(r,g,b)
          rect x,y,w,h
          rect 300-x-w,y,w,h
          rect x,300-y-h,w,h
          rect 300-x-w,300-y-h,w,h
     next t
     refresh
     pause 1
end while
```

Program 46: Kaleidoscope



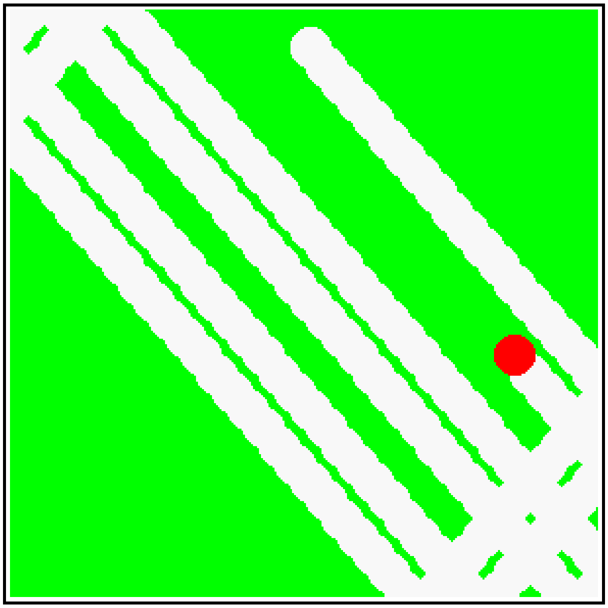
Sample Output 46: Kaleidoscope

Big Program	In this chapter's "Big Program" let's use a while loop to animate a ball bouncing around on the graphics display area.

```
# bouncingball.kbs

fastgraphics

# starting position of ball
x = rand * 300
y = rand * 300
# size of ball
```

```
r = 10
# speed in x and y directions
dx = rand * r - r / 2
dy = rand * r - r / 2

clg green

while true
     # erase old ball
     color white
     circle x,y,r
     # calculate new position
     x = x + dx
     y = y + dy
     # if off the edges turn the ball around
     if x < 0 or x > 300 then
          dx = dx * -1
          sound 1000,50
     end if
     # if off the top or bottom turn the ball around
     if y < 0 or y > 300 then
          dy = dy * -1
          sound 1500,50
     end if
     # draw new ball
     color red
     circle x,y,r
     # update the display
     refresh
     # slow the ball down
     pause .05
end while
```

Program 47: Big Program - Bouncing Ball



Sample Output 47: Big Program - Bouncing Ball

Exercises:

Word Search

```
f l g b w p e t s w i i
f a w t b q l i t n u i
t n s n v h p h b c f e
i a k t c v r o o e l l
x d r k g e w n o i l c
e x o u f r d e h l o i
i g f r y i a w l n l c
t x e n t g d p t i w k
g s d i o n e i h p h a
h w o a e d n z m i g w
x n s d z u u d w t c d
x o m i e h d g m o v s
```

condition, continue, do, endwhile, exit, fastgraphics, for, loop, next, refresh, step, until, while

Problems

1. Write a program that uses the **for** loop to sum the integers from 1 to 42 and display the answer. Hint: before the loop assign a variable to zero to accumulate the total.

2. Write a program that asks the user for an integer from 2 to 12 in a loop. Keep looping until the user enters a number in the range. Calculate the factorial (n!) of the number using a **for** loop and display it. Remember 2! is 1*2, 3! is 1*2*3, and n! Is n * (n-1)!.

3. Write a program to display one through 8 multiplied by 1 through 8. Hint: use a **for** loop inside another **for** loop. Format your output to look like:

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
1 * 7 = 7
1 * 8 = 8
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
…
```

4. Re-write #3 to make your output in table format, like:

```
 1   2   3   4   5   6   7   8
 2   4   6   8  10  12  14  16
 3   6   9  12  15  18  21  24
 4   8  12  16  20  24  28  32
 5  10  15  20  25  30  35  40
 6  12  18  24  30  36  42  48
 7  14  21  28  35  42  49  56
 8  16  24  32  40  48  56  64
```

Chapter 9: Custom Graphics – Creating Your Own Shapes.

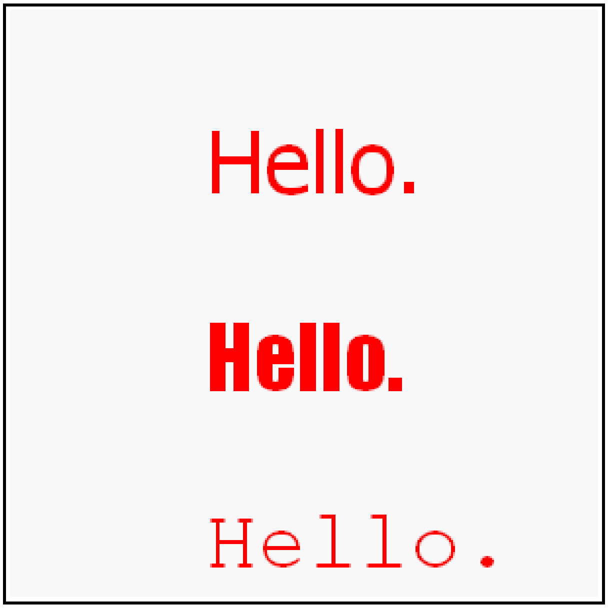
This chapter we will show you how to draw colorful words and special shapes on your graphics window. Several topics will be covered, including: fancy text; drawing polygons on the graphics output area; and stamps, where we can position, re-size, and rotate polygons. You also will be introduced to angles and how to measure them in radians.

Fancy Text for Graphics Output:

You have been introduced to the printstatement (Chapter 1) and can output strings and numbers to the text output area. The textand fontstatements allow you to place numbers and text on the graphics output area in a variety of styles.

```
# graphichello.kbs
# drawing text

clg
color red
font "Tahoma",33,100
text 100,100,"Hello."
font "Impact",33,50
text 100,150,"Hello."
font "Courier New",33,50
text 100,250,"Hello."
```

Program 48: Hello on the Graphics Output Area



Sample Output 48: Hello on the Graphics Output Area


```
text x, y, expression
```

Draw the contents of the *expression* on the graphics output area with it's top left corner specified by *x* and *y*. Use the font, size, and weight specified in the last **font** statement.

New Concept

`font font_name, size_in_point, weight`

Set the font, size, and weight for the next textstatement to use to render text on the graphics output area.

Argument	Description
font_name	String containing the system font name to use. A font must be previously loaded in the system before it may be used. Common font names are displayed below.
size_in_point	Height of text to be rendered in a measurement known as point. There are 72 points in an inch.
weight	Number from 1 to 100 representing how dark letter should be. Use 25 for light, 50 for normal, and 75 for bold.

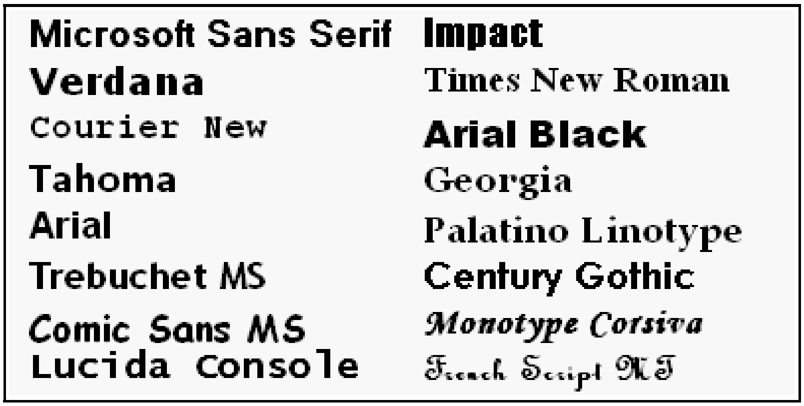


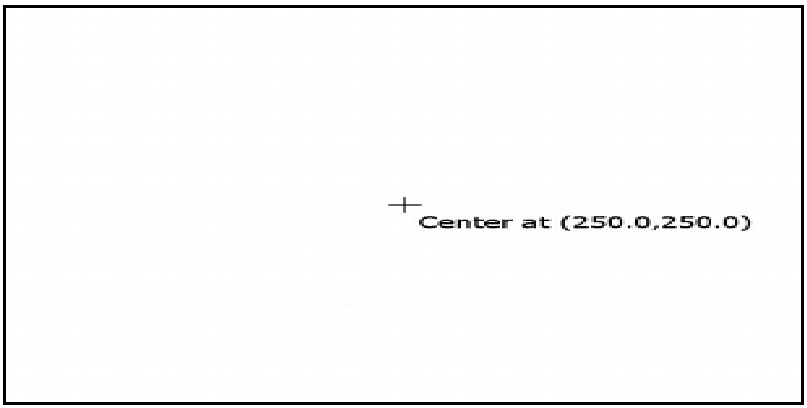
Illustration 17: Common Windows Fonts

Resizing the Graphics Output Area:

By default the graphics output area is 300x300 pixels. While this is sufficient for many programs, it may be too large or too small for others. The **graphsize** statement will re-size the graphics output area to what ever custom size you require. Your program may also use the **graphwidth** and **graphheight** functions to see what the current graphics size is set to.

```
# resizegraphics.kbs
# resize the graphics output area

graphsize 500,500
xcenter = graphwidth/2
ycenter = graphheight/2

color black
line xcenter, ycenter - 10, xcenter, ycenter + 10
line xcenter - 10, ycenter, xcenter + 10, ycenter

font "Tahoma",12,50
text xcenter + 10, ycenter + 10, "Center at (" + xcenter + "," + ycenter + ")"
```

Program 49: Re-size Graphics



Sample Output 49: Re-size Graphics

New Concept	**graphsize *width*, *height*** Set the graphics output area to the specified *height* and *width*.

New Concept	**graphwidth or graphwidth()** **graphheight or graphheight()** Functions that return the current graphics height and width for you to use in your program.

Creating a Custom Polygon:

In previous chapters we learned how to draw rectangles and circles. Often we want to draw other shapes. The poly statement will allow us to draw a custom polygon anywhere on the screen.

Let's draw a big red arrow in the middle of the graphics output area. First, draw it on a piece of paper so we can visualize the coordinates of the vertices of the arrow shape.

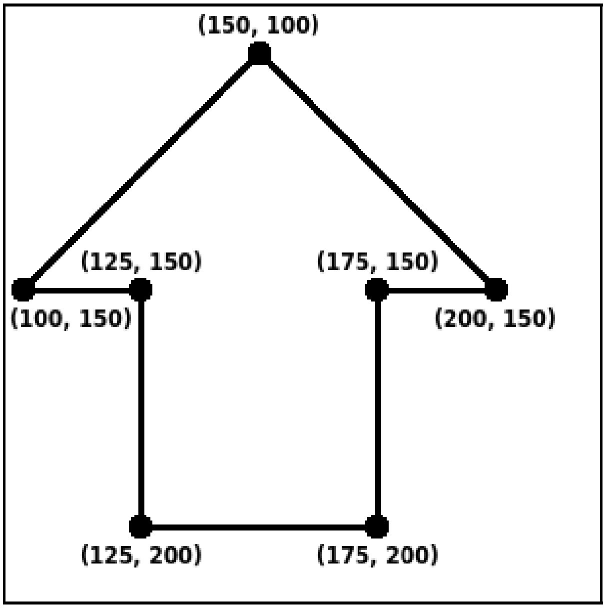
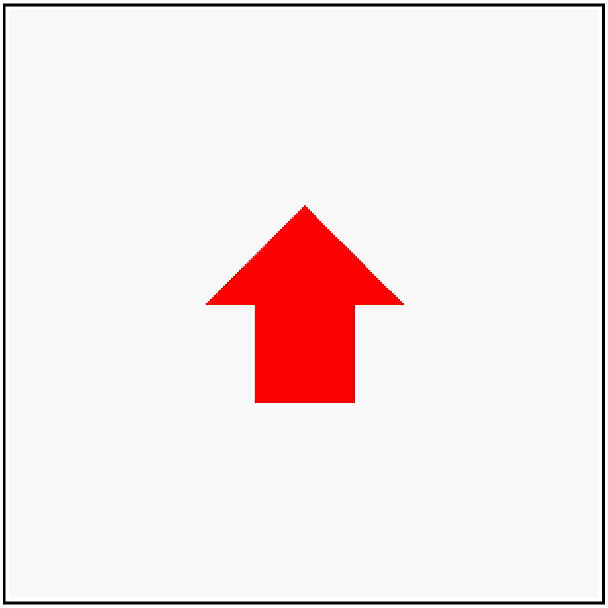


Illustration 18: Big Red Arrow

Now start at the top of the arrow going clockwise and write down the x and y values.

```
# bigredarrow.kbs
clg
color red
poly { 150, 100, 200, 150, 175, 150, 175, 200, 125, 200, 125, 150, 100, 150 }
```

Program 50: Big Red Arrow



Sample Output 50: Big Red Arrow


```
poly {x1, y1, x2, y2 ... }
poly numeric_array[]
```

Draw a polygon using the points for the corners. The array is evaluated by taking two values at a time and using them for the x and y values to plot a vertex.

Stamping a Polygon:

The **poly** statement allowed us to place a polygon at a specific location on the screen but it would be difficult to move it around or adjust it. These problems are solved with the **stamp** statement. The **stamp** statement takes a location on the screen, optional scaling (re-sizing), optional rotation, and a polygon definition to allow us to place a polygon anywhere we want it in the screen.

Let's draw an equilateral triangle (all sides are the same length) on a piece of paper. Put the point (0,0) at the top and make each leg 10 units long (see Illustration 19).

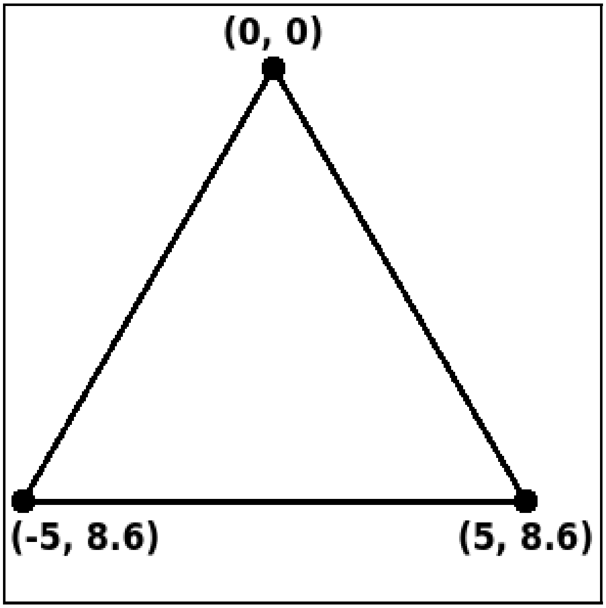


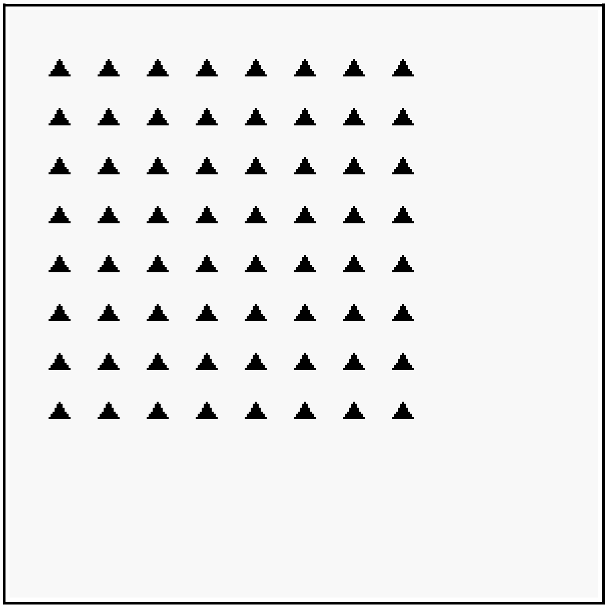
Illustration 19: Equilateral Triangle

Now we will create a program, using the simplest form of the **stamp** statement, to fill the screen with triangles. Program 51 Will do just that. It uses the triangle stamp inside two nested loops to fill the screen.

```
# stamptriangle.kbs - use a stamp to draw many triangles

clg
color black
```

```
for x = 25 to 200 step 25
   for y = 25 to 200 step 25
      stamp x, y, {0, 0, 5, 8.6, -5, 8.6}
   next y
next x
```

Program 51: Fill Screen with Triangles



Sample Output 51: Fill Screen with Triangles

New Concept

```
stamp x, y, {x1, y1, x2, y2 ...}}
stamp x, y, numeric_array[]
stamp x, y, scale, {x1, y1, x2, y2 ...}
stamp x, y, scale, numeric_array[]
stamp x, y, scale, rotate, {x1, y1, x2, y2 ...}
stamp x, y, scale, rotate, numeric_array[]
```

Draw a polygon with it's origin (0,0) at the screen position (x,y). Optionally scale (re-size) it by the decimal scale where 1 is full size. Also you may also rotate the stamp clockwise around it's origin by specifying how far to rotate as an angle expressed in radians (0 to 2π).

New Concept

Radians 0 to 2π

Angles in BASIC-256 are expressed in a unit of measure known as a radian. Radians range from 0 to 2π. A right angle is $\pi/2$ radians and an about face is π radians. You can convert degrees to radians with the formula $r = d / 180 * \pi$.

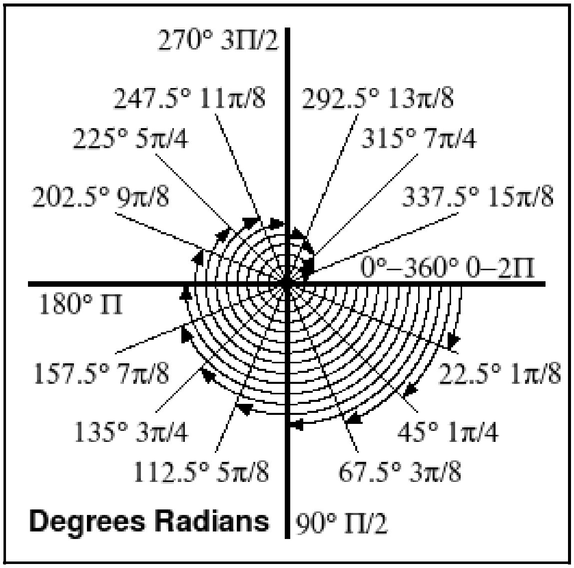
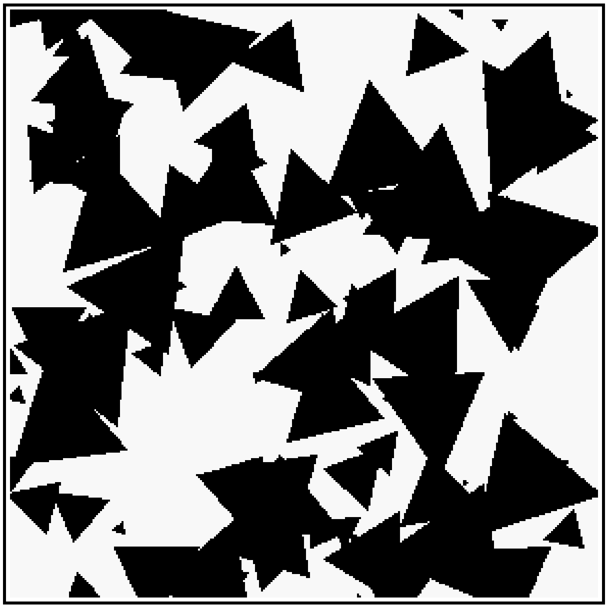


Illustration 20: Degrees and Radians

Let's look at another example of the stamp program. Program 52 used the same isosceles triangle as the last program but places 100 of them at random locations, randomly scaled, and randomly rotated on the screen.

```
# stamptriangle2.kbs - stamp randomly sized and rotated triangles

clg
color black
for t = 1 to 100
  x = rand * graphwidth
  y = rand * graphheight
  s = rand * 7          # scale up to 7 times larger
  r = rand * 2 * pi     # rotate up to 2pi (360 degrees)
  stamp x, y, s, r, {0, 0, 5, 8.6, -5, 8.6}
next t
```

Program 52: One Hundred Random Triangles



Sample Output 52: One Hundred Random Triangles

`pi`

The constant pican be used in expressions so that you do not have to remember the value of π. Π is approximately 3.1415.

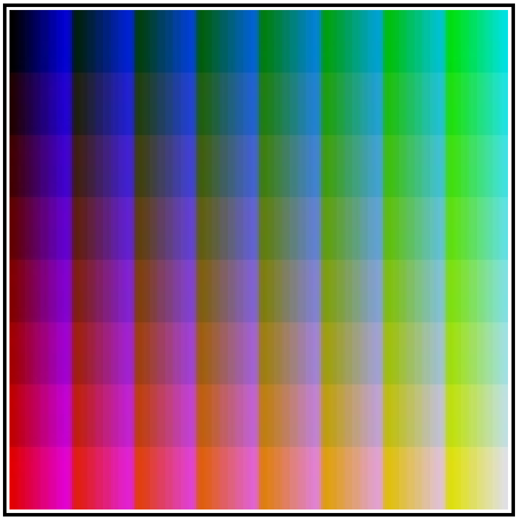
Sixteen Million Different Colors

BASIC-256 will allow you to define up to 16,777,216 unique colors when you draw. The RGB color model adds red (R), green (G), and blue (B) light together to form new colors. If all of the three colors are set to zero the color

Black will be created, if All three colors are set to the maximum value of 255 then the color will be white.

```
# 512colors.kbs - show a few of the 16 million colors
graphsize 256, 256
clg

for r = 0 to 255 step 32
   for g = 0 to 255 step 32
         for b = 0 to 255 step 32
            color rgb(r,g,b)
            rect b/8+g, r, 4, 32
         next b
      next g
next r
```

Program 53: 512 colors of the 16 million



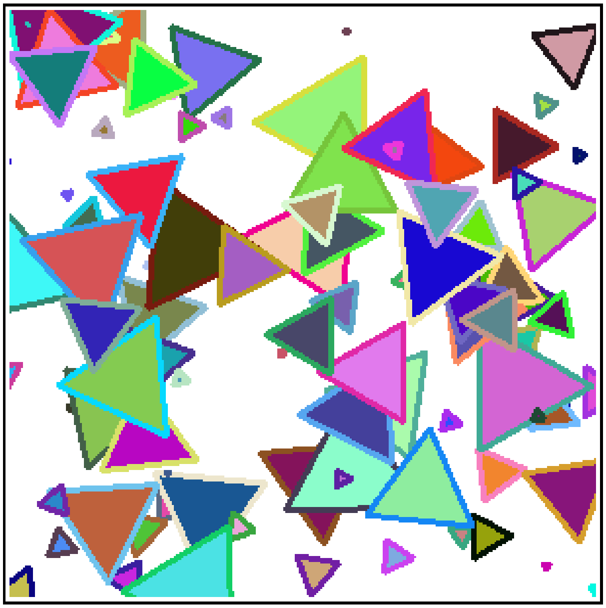
Sample Output 53: 512 colors of the 16 million

rgb(*red*, *green*, *blue*)
rgb(*red*, *green*, *blue*, *alpha*)

The **rgb** function returns a single number that represents a color expressed by the three or four values. The `red`, `blue`, and `green` values represent how much of those colors to include (255-on to 0-off). The optional alpha value represents how transparent the color is (255-solid to 0-totally transparent).

```
# stamptriangle3.kbs - stamp randomly colored, sized and rotated triangles

clg
penwidth 3

for t = 1 to 100
   x = rand * graphwidth
   y = rand * graphheight
   s = rand * 7           # scale up to 7 times larger
   r = rand * 2 * pi      # rotate up to 2pi (360 degrees)
   rpen = rand * 256      # get the RGBparts of a random pen color
   gpen = rand * 256
   bpen = rand * 256
   rbrush = rand * 256   # random brush (fill) color
   gbrush = rand * 256
   bbrush = rand * 256
   color rgb(rpen, gpen, bpen), rgb(rbrush, gbrush, bbrush)
   stamp x, y, s, r, {0, 0, 5, 8.6, -5, 8.6}
next t
```

Program 54: 100 Random Triangles with Random Colors



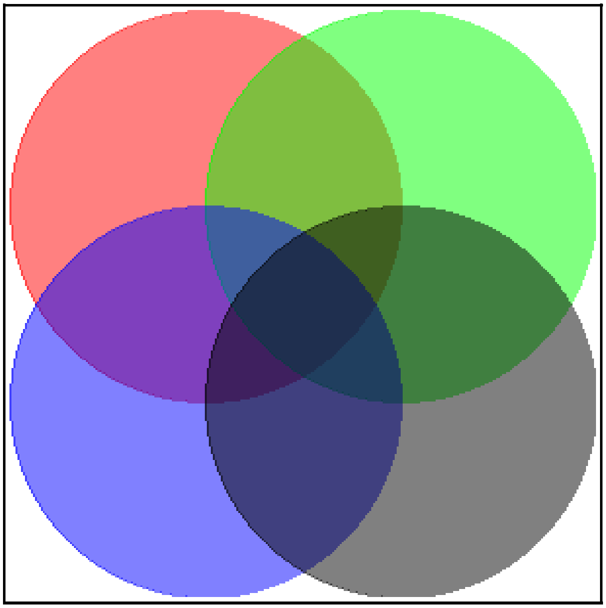
Sample Output 54: 100 Random Triangles with Random Colors

In addition to setting the exact color we want we can also define a color to be transparent. The RGB function has a fourth optional argument to set the alpha (transparency) property of a color. Zero is totally see through, and invisible, while 255 is totally opaque.

```
# transparent.kbs - show the nature of transparent colors
clg white

color rgb(255,0,0,127)
circle 100,100,100

color rgb(0,255,0,127)
circle 200,100,100

color rgb(0,0,255,127)
circle 100,200,100

color rgb(0,0,0,127)
```

```
circle 200,200,100
```

Program 55: Transparent Circles

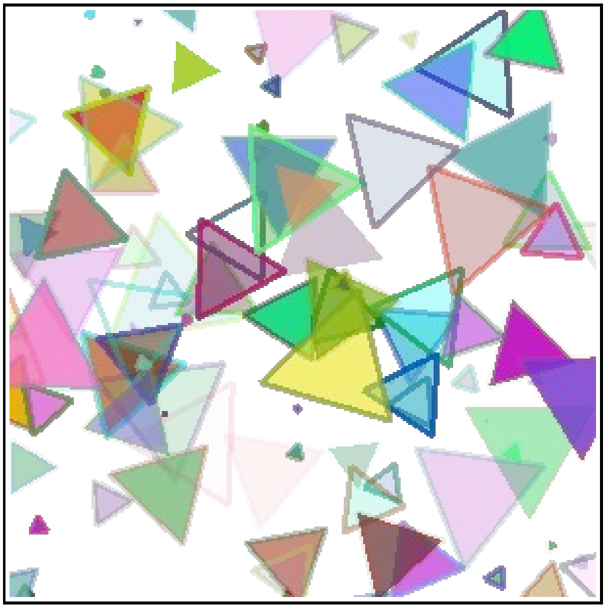


Sample Output 55: Transparent Circles

```
# stamptriangle4.kbs - stamp randomly colored, sized and rotated triangles

clg
penwidth 3

for t = 1 to 100
   x = rand * graphwidth
   y = rand * graphheight
   s = rand * 7          # scale up to 7 times larger
   r = rand * 2 * pi     # rotate up to 2pi (360 degrees)
   rpen = rand * 256     # get the RGBparts of a random pen color
   gpen = rand * 256
   bpen = rand * 256
```

```
    apen = rand * 256
    rbrush = rand * 256   # random brush (fill) color
    gbrush = rand * 256
    bbrush = rand * 256
    abrush = rand * 256
    color rgb(rpen, gpen, bpen, apen), rgb(rbrush, gbrush, bbrush, abrush)
    stamp x, y, s, r, {0, 0, 5, 8.6, -5, 8.6}
next t
```

Program 56: 100 Random Triangles with Random Transparent Colors



Sample Output 56: 100 Random Triangles with Random Transparent Colors

Big Program

Let's send flowers to somebody special. The following program draws a flower using rotation and a stamp.

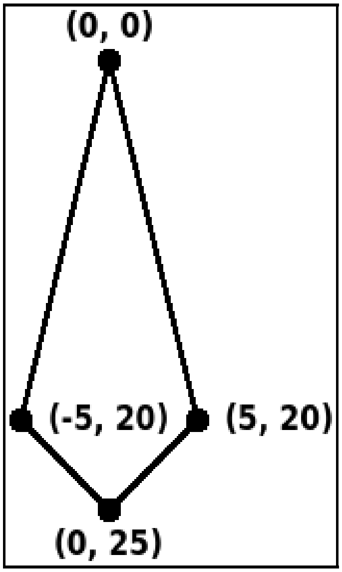
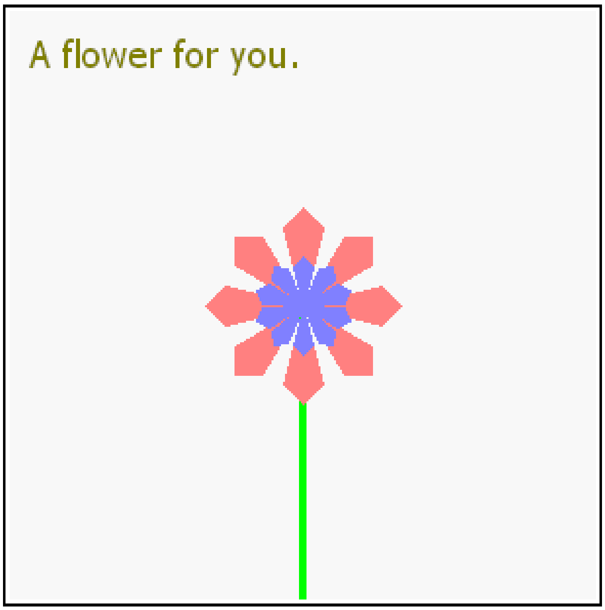


Illustration 21: Big Program - A Flower For You - Flower Petal Stamp

```
# aflowerforyou.kbs - use stamps to draw a flower

clg

color green
rect 148,150,4,150

color rgb(255,128,128)
for r = 0 to 2*pi step pi/4
```

```
   stamp graphwidth/2, graphheight/2, 2, r, {0, 0, 5, 20, 0, 25, -5, 20}
next r

color rgb(128,128,255)
for r = 0 to 2*pi step pi/5
   stamp graphwidth/2, graphheight/2, 1, r, {0, 0, 5, 20, 0, 25, -5, 20}
next r

message = "A flower for you."

color darkyellow
font "Tahoma", 14, 50
text 10, 10, message
say message
```

Program 57: Big Program - A Flower For You



Sample Output 57: Big Program - A Flower For You

Exercises:

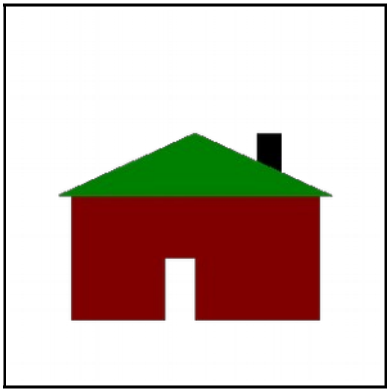
Word Search

```
t n e r a p s n a r t j
k c r l s e u l b h e s
v g p r t r z a g c c g
b h d x a r x i t i f r
a s e m s d e f h g w a
p t e t f h i p p r i p
a o a e h o a a f e t h
e m i p r r n r n e h s
p w a n g g e t q n g i
l r u o t d e u u j i z
g r a p h w i d t h e e
s i p o l y g o n c w f
```

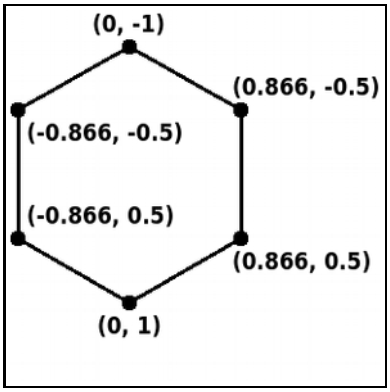
alpha, blue, degrees, font, graphheight, graphics, graphsize, graphwidth, green, pi, point, polygon, radian, red, rgb, stamp, text, transparent, weight

Problems

1. Use two **poly** and one **rect** statements to draw a simple house similar to the one shown below. Your house can be any combination of colors you wish it to be.

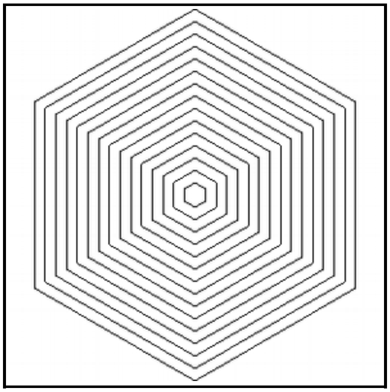
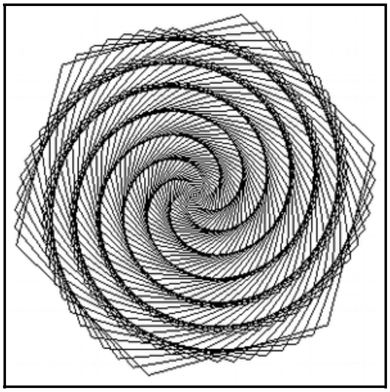


Use the hexagon below as a guide to help you to solve Problems 2 through 4. The sides of the hexagon are one unit long and the origin (0,0) is in the center of the shape.



2. Use a **color** statement with a clear brush and a single **poly** statement to draw a hexagon in the center of the graphics screen with each side 100 pixels long.

3. Rewrite #2 to use a **stamp** statement. Use the scale feature of stamp so that you may draw a hexagon of any size by only changing one number.

	4. Put the **stamp** statement from #3 inside a **for** loop and draw a series of nested hexagons by changing the scale. You may want to experiment with the step clause and with rotating the hexagon at the same time.  

Chapter 10: Functions and Subroutines – Reusing Code.

This chapter introduces the use of Functions and Subroutines. Programmers create subroutines and functions to test small parts of a program, reuse these parts where they are needed, extend the programming language, and simplify programs.

Functions:

A function is a small program within your larger program that does something for you. You may send zero or more values to a function and the function will return one value. You are already familiar with several built in functions like: **rand** and **rgb**. Now we will create our own.

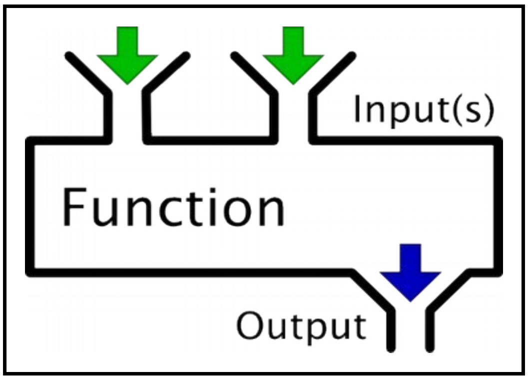


Illustration 22: Block Diagram of a Function

New Concept

```
Function functionname( argument(s) )
   statements
End Function
```

The **Function** statement creates a new block of programming statements and assigns a name to that code. It is recommended that you do not name your function the same name as a variable in your program, as it may cause confusion later.

In the required parenthesis you may also define a list of variables that will receive values from the “calling” part of the program. These variables belong to the function and are not available to the part of the program that calls the function.

A function definition must be closed or finished with an **End Function**. This tells the computer that we are done defining the function.

The value being returned by the function may be set in one of two ways: 1) by using the **return** statement with a value following it or 2) by setting the function name to a value within the function.

New Concept

```
Return value
```

Execute the **return** statement within a function to return a value and send control back to where it was called from.

New Concept

```
end
```

Terminates the program (stop).

```
# minimum.kbs
# minimum function

inputfloat "enter a number ", a
inputfloat "enter a second number ", b

print "the smaller one is ";
print minimum(a,b)
end

function minimum(x,y)
# return the smallest of the two numbers passed
   if x<y then return x
   return y
end function
```

Program 58: Minimum Function

```
enter a number 7
enter a second number 3
the smaller one is 3.0
```

Sample Output 58: Minimum Function

```
# gameroller.kbs
# Game Dice Roller
```

```

print "die roller"
s = get("sides on the die",6)
n = get("number of die", 2)
total = 0
for x = 1 to n
     d = die(s)
     print d
     total = total + d
next x
print "total "+ total
end

function get(message, default)
     # get an integer number
     # if they press enter or type in a non integer then default to another value
     input message + " (default " + default + ") ?" , n
     if typeof(n) <> 1 then n = default
     return n
end function

function die(sides)
     # roll a die and return 1 to sides
     return int(rand*sides)+1
end function
```

Program 59: Game Dice Roller

```
die roller
sides on the die (default 6) ?6
number of die (default 2) ?3
6
3
1
total 10
```

Sample Output 59: Game Dice Roller

In the examples above we have created functions that returned a numeric value. Functions may also be created that return a string value. A string function, like a variable, has a dollar sign after its name to specify that is returns a string.

```
# repeatstring.kbs
# simple string function - make copies

a = "hi"
b = repeat(a,20)
print a
print b
end

function repeat(word,numberoftimes)
     result = ""
     for t = 1 to numberoftimes
          result ;= word
     next t
     return result
end function
```

Program 60: Repeating String Function

```
hi
hihihihihihihihihihihihihihihihihihihihi
```

Sample Output 60: Repeating String Function

Observe in the function samples, above, that variables within a function exist only within the function. If the same variable name is used in the function it DOES NOT change the value outside the function.

Subroutines:

A subroutine is a small subprogram within your larger program that does something specific. Subroutines allow for a single block of code to be used by different parts of a larger program. A subroutine may have values sent to it to tell the subroutine how to react.

Subroutines are like functions except that they do not return a value and that they require the use of the **call** statement to execute them.


```
Subroutine subroutinename( argument(s) )
   statements
End Subroutine
```

The **Subroutine** statement creates a new block of programming statements and assigns a name to that block of code. It is recommended that you do not name your subroutine the same name as a variable in your program, as it may cause confusion later.

In the required parenthesis you may also define a list of variables that will receive values from the "calling" part of the program. These variables are local to the subroutine and are not directly available to the calling program.

A subroutine definition must be closed or finished with an **End Subroutine**. This tells the computer that we are done defining the subroutine.

`Call subroutinename( value(s))`

The **Call** statement tells BASIC-256 to transfer program control to the subroutine and pass the values to the subroutine for processing.

`Return`

Execute the **return** statement within a subroutine to send control back to where it was called from.

This version of the return statement does not include a value to return, as a subroutine does not return a value.

```
# subroutineclock.kbs
# display a comple ticking clock

fastgraphics
font "Tahoma", 20, 100
color blue
rect 0, 0, 300, 300
color yellow
text 0, 0, "My Clock."

while true
     call displaytime()
     pause 1.0
end while

end
```

```

subroutine displaytime()
	color blue
	rect 100, 100, 200, 100
	color yellow
	text 100, 100, padtwo(hour) + ":" + padtwo(minute) + ":" + padtwo(second)
	refresh
end subroutine

function padtwo(x)
	# if x is a single digit then prepend a zero
	if x < 10 then x = "0"+x
	return x
end function
```

Program 61: Subroutine Clock

Sample Output 61: Subroutine Clock


```
hour  or hour()
minute or minute()
second  or second()
month or month()
day  or day()
year or year()
```

The functions **year**, **month**, **day**, **hour**, **minute**, and **second** return the components of the system clock. They allow your program to tell what time it is.

year	Returns the system 4 digit year.
month	Returns month number 0 to 11. 0 – January, 1- February...
day	Returns the day of the month 1 to 28,29,30, or 31.
hour	Returns the hour 0 to 23 in 24 hour format. 0 – 12 AM, 1- 1 AM, … 12 – 12 PM, 13 – 1 PM, 23 – 11 PM ...
minute	Returns the minute 0 to 59 in the current hour.
second	Returns the second 0 to 59 in the current minute.

```
## subroutineclockimproved.kbs
# better ticking clock

fastgraphics
font "Tahoma", 20, 100
clg blue

call displaydate()
while true
     call displaytime()
     pause 1.0
```

```
end while

end

subroutine displaydate()
     # draw over old date
     color blue
     rect 50,50, 200, 100
     # draw new date
     color yellow
     text 50,50, padnumber(month) + "/" + padnumber(day) + "/" + padnumber(year)
     refresh
end subroutine

subroutine displaytime()
     # draw over old time
     color blue
     rect 50,100, 200, 100
     #draw new time
     color yellow
     text 50, 100, padnumber(hour) + ":" + padnumber(minute) + ":" + padnumber(second)
     refresh
end subroutine

function padnumber(n)
     if n < 10 then n = "0" + n
     return n
end function
```

Program 62: Subroutine Clock - Improved

Sample Output: 62: Subroutine Clock - Improved

Using the Same Code in Multiple Programs:

Once a programmer creates a subroutine or function they may want to re-use these blocks of code in other programs. You may copy and paste the code from one program to another but what if you want to make small changes and want the change made to all of your programs. This is where the **include** statement comes in handy.

The include statement tells BASIC-256 at compile time (when you first press the run button) to bring in code from other files. In Program 63 (below) you can see that the functions have been saved out as their own files and included back into the main program.

```
# gamerollerinclude.kbs
# Game Dice Roller

include "diefunction.kbs"
include "getintegerfunction.kbs"

print "die roller with included functions"
s = getinteger("sides on the die",6)
n = getinteger("number of die",2)
total = 0

for x = 1 to n
    d = die(s)
```

```
     print d
     total = total + d
next x
print "total "+ total
end
```

Program 63: Game Dice Roller – With Included Functions

```
# diefunction.kbs
# function to roll a N sided die

function die(sides)
     return int(rand*sides)+1
end function
```

Program 64: Game Dice Roller – die Function

```
# getintegerfunction.kbs
# get an integer number
# if they press enter or type in a non integer then default to another value

function getinteger(message, default)
     input message + " (default " + default + ") ?" , n
     if typeof(n) <> TYPE_INT then n = default
     return n
end function
```

Program 65: Game Dice Roller – getinteger Function

Now that we have split out the functions we can use them in different programs, without having to change the function code or re-typing it.

```
# addingmachine.kbs
# create a nice adding machine

include "getintegerfunction.kbs"

print "adding machine"
print "press stop to end"

total = 0
while true
     a = getinteger("+ ",0)
     total = total + a
     print total
end while
```

Program 66: Adding Machine – Using the inputintegerdefault Function

```
adding machine
press stop to end
+ (default 0) ?6
6
+ (default 0) ?
6
+ (default 0) ?55
61
+ (default 0) ?
```

Sample Output 66: Adding Machine – Using the inputintegerdefault Function

`include "string constant"`

Include code from an external file at compile (when run is clicked).

The file name must be in quotes and can not be a variable or other expression.

Labels, Goto, and Gosub:

This section contains a discussion of labels and how to cause your program to jump to them. These methods are how we used to do it before subroutines and functions were added to the language. **These statements can be used to create ugly and overly complex programs and should be avoided.**

In Program 43 Loop Forever we saw an example of looping forever. This can also be done using a label and a gotostatement.

```
# goto.kbs
top:
print "hi"
goto top
```

Program 67: Goto With a Label

```
hi
hi
hi
hi
... repeats forever
```

Sample Output 67: Goto With a Label

***label*:**

A label allows you to name a place in your program so you may jump to that location later in the program. You may have multiple labels in a single program, but each label can only exist in one place.

A label name is followed with a colon (:); must be at the beginning of a line. The line may contain statements or not that follow the label. Labels must begin with a letter; may contain letters and numbers; and are case-sensitive. Also, you can not use words reserved by the BASIC-256 language when naming labels (see Appendix I), or the names of variables, subroutines and functions.

Examples of valid labels include: top:, far999:, and About:.

goto *label*

The **goto** statement causes the execution to jump to the statement directly following the label.

Subroutines and functions allow us to reuse blocks of code. The gosub statement also allows a programmer to reuse code. The major difference between the two, is that variables in a gosub block are global to the entire program.

Program 68 shows an example of a subroutine that is called three times.

```
# gosub.kbs
# a simple gosub

a = 10
for t = 1 to 3
   print "a equals " + a
   gosub showline
next t
end

showline:
print "------------------"
a = a * 2
return
```

Program 68: Gosub

```
a equals 10
------------------
a equals 20
------------------
a equals 40
------------------
```

Sample Output 68: Gosub

New Concept

```
gosub label
```

The **gosub** statement causes the execution to jump to the subroutine defined by the label

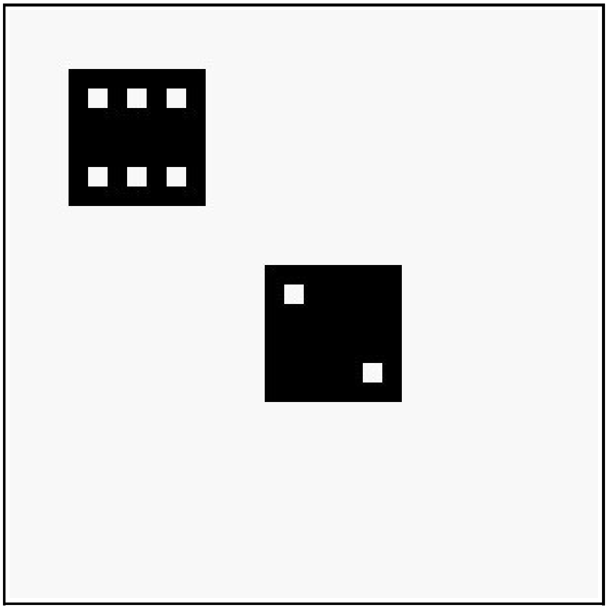
Big Program

In our "Big Program" this chapter, let's make a program to roll two dice, draw them on the screen, and give the total. Let's use an included function to generate the random number of spots and a subroutine to draw the image so that we only have to write it once.

```
# rollgraphicaldice.kbs
# roll two dice graphically

include "diefunction.kbs"

clg
total = 0

roll = die(6)
total = total + roll
call drawdie(30,30, roll)

roll = die(6)
total = total + roll
call drawdie(130,130, roll)

print "you rolled " + total + "."
```

```
 end

 subroutine drawdie(x,y,n)
     # draw 70x70 with dots 10x10 pixels
     # set x,y for top left and n for number of dots
     color black
     rect x,y,70,70
     color white
     # top row
     if n <> 1 then rect x + 10, y + 10, 10, 10
     if n = 6 then rect x + 30, y + 10, 10, 10
     if n >= 4 and n <= 6 then rect x + 50, y + 10, 10, 10
     # middle
     if n = 1 or n = 3 or n = 5 then rect x + 30, y + 30, 10, 10
     # bottom row
     if n >= 4 and n <= 6 then rect x + 10, y + 50, 10, 10
     if n <> 1 then rect x + 50, y + 50, 10, 10
     if n = 6 then rect x + 30, y + 50, 10, 10
end subroutine
```

Program 69: Big Program - Roll Two Dice Graphically



Sample Output 69: Big Program - Roll Two Dice Graphically

Exercises:

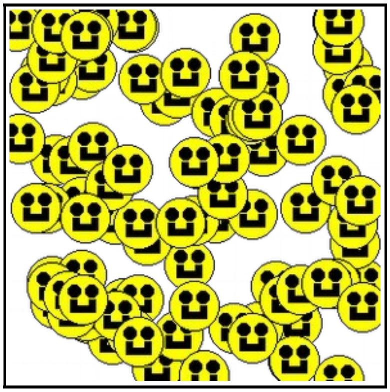
Word Search

```
g o t o d e j j v e q y
k x a w r n x d s q a n
u i d r x i o p i d r o
l n h r g t z c s c e i
k c l e p u j d e p t t
g l e t a o m n h s a c
o u b u l r h e t v n n
s d a r l b f r n h i u
u e l n a u i a e t m f
b m z j c s l e r n r n
e t u n i m e y a o e b
h o u r s o w w p m t n
```

argument, call, day, end, file, function, gosub, goto, hour, include, label, minute, month, parenthesis, return, second, subroutine, terminate, year

Problems

1. Create a subroutine that will accept two numbers representing a point on the screen. Have the routine draw a smiling face with a radius of 20 pixels at that point. You may use circles, rectangles, or polygons as needed. Call that subroutine in a loop 100 times and draw the smiling faces at random locations to fill the screen.



2. Write a program that asks for two points x1, y1 and x2, y2 and displays the formula for the line connecting those two points in slope-intercept format ($y=mx+b$). Create a function that returns the slope (m) of the connecting line using the formula $\frac{y1-y2}{x1-x2}$. Create a second function that returns the y intercept (b) when the x and y coordinates of one of the points and the slope are passed to the function.

```
x1? 1
y1? 1
x2? 3
y2? 2
y = 0.5x + 0.5
```

3. In mathematics the term factorial means the product of consecutive numbers and is represented by the exclamation point. The symbol n! means n * (n-1) * (n-2) * … * 3 * 2 * 1 where n is an integer and 0! is 1 by definition.

Write a function that accepts one number and returns its factorial. Call that new function within a for loop to display 1! to 10!. Your output should look like:

	<pre>1! is 1
	2! is 2
	3! is 6
	4! is 24
	5! is 120
	6! is 720
	7! is 5040
	8! is 40320
	9! is 362880
	10! is 3628800</pre>
	4. A recursive function is a special type of function that calls itself. Knowing that n! = n * (n-1)! and that 0! = 1 rewrite #3 to use a recursive function to calculate a factorial.

Chapter 11: Mouse Control – Moving Things Around.

This chapter will show you how to make your program respond to a mouse. There are two different ways to use the mouse: tracking mode and clicking mode. Both are discussed with sample programs.

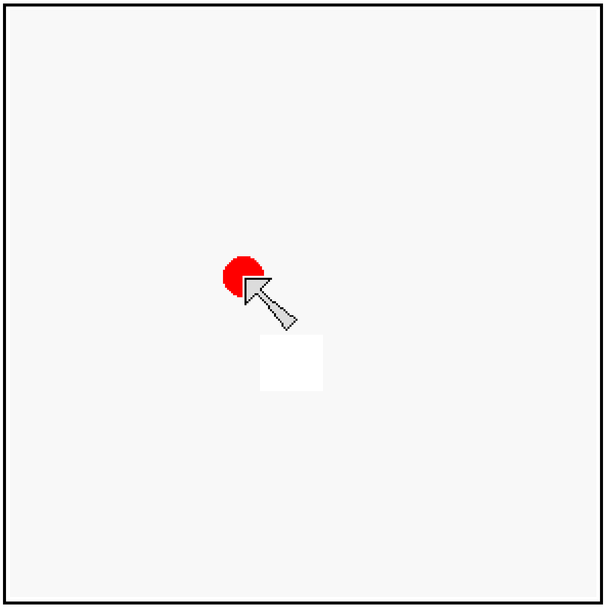
Tracking Mode:

In mouse tracking mode, there are three numeric functions (**mousex**, **mousey**, and **mouseb**) that will return the coordinates of the mouse pointer over the graphics output area. If the mouse is not over the graphics display area then the mouse movements will not be recorded (the last location will be returned).

```
# mousetrack.kbs
# track the mouse with a circle

print "Move the mouse around the graphics window."
print "Click left mouse button to quit."

fastgraphics

# do it over and over until the user clicks left
while mouseb <> MOUSEBUTTON_LEFT
     # erase screen
     clg
     # draw new ball
     color red
     circle mousex, mousey, 10
     refresh
end while

```

```
print "all done."
end
```

Program 70: Mouse Tracking



Sample Output 70: Mouse Tracking


```
mousex or mousex()
mousey or mousey()
mouseb or mouseb()
```

The three mouse functions will return the current location of the mouse as it is moved over the graphics display area. Any mouse motions outside the graphics display area are not recorded, but the last known coordinates will be returned.

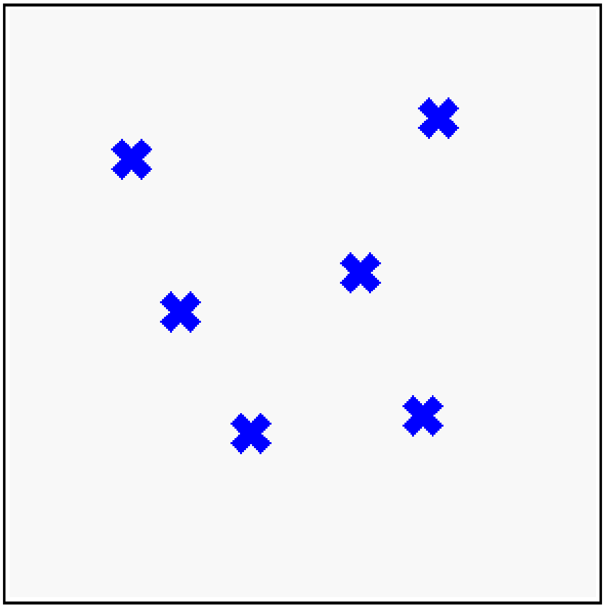
<table>
<tr><td>mousex</td><td>Returns the x coordinate of the mouse pointer position. Ranges from 0 to graphwidth -1.</td></tr>
<tr><td>mousey</td><td>Returns the y coordinate of the mouse pointer position. Ranges from 0 to graphheight -1.</td></tr>
<tr><td>mouseb</td><td>
<table>
<tr><td>0 or
MOUSEBUTTON_NONE</td><td>Returns this value when no mouse button is being pressed.</td></tr>
<tr><td>1 or
MOUSEBUTTON_LEFT</td><td>Returns this value when the "left" mouse button is being pressed.</td></tr>
<tr><td>2 or
MOUSEBUTTON_RIGHT</td><td>Returns this value when the "right" mouse button is being pressed.</td></tr>
<tr><td>4 or
MOUSEBUTTON_CENTER</td><td>Returns this value when the "center" mouse button is being pressed.</td></tr>
</table>
If multiple mouse buttons are being pressed at the same time then the value returned will be the button values added together.
</td></tr>
</table>

Clicking Mode:

The second mode for mouse control is called "Clicking Mode". In clicking mode, the mouse location and the button (or combination of buttons) are stored when the click happens. Once a click is processed by the program a clickclearcommand can be executed to reset the click, so the next one can be recorded.

```
# mouseclick.kbs
# X marks the spot where you click

print "Move the mouse around the graphics window"
print "click left mouse button to mark your spot"
print "click right mouse button to stop."
clg
clickclear
while clickb <> MOUSEBUTTON_RIGHT
     # clear out last click and
     # wait for the user to click a button
     clickclear
     while clickb = MOUSEBUTTON_NONE
          pause .01
     end while
     #
     color blue
     stamp clickx, clicky, 5, {-1,-2, 0,-1, 1,-2, 2,-1, 1,0, 2,1, 1,2, 0,1, -1,2, -2,1, -1,0, -2,-1}
end while
print "all done."
end
```

Program 71: Mouse Clicking



Sample Output 71: Mouse Clicking

New Concept

```
clickx  or clickx()
clicky  or clicky()
clickb or clickb()
```

The values of the three click functions are updated each time a mouse button is clicked when the pointer is on the graphics output area. The last location of the mouse when the last click was received are available from these three functions.

New Concept	`clickclear` The **clickclear** statement resets the **clickx**, **clicky**, and **clickb** functions to zero so that a new click will register when **clickb** <> 0.

Big Program	The big program this chapter uses the mouse to move color sliders so that we can see all 16,777,216 different colors on the screen.

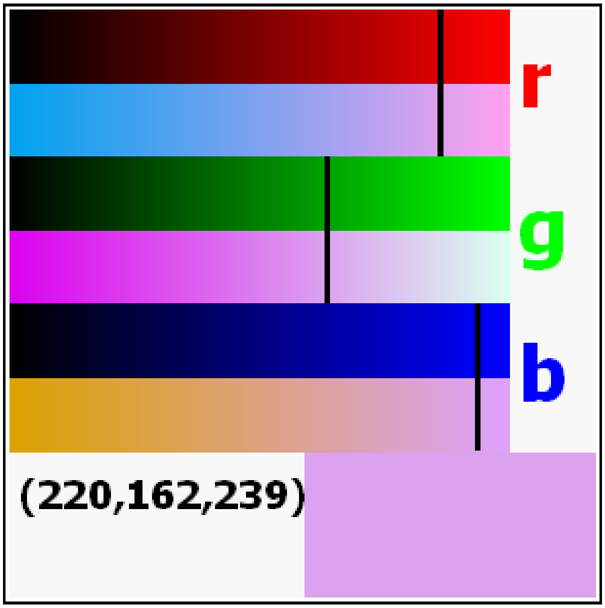
```
# colorchooser.kbs
fastgraphics

print "colorchooser - find a color"
print "click and drag red, green and blue sliders"

# variables to store the color parts
r = 128
g = 128
b = 128

call display(r,g,b)

while true
     # wait for click
     while mouseb = 0
          pause .01
```

```
    end while
    # change color sliders
    # the red slider y range is 0 >= red < 75
    if mousey < 75 then
        r = mousex
        if r > 255 then r = 255
    end if
    # the green slider y range is 75 >= red < 150
    if mousey >= 75 and mousey < 150 then
        g = mousex
        if g > 255 then g = 255
    end if
    # the blue slider y range is 150 >= red < 225
    if mousey >= 150 and mousey < 225 then
        b = mousex
        if b > 255 then b = 255
    end if
    call display(r,g,b)
end while
end

subroutine colorline(r,g,b,x,y)
    # draw part of the color bar the color r,g,b from x,y to x,y+37
    color rgb(r, g, b)
    line x, y, x, y+37
end subroutine

subroutine redsliderbar(r,g,b)
    # draw the red bar from 0,0 to 255,74
    font "Tahoma", 30, 100
    color rgb(255, 0, 0)
    text 260, 0, "r"
    for t = 0 to 255
        # red and red hues
        call colorline(t, 0, 0, t, 0)
        call colorline(t, g, b, t, 38)
    next t
    color black
```

```
     rect r-1, 0, 3, 75
end subroutine

subroutine greensliderbar(r,g,b)
     # draw thegreen bar from 0,75 to 255,149
     font "Tahoma", 30, 100
     color rgb(0, 255, 0)
     text 260, 75, "g"
     for t = 0 to 255
          # green and green hues
          call colorline(0, t, 0, t, 75)
          call colorline(r, t, b, t, 113)
     next t
     # slider
     color black
     rect g-1, 75, 3, 75
end subroutine

subroutine bluesliderbar(r,g,b)
     # draw the blue bar from 0,150 to 255,224
     font "Tahoma", 30, 100
     color rgb(0, 0, 255)
     text 260, 150, "b"
     for t = 0 to 255
          # blue and blue hues
          call colorline(0, 0, t, t, 150)
          call colorline(r, g, t, t, 188)
     next t
     # slider
     color black
     rect b-1, 150, 3, 75
end subroutine

subroutine display(r, g, b)
     clg
     call redsliderbar(r,g,b)
     call greensliderbar(r,g,b)
     call bluesliderbar(r,g,b)
     # draw swatch
```

```
     color rgb(r,g,b)
     rect 151,226,150,75
     refresh
     # draw the RGB values
          color black
     font "Tahoma", 13, 100
     text 5, 235, "(" + r + "," + g + "," + b + ")"
end subroutine
```

Program 72: Big Program - Color Chooser



Sample Output 72: Big Program - Color Chooser

Exercises:

Word Search

```
r f m t x v t x n j
j a a o h k s f o u
n c e y u t c l e c
b e x l e s h i y l
k n z m c s e w l i
c t m o r k u b k c
i e z u n i c o g k
l r p s g s g i m y
c j i e h w l h l m
c x l x m f z a t c
```

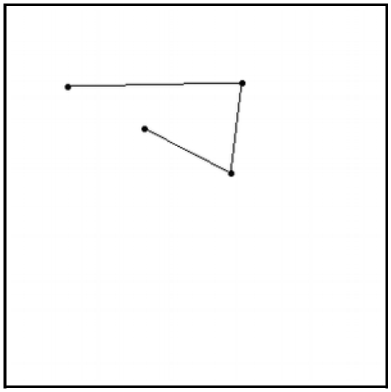
center, clickb, clickclear, clickx, clicky, left, mouseb, mousex, mousey, right

Problems

1. Create a program that will draw a series of connected lines and display the points on the screen as the lines are drawn.

When the left button of the mouse is clicked draw a small circle, print the coordinates, draw a line to the previous coordinates (if not the first point), and remember the point so that it can be the start of the next line. Repeat this until the user clicks stop.

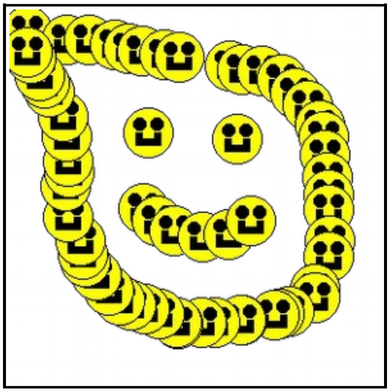
```
46,62
187,59
178,132
108,96
```



2. Create a program that will allow the user to use the mouse like a paintbrush. When the user has the left button depressed then plot a point at that location. To make the line wider you may draw a circle with a radius of 2 or 3.

For extra skill when the user presses the right button make the pen color a random color

3. Use the smiling face subroutine from Problem 1 in the subroutines chapter to make a mouse drawing program with the smile. When the user clicks on a point of the screen draw a face there.

Chapter 12: Keyboard Control – Using the Keyboard to Do Things.

This chapter will show you how to make your program respond to the user when a key is pressed (arrows, letters, and special keys) on the keyboard.

Getting the Last Key Press:

The keyfunction returns the last raw keyboard code generated by the system when a key was pressed. Certain keys (like control-c and function-1) are captured by the BASIC256 window and will not be returned by key. After the last key press value has been returned the function value will be set to zero (0) until another keyboard key has been pressed.

The key values for printable characters (0-9, symbols, letters) are the same as their upper case Unicode values regardless of the status of the caps-lock or shift keys.

```
# readkey.kbs
print "press a key - Q to quit"
do
     k = key
     if k <> 0 then
          if k >=32 and k <= 127 then
               print chr(k) + "=";
          end if
          print k
     end if
until k = asc("Q")
end
```

Program 73: Read Keyboard

```
press a key - Q to quit
A=65
Z=90
M=77
16777248
&=38
7=55
```

Sample Output 73: Read Keyboard

New Concept

```
key
key()
```

The **key** function returns the value of the last keyboard key the user has pressed. Once the key value is read by the function, it is set to zero to denote that no key has been pressed.

Partial List of Keys

ESC= 16777216		Space= 32			
0=48	1=49	2=50	3=51	4=52	5=53
6=54	7=55	8=56	9=57		
A=65	B=66	C=67	D=68	E=69	F=70
G=71	H=72	I=73	J=74	K=75	L=76
M=77	N=78	O=79	P=80	Q=81	R=82
S=83	T=84	U=85	V=86	W=87	X=88
Y=89	Z=90				
Down Arrow= 16777237			Up Arrow= 16777235		
Right Arrow= 16777236			Left Arrow= 16777234		

See http://qt-project.org/doc/qt-4.8/qt.html#Key-enum for a complete list of key values.

Unicode

The Unicode standard was created to assign numeric values to letters or characters for the world's writing systems. There are more than 107,000 different characters defined in the Unicode 5.0 standard.

See: http://www.unicode.org

`asc(`***`expression`***`)`

The **asc** function returns an integer representing the Unicode value of the first character of the string expression.

`chr(`***`expression`***`)`

The **chr** function returns a string, containing a single character with the Unicode value of the integer expression.

Another example of a key press program would be a program to display a letter and to time the user to see how long it took them to press the letter on the keyboard. This program also introduces the **msec** statement that returns the number of milliseconds (1/1000 of a second) that the program has been running.

```
# keymsec.kbs

# get the code for a random character from A-Z
c = asc("A") + int(rand*26)

# display the letter (from the numeric code)
print "press '" + chr(c) + "'"

time = msec                   # get the start time
do                            # wait for the key
     k = key
until k = c
time = msec - time            # calculate how long (in ms)

print "it took you " + (time/1000) + " seconds to find that letter."
```

Program 74: Keyboard Speed Drill

```
press 'C'
it took you 1.833 seconds to find that letter.
```

Sample Output 74: Keyboard Speed Drill


```
msec()
msec
```

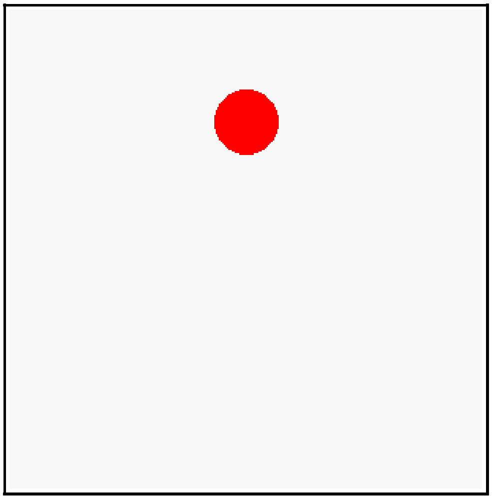
The **msec** function returns the length of time that a program has been running in milliseconds (1/1000 of a second).

How about we look at a more complex example? Program 75 Draws a red ball on the screen and the user can move it around using the keyboard.

```
# keymoveball.kbs
# move a ball on the screen with the keyboard

print "use i for up, j for left, k for right, m for down, q to quit"

fastgraphics
clg

# position of the ball
# start in the center of the screen
x = graphwidth /2
y = graphheight / 2
r = 20  # size of the ball (radius)

# draw the ball initially on the screen
call drawball(x, y, r)

# loop and wait for the user to press a key
while true
     k = key
     if k = asc("I") then
          y = y - r
```

```
            if y < r then y = graphheight - r
            call drawball(x, y, r)
        end if
        if k = asc("J") then
            x = x - r
            if x < r then x = graphwidth - r
            call drawball(x, y, r)
        end if
        if k = asc("K") then
            x = x + r
            if x > graphwidth - r then x = r
            call drawball(x, y, r)
        end if
        if k = asc("M") then
            y = y + r
            if y > graphheight - r then y = r
            call drawball(x, y, r)
        end if
        if k = asc("Q") then exit while
end while
print "all done."
end

subroutine drawball(ballx, bally, ballr)
        clg white
        color red
        circle ballx, bally, ballr
        color rgb(255,100,100)
        circle ballx+.25*ballr, bally+.25*ballr, ballr*.50
        color rgb(255,150,150)
          circle ballx+.25*ballr, bally+.25*ballr, ballr*.30
        color rgb(255,200,200)
        circle ballx+.25*ballr, bally+.25*ballr, ballr*.10
        refresh
end subroutine
```

Program 75: Move Ball



Sample Output 75: Move Ball

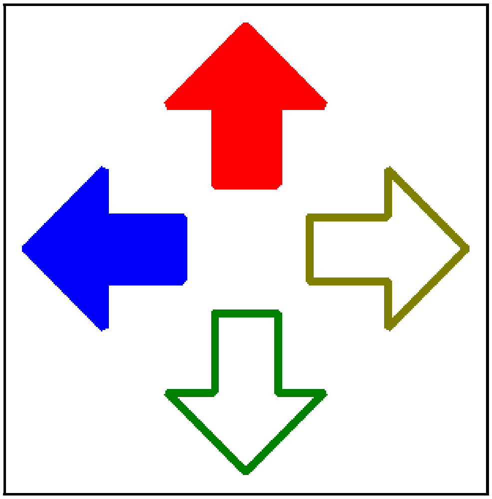
Getting the Currently Pressed Keys

The **key** function in the first half of this chapter returns the last key pressed, even if the user has released the key. We will now see the **keypressed** function that will let us know what keys are being pressed, right now.

```
# keypressarrows.kbs

arrow = { {5, 0}, {10, 5}, {7, 5}, {7, 10}, {3, 10},
{3, 5}, {0, 5}}

ar_down = 16777237
ar_up = 16777235
ar_left = 16777234
ar_right = 16777236
```

```
space = 32

clg white
penwidth 5

print  "press arrow keys on keyboard (even more than one) or space to end"
while not keypressed(space)
     if keypressed(ar_up) then
          color red
     else
          color darkred, white
     endif
     stamp 100,10,10,arrow

     if keypressed(ar_down) then
     color green
     else
          color darkgreen, white
     endif
     stamp 200,290,10,pi,arrow

     if keypressed(ar_left) then
     color blue
     else
          color darkblue, white
     endif
     stamp 10,200,10,1.5*pi,arrow

     if keypressed(ar_right) then
     color yellow
     else
          color darkyellow, white
     endif
     stamp 290,100,10,.5*pi,arrow

end while
```

Program 76: Keys Pressed



Sample Output 76: Keys Pressed


```
keypressed(key_value)
```

The **keypressed** function returns true if the key number is currently being pressed. This statement may be used to see if multiple keys are being pressed at the same time.

See the key function above for a list of common keycodes.

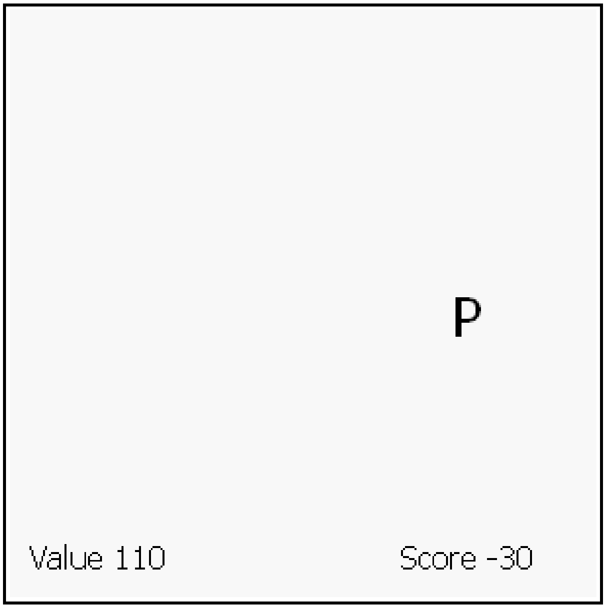
Big Program

The big program this chapter is a game using the keyboard. Random letters are going to fall down the screen and you score points by pressing the key as fast as you can.

```
# fallinglettergame.kbs

speed = .15 # drop speed - lower to make faster
nletters = 10 # letters to play

score = 0
misses = 0
color black

fastgraphics

clg
font "Tahoma", 20, 50
text 20, 80, "Falling Letter Game"
font "Tahoma", 16, 50
text 20, 140, "Press Any Key to Start"
refresh
# clear keyboard and wait for any key to be pressed
k = key
while key = 0
   pause speed
end while

misses = nletters # assume they missed everything
for n = 1 to nletters
   letter = int((rand * 26)) + asc("A")
   x = 10 + rand * 225
   for y = 0 to 250 step 20
```

```
        clg
        # show letter
        font "Tahoma", 20, 50
        text x, y, chr(letter)
        # show score and points
        font "Tahoma", 12, 50
        value = (250 - y)
        text 10, 270, "Value "+ value
        text 200, 270, "Score "+ score
        refresh
        k = key
        if k <> 0 then
            if k = letter then
                score = score + value
                misses-- # didnt miss this one
            else
                score = score - value
            end if
            exit for
        end if
        pause speed
    next y
next n

clg
font "Tahoma", 20, 50
text 20, 40, "Falling Letter Game"
text 20, 80, "Game Over"
text 20, 120, "Score: " + score
text 20, 160, "Misses: " + misses
refresh
end
```

Program 77: Big Program - Falling Letter Game



Sample Output 77: Big Program - Falling Letter Game

Exercises:

Word Search

arrow, asc, capslock, chr, control, key, shift, unicode, keypressed, escape

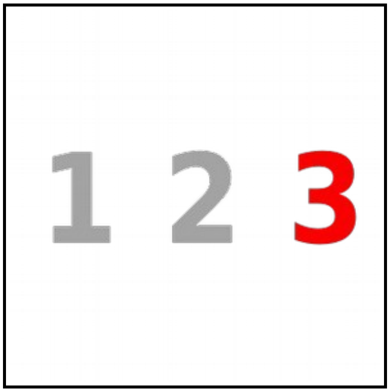
Problems

1. Take Program 74: Keyboard Speed Drill from this chapter and modify it to display ten letters, one at a time, and wait for the user to press that key. Once the user has pressed the correct letters display the total time it took the user.

As an added challenge add logic to count the number of errors and allow a user to retry a letter until they successfully type it.

```
press 'A'
press 'M'
press 'O'
error
press 'U'
press 'X'
press 'V'
press 'K'
press 'C'
press 'Z'
press 'Z'
it took you 15.372 seconds to find
them.
you made 1 errors.
```

2. Create a graphical game like “whack-a-mole” that displays a number on the screen and will wait a random length of time (try 0.5 to 1.5 seconds) for the user to press that number. If they do play a happy sound and display the next, if they miss it or are not fast enough play a sad sound. When they have missed 5 then show them how many they were able to get.



3. Create a piano program using the keys of your keyboard. Wait in a loop so that when the user presses a key the program will play a sound for a short period of time. Assign keys on the keyboard frequencies that correspond to notes on Illustration 10 found on page 52.

4. Use the keypressed function to animate a ball on the screen. You may want to start with Program 75, above.

Chapter 13: Images, WAVs, and Sprites

This chapter will introduce the really advanced multimedia and graphical statements. Saving images to a file, loading them back, playing sounds from WAV files, and really cool animation using sprites.

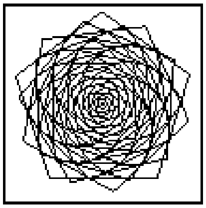
Saving Images to a File:

So far we have seen how to create shapes and graphics using the built in drawing statements. The **imgsave** statement allows you to save your images to one of many standard image formats.

Program 78 Draws a series of pentagons, each a little bigger and rotated to make a beautiful geometric flower. It would be nice to use that image somewhere else. This program creates a PNG (Portable Network Graphics) file that can be used on a Website, presentation, or anywhere else you may want to use it.

```
# 5pointed.kbs
#
graphsize 100,100
clg
color black,clear
for s = 1 to 50 step 2
     stamp 50,50,s,s,{0,-1, .95,-.31, .59,.81, -.59,.81, -.95,-.31}
next s
#
imgsave "5pointed.png", IMAGETYPE_PNG
```

Program 78: Save an Image



Sample Output 78: Save an Image


```
imgsave filename
imgsave filename, type
```

Save the current graphics output to an image file. If the type is not specified the graphic will be saved as a Portable Network Graphic (PNG) file.

Type maybe specified with either a string extension or using a predefined constant.

String	Constant
"png"	IMAGETYPE_PNG
"jpg" or "jpeg"	IMAGETYPE_JPG
"gif"	IMAGETYPE_GIF

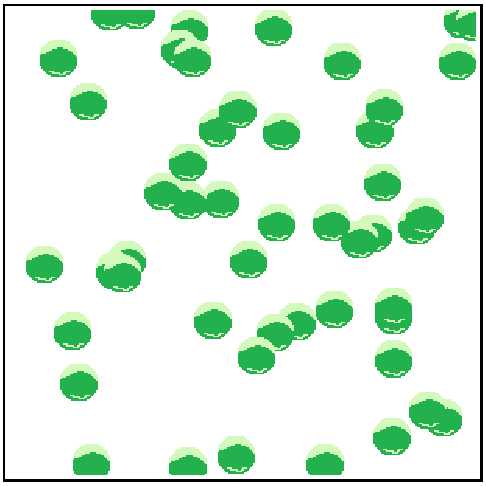
Images From a File:

The **imgload** statement allows you to load a picture from a file and display it in your BASIC-256 programs. These images can be ones you have saved yourself or pictures from other sources.

```
# imgloadball.kbs
```

```
# load an image from a file

clg
for i = 1 to 50
   imgload rand * graphwidth, rand * graphheight, "greenball.png"
next i
```

Program 79: Imgload a Graphic



Sample Output 79: Imgload a Graphic

Program 79 Shows an example of this statement in action. The last argument is the name of a file on your computer. It needs to be in the same folder as the program, unless you specify a full path to it. Also notice that the coordinates (x,y) represent the CENTER of the loaded image and not the top left corner.

Warning	Most of the time you will want to save the program into the same folder that the image or sound file is in BEFORE you run the program. This will set your current working directory so that BASIC-256 can find the file to load.

New Concept	
	imgload ***x*, *y*, *filename*** **imgload** ***x*, *y*, *scale*, *filename*** **imgload** ***x*, *y*, *scale*, *rotation*, *filename*** Read in the picture found in the file and display it on the graphics output area. The values of xand yrepresent the location to place the CENTER of the image. Images may be loaded from many different file formats, including: BMP, PNG, GIF, JPG, and JPEG. Optionally scale (re-size) it by the decimal scale where 1 is full size. Also you may also rotate the image clockwise around it's center by specifying how far to rotate as an angle expressed in radians (0 to 2π).

The **imgload** statement also allows optional scaling and rotation like the **stamp** statement does. Look at Program 80 for an example.

```
# imgloadpicasso.kbs
# show img with rotation and scaling
# photo from http://i988.photobucket.com/albums/af3/fikarvista/picasso_selfport1907.jpg

graphsize 500,500
```

```
clg
for i = 1 to 50
   imgload graphwidth/2, graphheight/2, i/50,
2*pi*i/50, "picasso_selfport1907.jpg"
next i
say "hello Picasso."
```

Program 80: Imgload a Graphic with Scaling and Rotation

Sample Output 80: Imgload a Graphic with Scaling and Rotation

Playing Sounds From a WAV file:

So far we have explored making sounds and music using the **sound** command and text to speech with the **say** statement. BASIC-256 will also play sounds stored in WAV files. The playback of a sound from a WAV file will happen in the background. Once the sound starts the program will continue to the next statement and the sound will continue to play.

```
# numberpopper.kbs
# mp3 files from
# http://www.grsites.com/archive/sounds/

fastgraphics
wavplay "cartoon002.mp3"

speed = .05
for t = 1 to 3
     n = int(rand * 6 + 1)
     for pt = 1 to 200 step 10
          font "Tahoma",pt,100
          clg
          color black
          text 10,10, n
          refresh
          pause speed
     next pt
     speed = speed / 2
next t
# wait for sound to complete
wavwait

wavplay "people055.mp3"
wavwait
end
```

Program 81: Popping Numbers with Sound Effects

New Concept

```
wavplay filename
wavplay ( filename )
wavwait
wavstop
```

The **wavplay** statement loads a wave audio file (.wav) from the current working folder and plays it. The playback will be synchronous meaning that the next statement in the program will begin immediately as soon as the audio begins playing.

Wavstop will cause the currently playing wave audio file to stop the synchronous playback and **wavwait** will cause the program to stop and wait for the currently playing sound to complete.

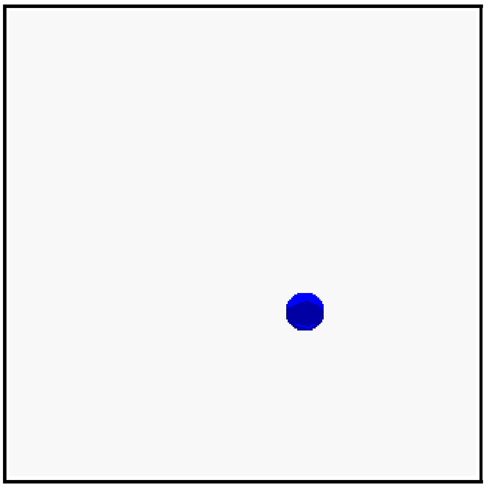
Moving Images - Sprites:

Sprites are special graphical objects that can be moved around the screen without having to redraw the entire screen. In addition to being mobile you can detect when one sprite overlaps (collides) with another. Sprites make programming complex games and animations much easier.

```
# sprite1ball.kbs
# sounds from
# http://www.freesound.org/people/NoiseCollector

clg

spritedim 1

spriteload 0, "blueball.png"
spriteplace 0, 100,100
spriteshow 0

dx = rand * 5 + 5
dy = rand * 5 + 5

```

```
while true
     spritemove 0, dx, dy
     if spritex(0) <= spritew(0)/2 or spritex(0) >= graphwidth - spritew(0)/2 then
          dx = dx * -1
          wavplay "4359__NoiseCollector__PongBlipF4.wav"
     end if
     if spritey(0) <= spriteh(0)/2 or spritey(0) >= graphheight - spriteh(0)/2 then
          dy = dy * -1
          wavplay "4361__NoiseCollector__pongblipA_3.wav"
     endif
     pause .05
end while
```

Program 82: Bounce a Ball with Sprite and Sound Effects



Sample Output 82: Bounce a Ball with Sprite and Sound Effects

As you can see in Program 82 the code to make a ball bounce around the screen, with sound effects, is much easier than earlier programs to do this type of animation. When using sprites we must tell BASIC-256 how many there will be (**spritedim**), we need to set them up (**spriteload** , **spritepoly**, or **spriteplace**), make them visible (**spriteshow**), and then move them around (**spritemove**). In addition to these statements there are functions that will tell us where the sprite is on the screen (**spritex** and **spritey**), how big the sprite is (**spritew** and **spriteh**) and if the sprite is visible (**spritev**).

New Concept

```
spritedim numberofsprites
spritedim ( numberofsprites )
```

The **spritedim** statement initializes, or allocates in memory, places to store the specified number of sprites. You may allocate as many sprites as your program may require but your program may slow down if you create too many sprites.

New Concept

```
spriteload spritenumber, filename
spriteload ( spritenumber, filename )
```

This statement reads an image file (GIF, BMP, PNG, JPG, or JPEG) from the specified path and creates a sprite.

By default the sprite will be placed with its center at 0,0 and it will be hidden. You should move the sprite to the desired position on the screen (**spritemove** or **spriteplace**) and then show it (**spriteshow**).


```
spritehide spritenumber
spritehide ( spritenumber )

spriteshow spritenumber
spriteshow ( spritenumber )
```

The **spriteshow** statement causes a loaded, created, or hidden sprite to be displayed on the graphics output area.

Spritehide will cause the specified sprite to not be drawn on the screen. It will still exist and may be shown again later.


```
spriteplace spritenumber, x, y
spriteplace ( spritenumber, x, y )
```

The **spriteplace** statement allows you to place a sprite's center at a specific location on the graphics output area.

spritemove *spritenumber*, *dx*, dy
spritemove (*spritenumber*, *dx*, dy)

Move the specified sprite xpixels to the right and ypixels down. Negative numbers can also be specified to move the sprite left and up.

A sprite's center will not move beyond the edge of the current graphics output window (0,0) to (**graphwidth**-1, **graphheight**-1).

You may move a hidden sprite but it will not be displayed until you show the sprite using the **showsprite** statement.

spritev(*spritenumber*)

This function returns a true value if a loaded sprite is currently displayed on the graphics output area. False will be returned if it is not visible.


```
spriteh(spritenumber)
spritew(spritenumber)
spritex(spritenumber)
spritey(spritenumber)
```

These functions return various pieces of information about a loaded sprite.

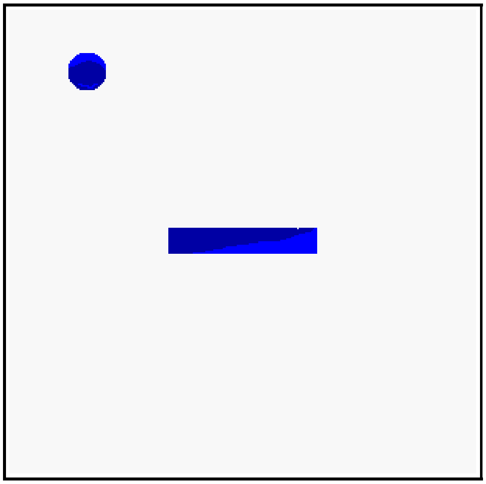
`spriteh`	Returns the height of a sprite in pixels.
`spritew`	Returns the width of a sprite in pixels.
`spritex`	Returns the position on the x axis of the center of the sprite.
`spritey`	Returns the position on the y axis of the center of the sprite.

The second sprite example (Program 83) we now have two sprites. The first one (number zero) is stationary and the second one (number one) will bounce off of the walls and the stationary sprite.

```
# spritebumper.kbs
# show two sprites with collision

color white
rect 0, 0, graphwidth, graphheight

spritedim 2

# stationary bumber
spriteload 0, "paddle.png"
spriteplace 0,graphwidth/2,graphheight/2
spriteshow 0

```

```
# moving ball
spriteload 1, "greenball.png"
spriteplace 1, 50, 50
spriteshow 1
dx = rand * 5 + 5
dy = rand * 5 + 5

while true
   if spritex(1) <=0 or spritex(1) >= graphwidth -1 then

   end if
   if spritey(1) <= 0 or spritey(1) >= graphheight -1 then
      dy = dy * -1
   end if
   if spritecollide(0,1) then
      dy = dy * -1
      print "bump"
   end if
   spritemove 1, dx, dy
   pause .05
end while
```

Program 83: Two Sprites with Collision



Sample Output 83: Two Sprites with Collision

spritecollide(*spritenumber1*, *spritenumber2*)

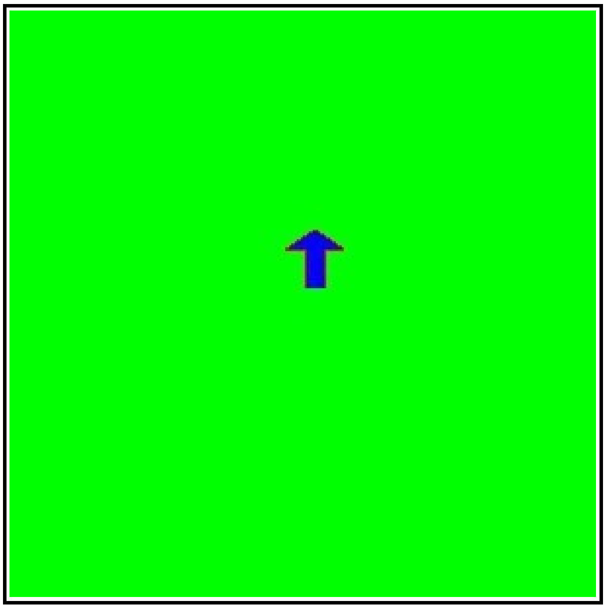
This function returns true of the two sprites collide with or overlap each other.

Sprites may also be created using a polygon as seen in Chapter 9: Custom Graphics – Creating Your Own Shapes. This is accomplished using the **spritepoly** statement.

```
# spritepoly.kbs
# create a sprite from a polygon
# that follows the mouse

spritedim 1
color red, blue
penwidth 1
```

```
spritepoly 0, {15,0, 30,10, 20,10, 20,30, 10,30,
10,10, 0,10}

color green
rect 0,0,graphwidth, graphheight

spriteshow 0
while true
   spriteplace 0, mousex, mousey
   pause .01
end while
```

Program 84: Creating a Sprite From a Polygon



Sample Output 84: Creating a Sprite From a Polygon

New Concept

***spritepoly** spritenumber, { points }*
***spritepoly** (spritenumber, { points })*

spritepoly spritenumber, array_variable
spritepoly (spritenumber, array_variable)

Create a new sprite from the list of points defining a polygon. The top left corner of the polygon should be in the position 0,0 and the sprite's size will be automatically created.

Big Program

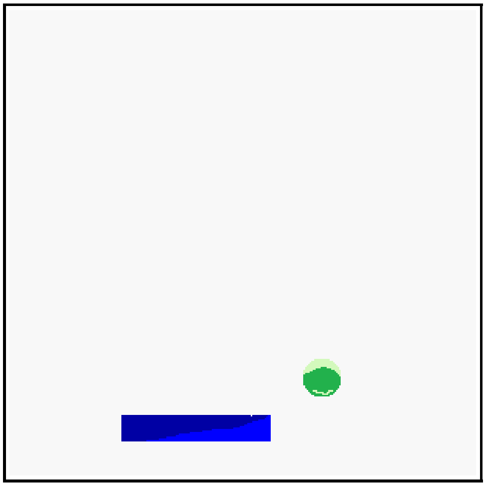
The "Big Program" for this chapter uses sprites and sounds to create a paddle ball game.

```
# sprite_paddleball.kbs
# paddleball game made with sprites
# sounds from http://www.freesound.org/people/NoiseCollector

print "paddleball game"
print "J and K keys move the paddle"
input "Press enter to start >", wait

color white
rect 0, 0, graphwidth, graphheight

spritedim 2
color blue, darkblue
spritepoly 0, {0,0, 80,0, 80,20, 70,20, 70,10, 10,10, 10,20, 0,20}
```

```
spriteplace 0, 100,270
spriteshow 0
spriteload 1, "greenball.png"
spriteplace 1, 100,100
spriteshow 1
penwidth 2

dx = rand * .5 + .25
dy = rand * .5 + .25

bounces = 0

while spritey(1) + spriteh(1) - 5 < spritey(0)
   k = key
   if chr(k) = "K" then
      spritemove 0, 20, 0
   end if
   if chr(k) = "J" then
      spritemove 0, -20, 0
   end if
   if spritecollide(0,1) then
      # bounce back ans speed up
      dy = dy * -1
      dx = dx * 1.1
      bounces = bounces + 1
      wavstop
      wavplay "96633__CGEffex__Ricochet_metal5.wav"
      # move sprite away from paddle
      while spritecollide(0,1)
         spritemove 1, dx, dy
      end while
   end if
   if spritex(1) <=0 or spritex(1) >= graphwidth -1 then
      dx = dx * -1
      wavstop
      wavplay "4359__NoiseCollector__PongBlipF4.wav"
   end if
   if spritey(1) <= 0 then
```

```
      dy = dy * -1
      wavstop
      wavplay "4361__NoiseCollector__pongblipA_3.wav"
   end if
   spritemove 1, dx, dy
   # adjust the speed here
   pause .002
end while

print "You bounced the ball " + bounces + " times."
```

Program 85: Paddleball with Sprites



Sample Output 85: Paddleball with Sprites

Exercises:

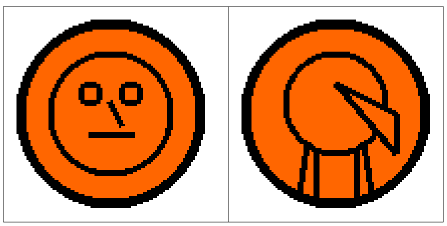
Word Search

```
i s d d i m e n s i o n o z u
s e j i e s c a l e h e w d w
k p v c i r z n r o y d a s o
z j r p m a u o z l u i v p h
a e m i t s t t o m e l w r s
c f v f t a m p c c l l a i e
q o h o t e e i a i g o i t t
w j l i m t l l d w p c t e i
q a o l i e p o a e f e w h r
w n v r i e t v a i t t j i p
q b p p t s s i m d h i s d s
o s v i l t i a r m t r r e c
u u r w o a g o y p s p r p z
h p a p g e y a n d s s e f s
s f t s b k i m g l o a d u o
```

collision, dimension, image, imgload, picture, rotation, scale, spritecollide, spritedim, spritehide, spriteload, spritemove, spriteplace, spritepoly, spriteshow, wavplay, wavstop, wavwait

Problems

1. Write a program to draw a coin, on a graphics window that is 100x100 pixels with a face on it. Save the image as "head.png". Have the same program erase the screen, draw the back side of the coin, and save it as "tail.png". Make the coins your own design.

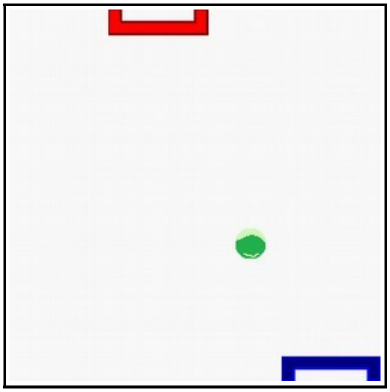


2. Now write a simple coin toss program that displays the results of a coin toss using the images created in program 1. Generate a random number and test if the number is less than .5 then show the heads image otherwise show the tails image.

For an extra challenge make random heads and tails appear on the screen until the user presses a key.

3. Use a program like "Audacity" to record two WAV audio files, one with your voice saying "heads" and the other saying "tails". Add these audio files to the program you wrote in 2.

4. Type in and modify Program 85: Paddleball with Sprites to create a two player "ping-pong" type game. You will need to add a third sprite for the "top" player and assign two keys to move their paddle.



Chapter 14: Printing

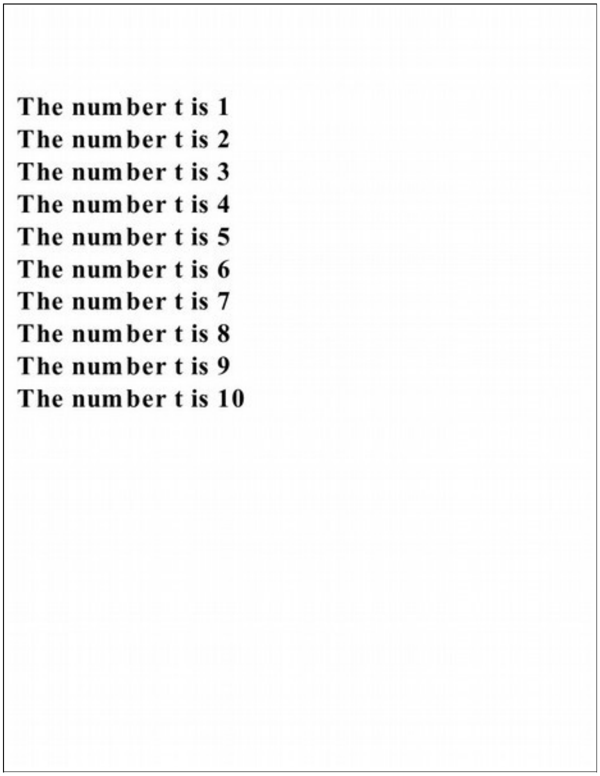
With BASIC-256 you can create output and send it to a printer or to a PDF document. The printer page it treated as if it was a big graphics area that you can draw text, shapes, polygons, stamps, lines, and points using the same graphics statements that you have used in previous chapters.

Turning Printing On and Off

To start printing, all you need to do is turn the printer on with the **print on** statement. Once you are finished creating your page or pages to print execute the **print off** statement.

```
# printpage.kbs
# print a page with text

printer on

x = 100 # start first line 100 pixes down on page

font "Times New Roman", 30, 100
for t = 1 to 10
   text 0, x, "The number t is " + t
   x = x + textheight()
next t

printer off
```

Program 86: Printing a Page with Text



The numbert is 1
The numbert is 2
The numbert is 3
The numbert is 4
The numbert is 5
The numbert is 6
The numbert is 7
The numbert is 8
The numbert is 9
The numbert is 10

Sample Output 86: Printing a Page with Text

New Concept

```
printer on
printeron
```

Turn printing on. Once printing is turned on the graphic statements (**line**, **plot**, **text**, **rect**, **circle**, **poly**, **stamp**, **graphwidth**, **graphheight**, **textwidth**, and **textheight**) now draw on and return information about the printer page.

printer off
printeroff

Ends the current print document. If your output is being send to a print device the document will start printing. If you output is going to a PDF file the file will be written to the specified location.

textwidth(***string*** **)**
textheight()

Returns the width or height of a string in pixels when it is draw on the graphics or printer output area with the **text** statement.

The actual width of the string is returned by **textwidth** but **textheight** returns the standard height in pixels of the currently active font.

You may change the printing destination and properties about the page by selecting "Printing” tab on the “Preferences" window. You may select any configured printer, the size of the page, and the orientation of the page.

Additionally you may select the printer page resolution. Screen resolution, the default, draws on the printer page in a similar manner to how the computer screen is drawn on. In this resolution there are approximately 96 pixels per inch (0.26mm/pixel) . In the High resolution mode you are drawing on the printer page in the printer's native resolution. For most printers and for PDF output that resolution is 1,200 pixels per inch (.021mm/pixel).

Remember that the **font** statement uses the unit of "point" to measure the size of text that is drawn to the graphics display. A point is 1/72 of an inch (3.5mm) so the text will remain constant regardless of the printer mode specified.

All of the examples in this chapter are formatted for Letter (8 ½ x 11 inch) paper in Screen resolution.

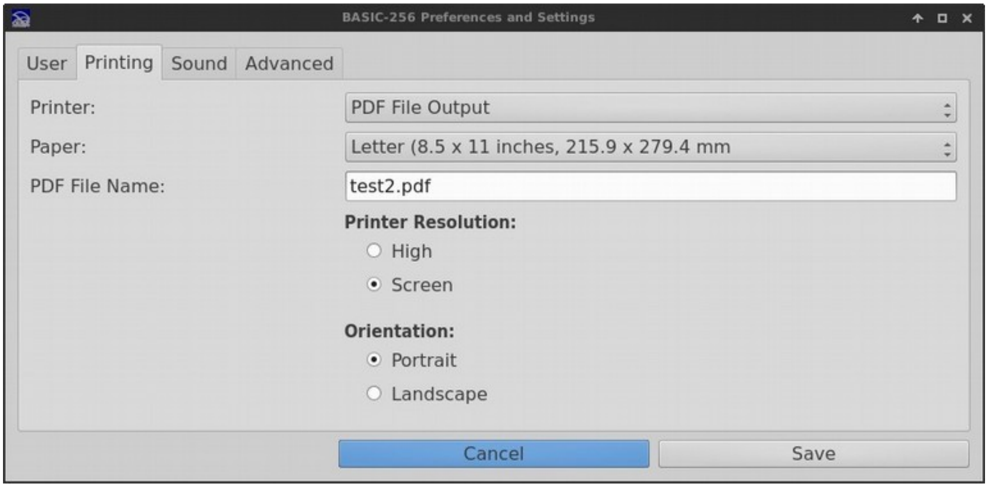
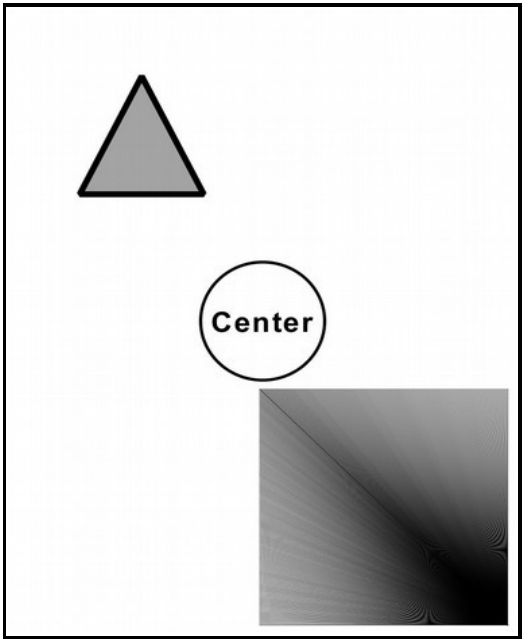


Illustration 23: Preferences – Printing Tab

```
# drawpage.kbs
# Draw on the page

printer on

# put the text in the CENTER of the page
color black
font "Arial", 40, 500
words = "Center"
x = ( graphwidth - textwidth(words) ) / 2
y = ( graphheight - textheight() ) / 2
text x,y,words

# draw a circle around the text
# fill with clear
color black, clear
penwidth 5
```

```
circle graphwidth/2, graphheight/2, 100

# draw a triangle using poly
color black, grey
penwidth 10
poly {200,100, 300,300, 100,300 }


# draw a morier pattern on the page
color black
penwidth 1
for t = 0 to 400 step 3
   line graphwidth, graphheight, graphwidth-400, graphheight-t
   line graphwidth, graphheight, graphwidth-t, graphheight-400
next t

printer off
```

Program 87: Printing a Page with Graphics



Sample Output 87: Printing a Page with Graphics

printer page
printerpage

if you need to print to a new page just execute the **printer page** statement. This will save the current page and all new output will go into the next page.

New Concept

```
printer cancel
printercancel
```

If you have started to print a document but decide you do not want to finish it, the **printer cancel** statement will turn off printing and not output the document.

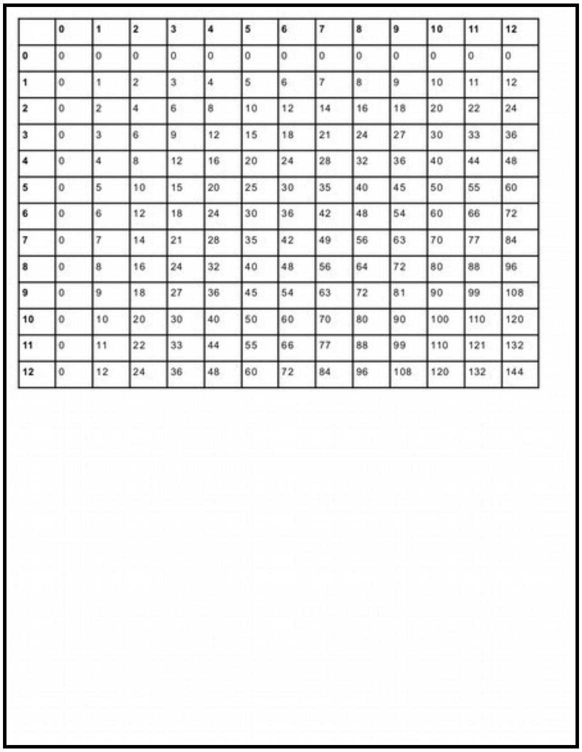
Big Program

The "Big Program" for this chapter uses the printer statements to generate and print a multiplication table.

```
# multtable.kbs
# print a 12x12 multiplication table

printer on
color black
font "Arial", 12, 100

# size of a cell on grid
w = 700/13
h = textheight()*2
#
pad = 5

# draw the grid
penwidth 2
for x = 0 to 14
   line x*w,0,x*w,14*h
```

```
next x
for y = 0 to 14
   line 0,y*h,14*w,y*h
next y

# put the row and column header numbers
font "Arial", 12, 100
for x = 0 to 12
   text (x+1)*w+pad,pad,x
next x
for y = 0 to 12
   text pad,(y+1)*h+pad,y
next y

# put the products
font "Arial", 12, 50
for x = 0 to 12
   for y = 0 to 12
      text (x+1)*w+pad,(y+1)*h+pad,(x*y)
   next y
next x

printer off
```

Program 88: Multiplication Table



	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	10	11	12
2	0	2	4	6	8	10	12	14	16	18	20	22	24
3	0	3	6	9	12	15	18	21	24	27	30	33	36
4	0	4	8	12	16	20	24	28	32	36	40	44	48
5	0	5	10	15	20	25	30	35	40	45	50	55	60
6	0	6	12	18	24	30	36	42	48	54	60	66	72
7	0	7	14	21	28	35	42	49	56	63	70	77	84
8	0	8	16	24	32	40	48	56	64	72	80	88	96
9	0	9	18	27	36	45	54	63	72	81	90	99	108
10	0	10	20	30	40	50	60	70	80	90	100	110	120
11	0	11	22	33	44	55	66	77	88	99	110	121	132
12	0	12	24	36	48	60	72	84	96	108	120	132	144

Sample Output 88: Multiplication Table

Exercises:

Word Search

k	l	a	n	d	s	c	a	p	e
j	f	d	r	e	p	a	p	t	g
p	o	r	t	r	a	i	t	x	a
b	s	g	n	i	t	t	e	s	p
t	h	g	i	e	h	t	x	e	t
r	e	s	o	l	u	t	i	o	n
o	k	p	r	i	n	t	e	r	o
m	a	r	g	i	n	d	f	d	p
g	h	t	d	i	w	t	x	e	t
o	z	c	a	n	c	e	l	x	p

cancel, landscape, margin, page, paper, pdf, portrait, printer, resolution, settings, textheight, textwidth

Problems

1. Take your program from Problem 1 or 2 from the sound and music chapter and have it print the song lyrics on a page after the user types in words to fill in the blanks.

You may need to keep a variable with the line number you are outputting so that you can calculate how far down the page each to start the line.

2. Use the smiling face subroutine you created for Problem 1 from the subroutines chapter to create a page with a smiling face in the four corners and “Smile!” centered on the page.

Chapter 15: Arrays – Collections of Information.

We have used simple string and numeric variables in many programs, but they can only contain one value at a time. Often we need to work with collections or lists of values. We can do this with either one-dimensioned or two-dimensioned arrays. This chapter will show you how to create, initialize, use, and re-size arrays.

One-Dimensional Arrays of Numbers:

A one-dimensional array allows us to create a list in memory and to access the items in that list by a numeric address (called an index). Arrays can contain any type of value (integer, decimal, or string).

Our first example of an array will be using numeric values.

```
# arraynumeric1d.kbs
# one-dimensional numeric array

dim a(4)

a[0] = 100
a[1] = 200
a[2] = a[0] + a[1]

inputfloat "Enter a number> ", a[3]

for t = 0 to 3
     print "a[" + t + "] = " + a[t]
next t
```

Program 89: One-dimensional Numeric Array

```
Enter a number> 63
a[0] = 100
a[1] = 200
a[2] = 300
a[3] = 63.0
```

Sample Output 89: One-dimensional Numeric Array


```
dim variable(items)
dim variable(rows, columns)
dim variable(items) fill expression
dim variable(rows, columns) fill expression
```

The **dim** statement creates an array in the computer's memory the size that was specified in the parenthesis. Sizes (items, rows, and columns) must be integer values greater than or equal to one (1).

The **dim** statement will NOT initialize the elements in the new array unless you specify a fill value. The **fill** clause will assign the value to all elements of the array.


```
variable[index]
variable[rowindex, columnindex]
```

You can use an array reference (variable with index(s) in square brackets) in your program almost anywhere you can use a simple variable. The index or indexes must be integer values between zero (0) and one less than the size used in the dim statement.

It may be confusing, but BASIC-256 uses zero (0) for the first element in an array and the last element has an index one less than the size. Computer people call this a zero-indexed array.

Arrays can also be used to store string values. All you have to do is store a string in the array element.

```
# listoffriends.kbs
# use an array to store any number of names

print "make a list of my friends"
inputinteger "how many friends do you have?", n

dim names(n)
for i = 0 to n-1
     input "enter friend name ?", names[i]
next i

# show the names
cls
print "my friends"
for i = 0 to n-1
     print "friend number ";
     print i + 1;
     print " is " + names[i]
next i

# pick one at random
```

```
x = int(rand * n)
print "The winner is " + names[x]
end
```

Program 90: List of My Friends

```
make a list of my friends
how many friends do you have?3
enter friend name ?Kendra
enter friend name ?Bob
enter friend name ?Susan
 - screen clears -
my friends
friend number 1 is Kendra
friend number 2 is Bob
friend number 3 is Susan
The winner is Kendra
```

Sample Output 90: List of My Friends

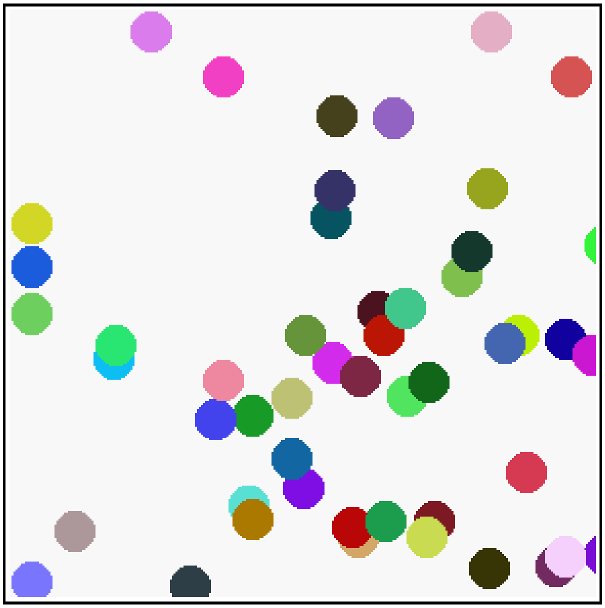
We can use arrays of numbers to draw many balls bouncing on the screen at once. Program 89 uses 5 arrays to store the location of each of the balls, it's direction, and color. Loops are then used to initialize the arrays and to animate the balls. This program also uses the **rgb()** function to calculate and save the color values for each of the balls.

```
# manyballbounce.kbs
# use arrays to keep up with the direction,
# location, and color of many balls on the screen

fastgraphics

r = 10  # size of ball
balls = 50  # number of balls
```

```

# position of the balls - start them all at 0,0
dim x(balls) fill 0
dim y(balls) fill 0

# speed of the balls (set randomly)
dim dx(balls)
dim dy(balls)

# color of the balls (set randomly)
dim colors(balls)

for b = 0 to balls-1
     # speed in x and y directions
     dx[b] = rand * r + 2
     dy[b] = rand * r + 2
     # each ball has it's own color
     colors[b] = rgb(rand*256, rand*256, rand*256)
next b

color green
rect 0,0,300,300

while true
     # erase screen
     clg
     # now position and draw the balls
     for b = 0 to balls -1
          # move ball to new location
          x[b] = x[b] + dx[b]
          y[b] = y[b] + dy[b]
          # if off the edges turn the ball around
          if x[b] < 0 or x[b] > graphwidth then
               dx[b] = dx[b] * -1
          end if
          # if off the top of bottom turn the ball around
          if y[b] < 0 or y[b] > graphheight then
               dy[b] = dy[b] * -1
```

```
            end if
            # draw new ball
            color colors[b]
            circle x[b],y[b],r
      next b
      # update the display
      refresh
      pause .05
end while
```

Program 91: Bounce Many Balls



Sample Output 91: Bounce Many Balls

Assigning Arrays:

We have seen the use of the curly brackets ({}) to play music, draw polygons, and define stamps. The curly brackets can also be used to create and assign an entire array with custom values.

```
# arrayassign.kbs
# using a list of values to create an assign an array

numbers = {56, 99, 145}
names = {"Bob", "Jim", "Susan"}

for i = 0 to 2
   print numbers[i] + " " + names[i]
next i
```

Program 92: Assigning an Array With a List

```
56 Bob
99 Jim
145 Susan
```

Sample Output 92: Assigning an Array With a List


```
variable = {value0, value1, … }
variable = {{v00, v01, …},{v10, v11, …},{v20,
     v21, …},…}
```

A variable will be dimensioned into an array and assigned values (starting with index 0) from a list enclosed in curly braces. The values can be both numbers and strings.

You may assign either a one or two-dimensional array using the braces.

Sound and Arrays:

In Chapter 3 we saw how to use a list of frequencies and durations (enclosed in curly braces) to play multiple sounds at once. The sound statement will also accept a list of frequencies and durations from an array. The array should have an even number of elements; the frequencies should be stored in element 0, 2, 4, …; and the durations should be in elements 1, 3, 5, ….

The sample (Program 93) below uses a simple linear formula to make a fun sonic chirp.

```
# spacechirp.kbs
```

```
# play a spacy sound

# even values 0,2,4... - frequency
# odd values 1,3,5... - duration

# chirp starts at 100hz and increases by 40 for each of the 50 total sounds in list, duration is always 10

dim a(100)
for i = 0 to 98 step 2
   a[i] = i * 40 + 100
   a[i+1] = 10
next i
sound a[]
end
```

Program 93: Space Chirp Sound

Explore	What kind of crazy sounds can you program. Experiment with the formulas to change the frequencies and durations.

Graphics and Arrays:

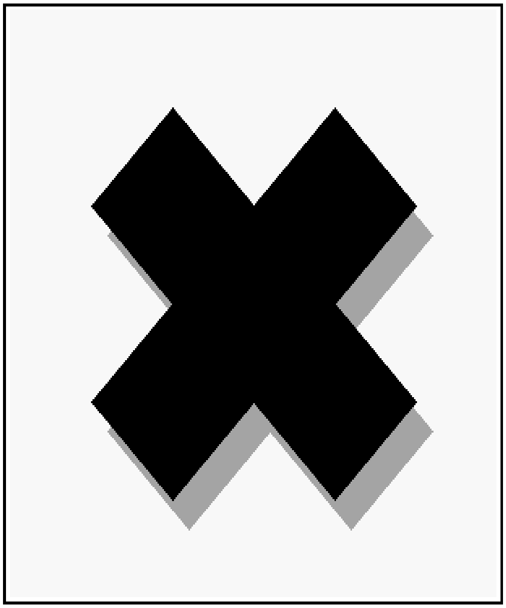
In Chapter 8 we also saw the use of lists for creating polygons and stamps. Arrays may also be used to draw stamps, polygons, and sprites. This may help simplify your code by allowing the same shape to be defined once, stored in an array, and used in various places in your program.

In an array used for a shape, the even elements (0, 2, 4, …) contain the x value for each of the points and the odd element (1, 3, 5, …) contain the y value for the points. The array will have two values for each point in the shape.

In Program 94 we will use the stamp from the mouse chapter to draw a big X with a shadow. This is accomplished by stamping a gray shape shifted in the direction of the desired shadow and then stamping the object that is projecting the shadow.

```
# shadowstamp.kbs
# create a stamp from an array

xmark = {-1, -2, 0, -1, 1, -2, 2, -1, 1, 0, 2, 1, 1, 2, 0, 1, -1, 2, -2, 1, -1, 0, -2, -1}

clg
color grey
stamp 160,165,50,xmark[]
color black
stamp 150,150,50,xmark[]
```

Program 94: Shadow Stamp



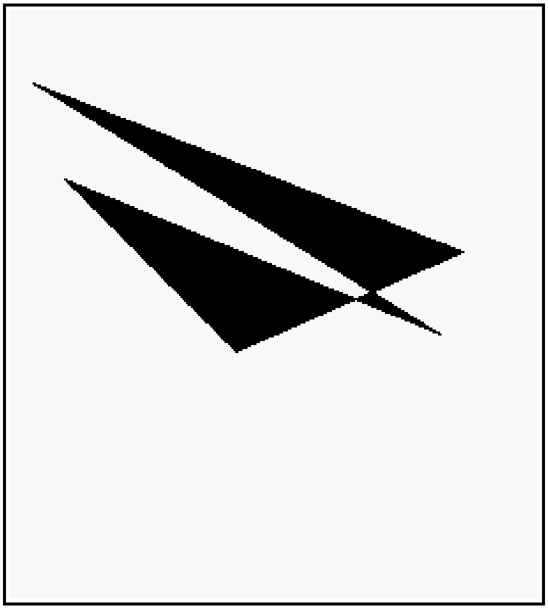
Sample Output 94: Shadow Stamp

Arrays can also be used to create stamps or polygons mathematically. In Program 95 we create an array with 10 elements (5 points) and assign random locations to each of the points to draw random polygons. BASIC-256 will fill the shape the best it can but when lines cross, as you will see, the fill sometimes leaves gaps and holes.

```
# randompoly.kbs
# make an 5 sided random polygon

dim shape(10)

for t = 0 to 8 step 2
   x = 300 * rand
   y = 300 * rand
   shape[t] = x
   shape[t+1] = y
next t

```

```
clg
color black
poly shape[]
```

Program 95: Randomly Create a Polygon



Sample Output 95: Randomly Create a Polygon

Advanced - Two Dimensional Arrays:

So far in this chapter we have explored arrays as lists of numbers or strings. We call these simple arrays one-dimensional arrays because they resemble a line of values. Arrays may also be created with two-dimensions representing rows and columns of data. Program 96 uses both one and two-dimensional arrays to calculate student's average grade.

```
# grades.kbs
# calculate average grades for each student
```

```
# and whole class using a two dimensional array

nstudents = 3 # number of students
nscores = 4 # number of scores per student

dim students(nstudents)
dim grades(nstudents, nscores)

# store the scores as columns and the students as rows
# first student
students[0] = "Jim"
grades[0,0] = 90
grades[0,1] = 92
grades[0,2] = 81
grades[0,3] = 55
# second student
students[1] = "Sue"
grades[1,0] = 66
grades[1,1] = 99
grades[1,2] = 98
grades[1,3] = 88
# third student
students[2] = "Tony"
grades[2,0] = 79
grades[2,1] = 81
grades[2,2] = 87
grades[2,3] = 73

total = 0
for row = 0 to nstudents-1
   studenttotal = 0
   for column = 0 to nscores-1
      studenttotal = studenttotal + grades[row, column]
      total = total + grades[row, column]
   next column
   print students[row] + "'s average is ";
   print studenttotal / nscores
```

```
next row
print "class average is ";
print total / (nscores * nstudents)

end
```

Program 96: Grade Calculator

```
Jim's average is 79.5
Sue's average is 87.75
Tony's average is 80
class average is 82.416667
```

Sample Output 96: Grade Calculator

Really Advanced - Array Sizes and Passing Arrays to Subroutines and Functions:

Sometimes we need to create programming code that would work with an array of any size. If you specify a question mark as a index, row, or column number in the square bracket reference of an array BASIC-256 will return the dimensioned size. In Program 92 we modified Program 91 to display the array regardless of it's length. You will see the special [?] used on line 16 to return the current size of the array.

```
1	# size.kbs
2	# arraylength and passing to subroutine
3
4	print "The Number Array:"
5	number = {77, 55, 33}
6	call showarray(ref(number))
7
8	print "The Random Array:"
9	dim r(5)
```

```
for a = 0 to r[?] - 1
   r[a] = int(rand*10)+1
next a
call showarray(ref(r))
#
end
#
subroutine showarray(a)
   print "has " + a[?] + " elements."
   for i = 0 to a[?] - 1
      print "element " + i + " " + a[i]
   next i
end subroutine
```

Program 97: Get Array Size

```
The Number Array:
has 3 elements.
element 0 77
element 1 55
element 2 33
The Random Array:
has 5 elements.
element 0 7
element 1 5
element 2 1
element 3 9
element 4 10
```

Sample Output 97: Get Array Size

New Concept

```
array[?]
array[?,]
array[,?]
```

The [?] returns the length of a one-dimensional array or the total number of elements (rows * column) in a two-dimensional array. The [?,] reference returns the number of rows and the [,?] reference returns the number of columns of a two dimensional array.

New Concept

```
ref(array)
```

The ref() function is used to pass a reference to an array to a function or subroutine.

If the subroutine changes an element in the referenced array the value in the array will change outside the subroutine or function. Remember this is different behavior than other variables, who's values are copied to new variables within the function or subroutine.

Really Really Advanced - Resizing Arrays:

BASIC-256 will also allow you to re-dimension an existing array. The redim statement will allow you to re-size an array and will preserve the existing data. If the new array is larger, the new elements will be filled with zero (0) or the empty string (""). If the new array is smaller, the values beyond the new size will be truncated (cut off).

```
# redim.kbs

number = {77, 55, 33}
# create a new element on the end
```

```
redim number(4)
number[3] = 22
#
for i = 0 to 3
   print i + " " + number[i]
next i
```

Program 98: Re-Dimension an Array

```
0 77
1 55
2 33
3 22
```

Sample Output 98: Re-Dimension an Array

New Concept	`redim variable(items)`
	`redim variable(rows, columns)`
	
	The **redim** statement re-sizes an array in the computer's memory. Data previously stored in the array will be kept, if it fits.
	
	When resizing two-dimensional arrays the values are copied in a linear manner. Data may be shifted in an unwanted manner if you are changing the number of columns.

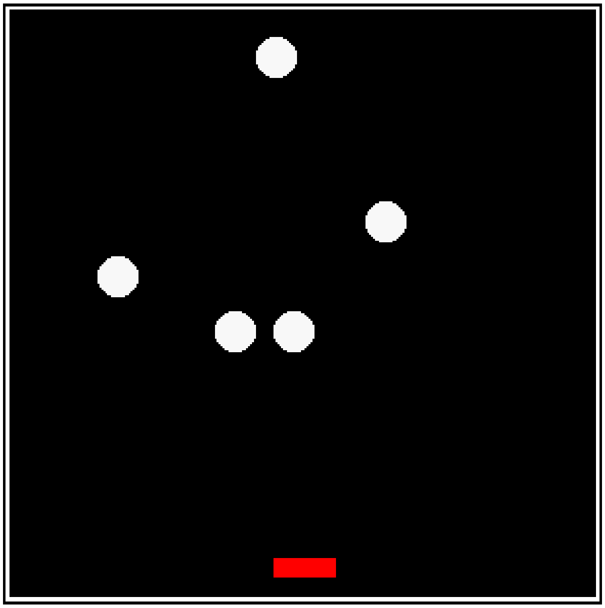
Big Program	The "Big Program" for this chapter uses three numeric arrays to store the positions and speed of falling space debris. You are not playing pong but you are trying to avoid all of them to score points.

```
# spacewarp.kbs
# the falling space debris game

# setup balls and arrays for them
balln = 5
dim ballx(balln)
dim bally(balln)
dim ballspeed(balln)
ballr = 10     # radius of balls

# setup minimum and maximum values
minx = ballr
maxx = graphwidth - ballr
miny = ballr
maxy = graphheight - ballr

# initial score
score = 0

# setup player size, move distance, and location
playerw = 30
playerm = 10
playerh = 10
playerx = (graphwidth - playerw)/2

# setup other variables
keyj = asc("J")     # value for the 'j' key
keyk = asc("K")     # value for the 'k' key
keyq = asc("Q")     # value for the 'q' key
growpercent = .20   # random growth - bigger is faster
speed = .15    # the lower the faster

print "spacewarp - use j and k keys to avoid the falling space debris"
print "q to quit"

fastgraphics
```

```

# setup initial ball positions and speed
for n = 0 to balln-1
   bally[n] = miny
   ballx[n] = int(rand * (maxx-minx)) + minx
   ballspeed[n] = int(rand * (2*ballr)) + 1
next n

more = true
while more
   pause speed
   score = score + 1

   # clear screen
   color black
   rect 0, 0, graphwidth, graphheight

   # draw balls and check for collission
   color white
   for n = 0 to balln-1
      bally[n] = bally[n] + ballspeed[n]
      if bally[n] > maxy then
         # ball fell off of bottom - put back at top
         bally[n] = miny
         ballx[n] = int(rand * (maxx-minx)) + minx
         ballspeed[n] = int(rand * (2*ballr)) + 1
      end if
      circle ballx[n], bally[n], ballr
      if ((bally[n]) >= (maxy-playerh-ballr)) and
((ballx[n]+ballr) >= playerx) and ((ballx[n]-ballr)
<= (playerx+playerw)) then more = false
   next n

   # draw player
   color red
   rect playerx, maxy - playerh, playerw, playerh
   refresh

   # make player bigger
```

```
   if (rand<growpercent) then playerw = playerw + 1

   # get player key and move if key pressed
   k = key
   if k = keyj then playerx = playerx - playerm
   if k = keyk then playerx = playerx + playerm
   if k = keyq then more = false

   # keep player on screen
   if playerx < 0 then playerx = 0
   if playerx > graphwidth - playerw then playerx = graphwidth - playerw

end while

print "score " + string(score)
print "you died."
end
```

Program 99: Big Program - Space Warp Game



Sample Output 99: Big Program - Space Warp Game

Exercises:

Word Search

```
a t d v i t f p a u
y o y n s z o n c b
e r d q a i m n o e
o e o s c o l u m n
x e d m c z d y v i
c o l l e c t i o n
a r r a y m n h z y
y h t s i l e g d f
d i m e n s i o n l
y j n f z r o w l t
```

array, collection, column, dimension, index, list, memory, row

Problems

1. Ask the user for how many numbers they want to add together and display the total. Create an array of the user chosen size, prompt the user to enter the numbers and store them in the array. Once the numbers are entered loop through the array elements and print the total of them.

2. Add to Problem 1 logic to display the average after calculating the total.

3. Add to Problem 1 logic to display the minimum and the maximum values. To calculate the minimum: 1) copy the first element in the array into a variable; 2) compare all of the remaining elements to the variable and if it is less than the saved value then save the new minimum.

4. Take the program from Problem 2 and 3 and create functions

	to calculate and return the minimum, maximum, and average. Pass the array to the function and use the array length operator to make the functions work with any array passed.
	
	5. Create a program that asks for a sequence of numbers, like in Problem 1. Once the user has entered the numbers to the array display a table of each number multiplied by each other number. Hint: you will need a loop nested inside another loop.
	
	`n> 5`
	`number 0> 4`
	`number 1> 7`
	`number 2> 9`
	`number 3> 12`
	`number 4> 45`
	`16 28 36 48 180`
	`28 49 63 84 315`
	`36 63 81 108 405`
	`48 84 108 144 540`
	`180 315 405 540 2025`

Chapter 16: Mathematics – More Fun With Numbers.

In this chapter we will look at some additional mathematical operators and functions that work with numbers. Topics will be broken down into four sections: 1) new operators; 2) new integer functions, 3) new floating-point functions, and 4) trigonometric functions.

New Operators:

In addition to the basic mathematical operations we have been using since the first chapter, there are three more operators in BASIC-256. Operations similar to these three operations exist in most computer languages. They are the operations of modulo, integer division, and power.

Operation	Operator	Description
Modulo	%	Return the remainder of an integer division.
Integer Division	\	Return the whole number of times one integer can be divided into another.
Power	^	Raise a number to the power of another number.

Modulo Operator:

The modulo operation returns the remainder part of integer division. When you do long division with whole numbers, you get a remainder – that is the same as the modulo.

```
# modulo.kbs
```

```
inputinteger "enter a number ", n
if n % 2 = 0 then print "divisible by 2"
if n % 3 = 0 then print "divisible by 3"
if n % 5 = 0 then print "divisible by 5"
if n % 7 = 0 then print "divisible by 7"
end
```

Program 100: The Modulo Operator

```
enter a number 10
divisible by 2
divisible by 5
```

Sample Output 100: The Modulo Operator

New Concept

expression1 % expression2

The Modulo (%) operator performs integer division of expression1 divided by expression2and returns the remainder of that process.

If one or both of the expressions are not integer values (whole numbers) they will be converted to an integer value by truncating the decimal (like in the int()function) portion before the operation is performed.

You might not think it, but the modulo operator (%) is used quite often by programmers. Two common uses are; 1) to test if one number divides into another (Program 100) and 2) to limit a number to a specific range (Program 101).

```
# moveballmod.kbs
```

```
# rewrite of moveball.kbs using the modulo operator to wrap the ball around the screen

print "use i for up, j for left, k for right, m for down, q to quit"

fastgraphics
clg
ballradius = 20

# position of the ball
# start in the center of the screen
x = graphwidth /2
y = graphheight / 2

# draw the ball initially on the screen
call drawball(x, y, ballradius)

# loop and wait for the user to press a key
while true
   k = key
   if k = asc("I") then
      # y can go negative, + graphheight keeps it positive
      y = (y - ballradius + graphheight) % graphheight
      call drawball(x, y, ballradius)
   end if
   if k = asc("J") then
      x = (x - ballradius + graphwidth) % graphwidth
      call drawball(x, y, ballradius)
   end if
   if k = asc("K") then
      x = (x + ballradius) % graphwidth
      call drawball(x, y, ballradius)
   end if
   if k = asc("M") then
      y = (y + ballradius) % graphheight
      call drawball(x, y, ballradius)
```

```
   end if
   if k = asc("Q") then end
end while

subroutine drawball(bx, by, br)
   color white
   rect 0, 0, graphwidth, graphheight
   color red
   circle bx, by, br
   refresh
end subroutine
```

Program 101: Move Ball - Use Modulo to Keep on Screen

Integer Division Operator:

The Integer Division (\) operator does normal division but it works only with integers (whole numbers) and returns an integer value. As an example, 13 divided by 4 is 3 remainder 1 – so the result of the integer division is 3.

```
# integerdivision.kbs
inputinteger "dividend ", dividend
inputinteger "divisor ", divisor
print dividend + " / " + divisor + " is ";
print dividend \ divisor;
print "r";
print dividend % divisor;
```

Program 102: Check Your Long Division

```
dividend 43
divisor 6
43 / 6 is 7r1
```

Sample Output 102: Check Your Long Division

expression1 \ expression2

The Integer Division (\) operator performs division of expression1 / expression2 and returns the whole number of times expression1 goes into expression2.

If one or both of the expressions are not integer values (whole numbers), they will be converted to an integer value by truncating the decimal (like in the int() function) portion before the operation is performed.

Power Operator:

The power operator will raise one number to the power of another number.

```
# power.kbs
for t = 0 to 16
    print "2 ^ " + t + " = ";
    print 2 ^ t
next t
```

Program 103: The Powers of Two

```
2 ^ 0 = 1
2 ^ 1 = 2
2 ^ 2 = 4
2 ^ 3 = 8
2 ^ 4 = 16
2 ^ 5 = 32
2 ^ 6 = 64
2 ^ 7 = 128
2 ^ 8 = 256
2 ^ 9 = 512
2 ^ 10 = 1024
```

```
2 ^ 11 = 2048
2 ^ 12 = 4096
2 ^ 13 = 8192
2 ^ 14 = 16384
2 ^ 15 = 32768
2 ^ 16 = 65536
```

Sample Output 103: The Powers of Two

expression1 ^ expression2

The Power (^) operator raises expression1to the expression2 power.

The mathematical expression $a=b^c$ would be written in BASIC-256 as a = b ^ c.

New Integer Functions:

The three new integer functions in this chapter all deal with how to convert strings and floating-point numbers to integer values. All three functions handle the decimal part of the conversion differently.

In the int()function the decimal part is just thrown away, this has the same effect of subtracting the decimal part from positive numbers and adding it to negative numbers. This can cause troubles if we are trying to round and there are numbers less than zero (0).

The ceil()and floor()functions sort of fix the problem with int(). Ceil() always adds enough to every floating-point number to bring it up to the next whole number while floor(0) always subtracts enough to bring the floating-point number down to the closest integer.

We have been taught to round a number by simply adding 0.5 and drop the decimal part. If we use the int() function, it will work for positive numbers but not for negative numbers. In BASIC-256 to round we should always use a formula like $a = floor(b+0.5)$.

New Concept	Function	Description
	`int(expression)`	Convert an expression (string, integer, or decimal value) to an integer (whole number). When converting a floating-point value the decimal part is truncated (ignored). If a string does not contain a number a zero is returned.
	`ceil(expression)`	Converts a floating-point value to the next highest integer value.
	`floor(expression)`	Converts a floating-point expression to the next lowers integer value. You should use this function for rounding $a = floor(b+0.5)$.

```
# intceilfloor.kbs
for t = 1 to 10
   n = rand * 100 - 50
   print n;
   print "  int=" + int(n);
   print "  ceil=" + ceil(n);
   print "  floor=" + floor(n)
next t
```

Program 104: Difference Between Int, Ceiling, and Floor

```
-46.850173  int=-46  ceil=-46  floor=-47
```

```
-43.071987  int=-43  ceil=-43  floor=-44
23.380133  int=23  ceil=24  floor=23
4.620722  int=4  ceil=5  floor=4
3.413543  int=3  ceil=4  floor=3
-26.608505  int=-26  ceil=-26  floor=-27
-18.813465  int=-18  ceil=-18  floor=-19
7.096065  int=7  ceil=8  floor=7
23.482759  int=23  ceil=24  floor=23
-45.463169  int=-45  ceil=-45  floor=-46
```

Sample Output 104: Difference Between Int, Ceiling, and Floor

New Floating-Point Functions:

The mathematical functions that wrap up this chapter are ones you may need to use to write some programs. In the vast majority of programs these functions will not be needed.

Function	Description
`abs(`*`expression`*`)`	Converts a floating-point or integer expression to an absolute value.
`log(`*`expression`*`)`	Returns the natural logarithm (base e) of a number.
`log10(`*`expression`*`)`	Returns the base 10 logarithm of a number.

Advanced - Trigonometric Functions:

Trigonometry is the study of angles and measurement. BASIC-256 includes support for the common trigonometric functions. Angular measure is done in radians (0-2p). If you are using degrees (0-360) in your programs you must convert to use the "trig" functions.

New Concept

Function	Description
`cos(expression)`	Return the cosine of an angle.
`sin(expression)`	Return the sine of an angle.
`tan(expression)`	Return the tangent of an angle.
`degrees(expression)`	Convert Radians (0 – 2π) to Degrees (0-360).
`radians(expression)`	Convert Degrees (0-360) to Radians (0 – 2π).
`acos(expression)`	Return the inverse cosine.
`asin(expression)`	Return the inverse sine.
`atan(expression)`	Return the inverse tangent.

The discussion of the first three functions will refer to the sides of a right triangle. Illustration 24 shows one of these with it's sides and angles labeled.

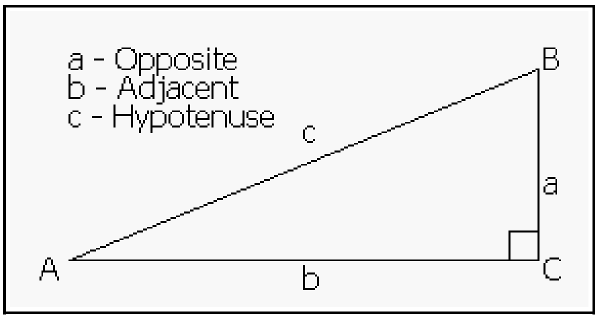


Illustration 24: Right Triangle

Cosine:

A cosine is the ratio of the length of the adjacent leg over the length of the hypotenuse $\cos A = \frac{b}{c}$. The cosine repeats itself every 2π radians and has a range from -1 to 1. Illustration 24 graphs a cosine wave from 0 to 2π radians.

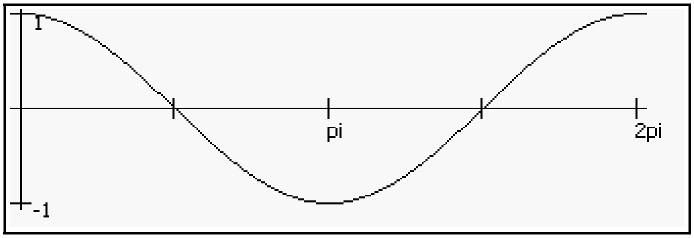


Illustration 25: Cos() Function

Sine:

The sine is the ratio of the opposite leg over the hypotenuse $\sin A = \frac{a}{c}$. The sine repeats itself every 2π radians and has a range from -1 to 1. You have seen diagrams of sine waves in Chapter 3 as music was discussed.

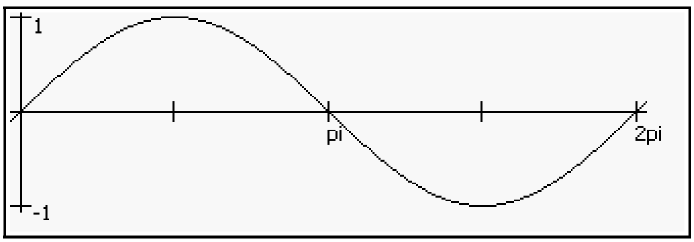


Illustration 26: Sin() Function

Tangent:

The tangent is the ratio of the adjacent side over the opposite side $\tan A = \frac{a}{b}$. The tangent repeats itself every π radians and has a range from $-\infty$ to ∞. The tangent has this range because when the angle approaches ½π radians the opposite side gets very small and will actually be zero when the angle is ½π radians.

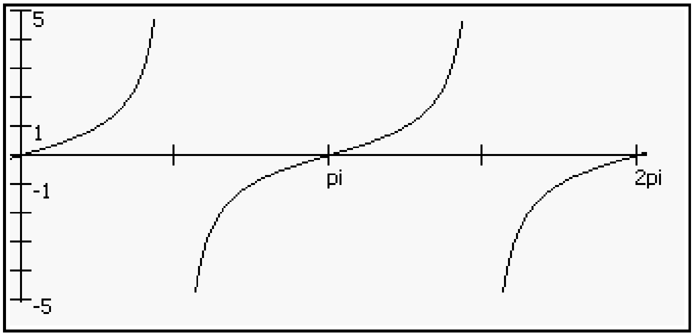


Illustration 27: Tan() Function

Degrees Function:

The **degrees**() function does the quick mathematical calculation to convert an angle in radians to an angle in degrees. The formula used is $degrees = radians / 2\pi * 360$.

Radians Function:

The **radians**() function will convert degrees to radians using the formula $radians = degrees / 360 * 2\pi$. Remember all of the trigonometric functions in BASIC-256 use radians and not degrees to measure angles.

Inverse Cosine:

The inverse cosine function **acos**() will return an angle measurement in radians for the specified cosine value. This function performs the opposite of the cos()function.

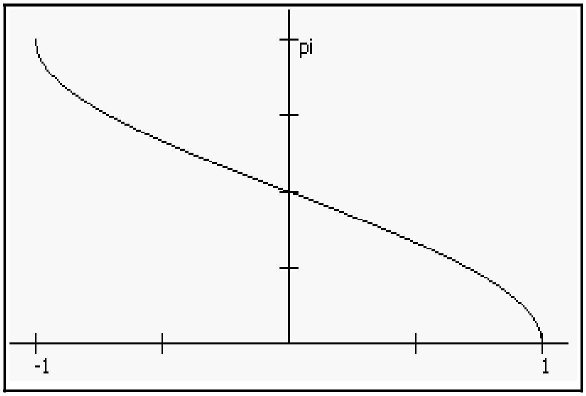


Illustration 28: Acos() Function

Inverse Sine:

The inverse sine function **asin**() will return an angle measurement in radians for the specified sine value. This function performs the opposite of the sin () function.

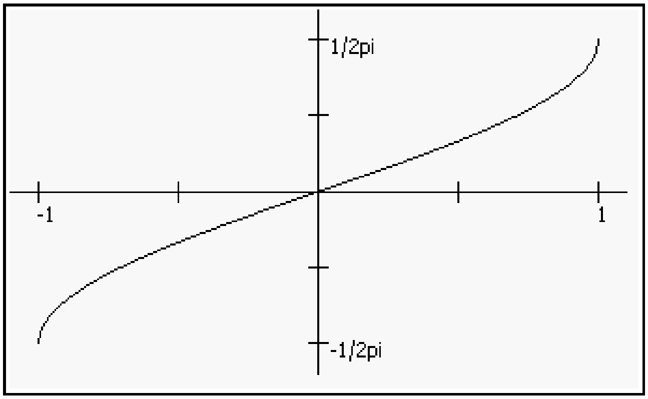


Illustration 29: Asin() Function

Inverse Tangent:

The inverse tangent function **atan**() will return an angle measurement in radians for the specified tangent value. This function performs the opposite of the **tan**() function.

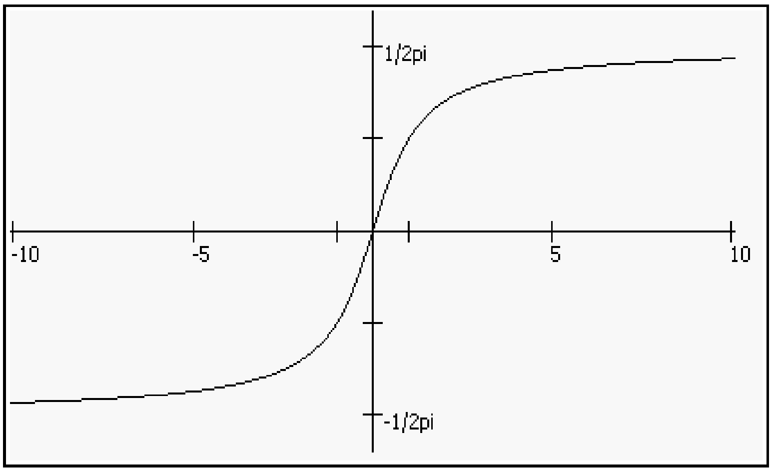


Illustration 30: Atan() Function

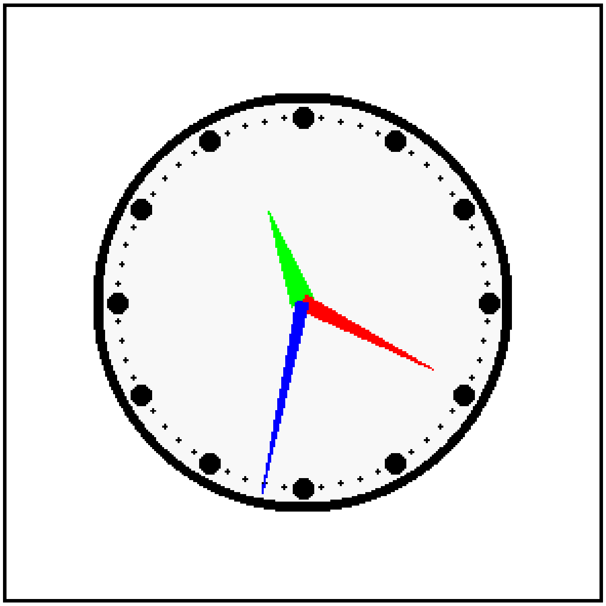
Big Program

The big program this chapter allows the user to enter two positive whole numbers and then performs long division. This program used logarithms to calculate how long the numbers are, modulo and integer division to get the individual digits, and is generally a very complex program. Don't be scared or put off if you don't understand exactly how it works, yet.

```
# handyclock.kbs

fastgraphics

while true
   clg
   # draw outline
   color black, white
   penwidth 5
   circle 150,150,105
   # draw the 60 marks (every fifth one make it larger)
   color black
   penwidth 1
   for m = 0 to 59
      a = 2 * pi * m / 60
      if m % 5 = 0 then
         pip = 5
      else
         pip = 1
      end if
      circle 150-sin(a)*95,150-cos(a)*95,pip
   next m
   # draw the hands
   h = hour % 12 * 60 / 12 + minute/12 + second / 3600
   call drawhand(150,150,h,50,6,green)
   m = minute + second / 60
```

```
   call drawhand(150,150,m,75,4,red)
   call drawhand(150,150,second,100,3,blue)
   refresh
   pause 1
end while

subroutine drawhand(x, y, f, l, w, handcolor)
   # pass the location x and y
   # f as location on face of clock 0-59
   # length, width, and color of the hand
   color handcolor
   stamp x, y, 1, f/60*2*pi - pi / 2, {0,-w,l,0,0,w}
end subroutine
```

Program 105: Big Program – Clock with Hands



Sample Output 105: Big Program – Clock with Hands

Exercises:

Word Search

```
e c e i l i n g n d a b
f t z n n u r a r b g s
c y i t a e t e s m o k
f s r s g a m p h c t j
a a r e o a l t a n i s
t o t o i p i l e p d n
t n l n o r p c c o e a
i a d u a l a o o w g i
r e o g d j f s s e r d
r o o l d o i x k r e a
r l p a f n m w c s e r
d s h y p o t e n u s e
```

abs, acos, adjacent, asin, atan, ceiling, cos, degrees, float, floor, hypotenuse, int, integer, logarithm, modulo, opposite, power, radians, remainder, sin, tan

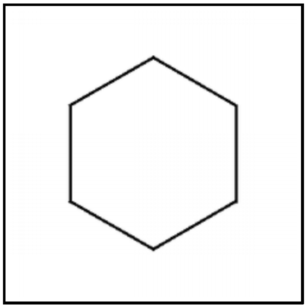
Problems

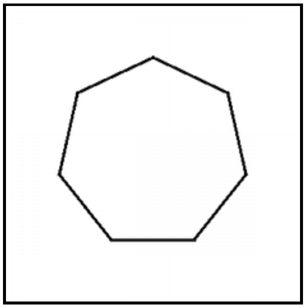
1. Have the user input a decimal number. Display the number it as a whole number and the closest faction over 1000 that is possible.

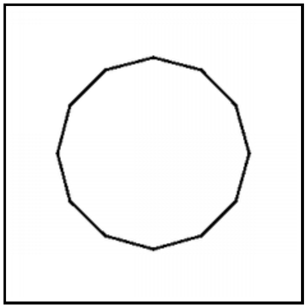
2. Take the program from Problem 1 and use a loop to reduce the fraction by dividing the numerator and denominator by common factors.

3. Write a program to draw a regular polygon with any number of sides (3 and up). Place it's center in the center of the graphics window and make its vertices 100 pixels from the center. Hint: A circle can be drawn by plotting points a specific radius from a point. The following plots a circle with a radius of 100 pixels around the point 150,150.

```
for a = 0 to 2*pi step .01
    plot 150-100*sin(a),150-100*cos(a)
next a
```


6 sided


7 sided


12 sided

Chapter 17: Working with Strings.

We have used strings to store non-numeric information, build output, and capture input. We have also seen, in Chapter 11, using the Unicode values of single characters to build strings.

This chapter shows several new functions that will allow you to manipulate string values.

The String Functions:

BASIC-256 includes eight common functions for the manipulation of strings. Table 8 includes a summary of them.

Function	Description
`string(`*`expression`*`)`	Convert expression (string, integer, or decimal value) to a string value.
`length(`*`string`*`)`	Returns the length of a string.
`left(`*`string, length`*`)`	Returns a string of length characters starting from the left.
`right(`*`string, length`*`)`	Returns a string of length characters starting from the right.
`mid(`*`string, start, length`*`)`	Returns a string of length characters starting from the middle of a string.
`upper(`*`expression`*`)`	Returns an upper case string.
`lower(`*`expression`*`)`	Returns a lower case string.
`instr(`*`haystack, needle`*`)`	Searches the string "haystack" for the "needle" and returns it's location.

Table 8: Summary of String Functions

String() Function:

The **string**() function will take an expression of any format and will return a string. This function is a convenient way to convert an integer or floating-point number into characters so that it may be manipulated as a string.

```
# string.kbs
# convert a number to a string

a = string(10 + 13)
print a
b = string(2 * pi)
print b
```

Program 106: The String Function

```
23
6.283185
```

Sample Output 106: The String Function

string(*expression*)

Convert expression (string, integer, or decimal value) to a string value.

Length() Function:

The length()function will take a string expression and return it's length in characters (or letters).

```
# length.kbs
# find length of a string

# should print 6, 0, and 17
print length("Hello.")
print length("")
print length("Programming Rulz!")
```

Program 107: The Length Function

```
6
0
17
```

Sample Output 107: The Length Function

length(*expression*)

Returns the length of the string expression. Will return zero (0) for the empty string "".

Left(), Right() and Mid() Functions:

The **left**(), **right**(), and **mid**() functions will extract sub-strings (or parts of a string) from a larger string.

```
# leftrightmid.kbs
# show right, left, and mid string functions

a = "abcdefghijklm"

print left(a,4)     # prints first 4 letters

print right(a,2)    # prints last 2 letters

print mid(a,4,3)    # prints 4th-7th letters
print mid(a,10,9)   # prints 10th and 11th letters
```

Program 108: The Left, Right, and Mid Functions

```
abcd
kl
def
jklm
```

Sample Output 108: The Left, Right, and Mid Functions

left(*string, length*)

Return a sub-string from the left end of a string. If length is equal or greater then the actual length of the string the entire string will be returned.

New Concept

right(*string, length*)

Return a sub-string from the right end of a string. If length is equal or greater then the actual length of the string the entire string will be returned.

New Concept

mid(*string, start, length*)

Return a sub-string of specified length from somewhere on the middle of a string. The start parameter specifies where the sub-string begins (1 = beginning of string).

Upper() and Lower() Functions:

The **upper**() and **lower**() functions simply will return a string of upper case or lower case letters. These functions are especially helpful when you are trying to perform a comparison of two strings and you do not care what case they actually are.

```
# upperlower.kbs

a = "Hello."

print lower(a)    # prints all lowercase

```

```
print upper(a)    # prints all UPPERCASE
```

Program 109: The Upper and Lower Functions

```
hello.
HELLO.
```

Sample Output 109: The Upper and Lower Functions


```
lower(string)
upper(string)
```

Returns an all upper case or lower case copy of the string expression. Non-alphabetic characters will not be modified.

Instr() Function:

The **instr**() function searches a string for the first occurrence of another string. The return value is the location in the big string of the smaller string. If the substring is not found then the function will return a zero (0).

```
# instr.kbs
# is one string inside another

a = "abcdefghijklm"
print 'the location of "hi" is ';
print instr(a,"hi")
print 'the location of "bye" is ';
print instr(a,"bye")
```

Program 110: The Instr Function

```
the location of "hi" is 8
the location of "bye" is 0
```

Sample Output 110: The Instr Function

New Concept

instr(haystack, needle)

Find the sub-string (needle)in another string expression (haystack). Return the character position of the start. If sub-string is not found return a zero (0).

Big Program

The decimal (base 10) numbering system that is most commonly used uses 10 different digits (0-9) to represent numbers.

Imagine if you will what would have happened if there were only 5 digits (0-4) – the number 23 ($2*10^1+3*10^0$) would become 43 ($4*5^1+3*5^0$) to represent the same number of items. This type of transformation is called radix (or base) conversion.

The computer internally does not understand base 10 numbers but converts everything to base 2 (binary) numbers to be stored in memory.

The "Big Program" this chapter will convert a positive integer from any base 2 to 36 (where letters are used for the 11th - 26th digits) to any other base.

```
# radix.kbs
# convert a number from one base (2-36) to another

digits = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"

frombase = getbase("from base")
inputstring "number in base " + frombase + " >", number
number = upper(number)

# convert number to base 10 and store in n
n = 0
for i = 1 to length(number)
   n = n * frombase
   n = n + instr(digits, mid(number, i, 1)) - 1
next i

tobase = getbase("to base")

# now build string in tobase
result = ""
while n <> 0
   result = mid(digits, n % tobase + 1, 1) + result
   n = n \ tobase
end while

print "in base " + tobase + " that number is " + result
end

function getbase(message)
   # get a base from 2 to 36
   do
      inputinteger message+"> ", base
   until base >= 2 and base <= 36
   return base
end function
```

Program 111: Big Program - Radix Conversion

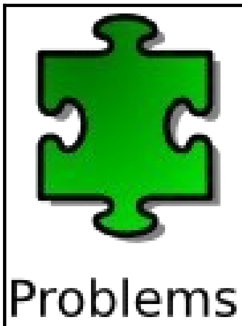
```
from base> 10
number in base 10 >999
to base> 16
in base 16 that number is 3E7
```

Sample Output 111: Big Program - Radix Conversion

Exercises:

u	r	h	t	g	n	e	l
p	g	i	r	a	g	k	f
p	r	n	l	c	f	l	r
e	q	i	i	e	f	e	t
r	d	r	g	r	f	x	s
v	i	i	r	h	t	t	n
p	m	m	x	o	t	s	i
r	e	w	o	l	f	w	i

instr, left, length, lower, mid, right, string, upper



1. Have the user enter a string and display the string backwards.

2. Modify problem 1 to create a palindrome testing program. Remove all characters from the string that are not letters before reversing it. Compare the results and print a message that the text entered is the same backwards as forwards.

```
enter a string >never odd or even
neveroddoreven
neveroddoreven
```

```
is a palindrome
```

3. You work for a small retail store that hides the original cost of an item on the price tag using an alphabetic code. The code is "roygbivace" where the letter 'r' is used for a 0, 'o' for a 1, … and 'e' is used for a 9. Write a program that will convert a numeric cost to the code and a code to a cost.

```
cost or code >9.84
ecb
```

```
cost or code >big
4.53
```

4: You and your friend want to communicate in a way that your friends can't easily read. The Cesar cipher (http://en.wikipedia.org/wiki/Caesar_cipher) is an easy but not very secure way to encode a message. If you and your friend agree to shift the same number of letters then you can easily share a secret message. Decoding a message is accomplished by applying a shift of 26 minus the original shift.

A sample of some of the shifts for the letters A-D are shown below. Notice that the letters wrap around.

Shift	A	B	C	D
1	B	C	D	E
13	M	N	O	P
25	Z	A	B	C

Write a program that asks for the shift and for a string and displays the text with the cipher applied.

```
shift >4
message >i could really go for
```

	`some pizza`
	`M GSYPH VIEPPC KS JSV WSQI TMDDE`
	
	`shift >22`
	`message >M GSYPH VIEPPC KS JSV`
	`WSQI TMDDE`
	`I COULD REALLY GO FOR SOME PIZZA`

Chapter 18: Files – Storing Information For Later.

We have explored the computer's short term memory with variables and arrays but how do we store those values for later? There are many different techniques for long term data storage.

BASIC-256 supports writing and reading information from files on your hard disk. That process of input and output is often written as I/O.

This chapter will show you how to read values from a file and then write them for long term storage.

Reading Lines From a File:

Our first program using files is going to show you many of the statements and constants you will need to use to manipulate file data. There are several new statements and functions in this program.

```
# readlfile.kbs
# read a simple text file

inputstring "file name> ", fn
if not exists(fn) then
   print fn + " does not exist."
   end
end if
#
n = 0
open fn
while not eof
   line = readline
   n = n + 1
```

```
   print n + " " + line
end while
#
print "the file " + fn + " is " + size + " bytes long."
close
```

Program 112: Read Lines From a File

```
file name> test.txt
1 These are the times that
2 try men's souls.
3 - Thomas Paine
the file test.txt is 57 bytes long.
```

Sample Output 112: Read Lines From a File

exist(*expression*)

Look on the computer for a file name specified by the string expression. Drive and path may be specified as part of the file name, but if they are omitted then the current working directory will be the search location.

Returns true if the file exists; else returns false.

```
open expression
open (expression)
open filenumber, expression
open (filenumber, expression)
```

Open the file specified by the expression for reading and writing to the specified file number. If the file does not exist it will be created so that information may be added (see write and writeline). Be sure to execute the close statement when the program is finished with the file.

BASIC-256 may have a total of eight (8) files open 0 to 7. If no file number is specified then the file will be opened as file number zero (0).

```
eof
eof()
eof(filenumber)
```

The **eof** function returns a value of true if we are at the end of the file for reading or false if there is still more data to be read.

If filenumber is not specified then file number zero (0) will be used.

New Concept

```
readline
readline()
readline(filenumber)
```

Return a string containing the contents of an open file up to the end of the current line. If we are at the end of the file [eof(filenumber) = true] then this function will return the empty string ("").

If filenumber is not specified then file number zero (0) will be used.

New Concept

```
size
size()
size(filenumber)
```

This function returns the length of an open file in bytes.

If filenumber is not specified then file number zero (0) will be used.

New Concept

```
close
close()
close filenumber
close(filenumber)
```

The **close** statement will complete any pending I/O to the file and allow for another file to be opened with the same number.

If filenumber is not specified then file number zero (0) will be used.

Writing Lines to a File:

In Program 112 we saw how to read lines from a file. The next two programs show different variations of how to write information to a file. In Program 113 we open and clear any data that may have been in the file to add our new lines and in Program 114 we append our new lines to the end (saving the previous data).

```
# resetwrite.kbs
# write text to a file, go back to begining
# and display the text

open "resetwrite.dat"

print "enter a blank line to close file"

# clear file (reset) and start over
reset
while true
   input ">", l
   if l = "" then exit while
   writeline l
end while

# go the the start and display contents
seek 0
k = 0
while not eof()
   k = k + 1
   print k + " " + readline()
end while

close
end
```

Program 113: Clear File and Write Lines

```
enter a blank line to close file
>this is some
>data, I am typing
>into the program.
>
1 this is some
2 data, I am typing
3 into the program.
```

Sample Output 113: Clear File and Write Lines


```
reset or
reset() or
reset filenumber
reset(filenumber)
```

Clear any data in an open file and move the file pointer to the beginning.

If filenumber is not specified then file number zero (0) will be used.

New Concept	**seek** ***expression*** **seek(*expression*)** **seek** ***filenumber,expression*** **seek (*filenumber,expression*)** Move the file pointer for the next read or write operation to a specific location in the file. To move the current pointer to the beginning of the file use the value zero (0). To seek to the end of a file use the **size**() function as the argument to the see statement. If filenumber is not specified then file number zero (0) will be used.

New Concept	**writeline** ***expression*** **writeline(*expression*)** **writeline** ***filenumber,expression*** **writeline (*filenumber,expression*)** Output the contents of the expression to an open file and then append an end of line mark to the data. The file pointer will be positioned at the end of the write so that the next write statement will directly follow. If filenumber is not specified then file number zero (0) will be used.

```
# appendwrite.kbs
# append new lines on the end of a file
# then display it

open "appendwrite.dat"

print "enter a blank line to close file"
```

```

# move file pointer to end of file and append
seek size
while true
   input ">", l
   if l = "" then exit while
   writeline l
end while

# move file pointer to beginning and show contents
seek 0
k = 0
while not eof()
   k = k + 1
   print k + " " + readline()
end while

close
end
```

Program 114: Append Lines to a File

```
enter a blank line to close file
>sed sed sed
>vim vim vim
>
1 bar bar bar
2 foo foo foo
3 grap grap grap
4 sed sed sed
5 vim vim vim
```

Sample Output 114: Append Lines to a File

Read() Function and Write Statement:

In the first three programs of this chapter we have discussed the **readline**() function and **writeline** statement. There are two other statements that will read and write a file. They are the **read**() function and **write** statement.


```
read
read()
read(filenumber)
```

Read the next word or number (token) from a file. Tokens are delimited by spaces, tab characters, or end of lines. Multiple delimiters between tokens will be treated as one.

If filenumber is not specified then file number zero (0) will be used.


```
write expression
write (expression)
write filenumber,expression
write (filenumber,expression)
```

Write the string expression to a file file. Do not add an end of line or a delimiter.

If filenumber is not specified then file number zero (0) will be used.

Big Program	This program uses a single text file to help us maintain a list of our friend's telephone numbers.

```
# phonelist.kbs
# add a phone number to the list and show

filename = "phonelist.txt"

print "phonelist.kbs - Manage your phone list."
do
   input "Add, List, Quit (a/l/q)? ",action
    if left(lower(action),1) = "a" then call addrecord(filename)
   if left(lower(action),1) = "l" then call listfile(filename)
until left(lower(action),1) = "q"
end

subroutine listfile(f)
   if exists(f) then
      # list the names and phone numbers in the file
      open f
      print "the file is " + size + " bytes long"
      while not eof
         # read next line from file and print it
         print readline
      end while
      close
   else
      print "No phones on file. Add first."
   end if
```

```
end subroutine

subroutine addrecord(f)
   input "Name to add? ", name
   input "Phone to add? ", phone
   open f
   # seek to the end of the file
   seek size()
   # we are at end of file - add new line
   writeline name + ", " + phone
   close
end subroutine
```

Program 115: Big Program - Phone List

```
phonelist.kbs - Manage your phone list.
Add, List, Quit (a/l/q)? l
the file is 46 bytes long
jim, 555-5555
sam, 555-7777
doug, 555-3333
Add, List, Quit (a/l/q)? a
Name to add? ang
Phone to add? 555-0987
Add, List, Quit (a/l/q)? l
the file is 61 bytes long
jim, 555-5555
sam, 555-7777
doug, 555-3333
ang, 555-0987
Add, List, Quit (a/l/q)? q
```

Sample Output 115: Big Program - Phone List

Exercises:

Word Search

```
e n i l e t i r w e
s y r o t c e r i d
n e k o t s q h e r
e f m e t s f l e p
p p s s i i i a s c
o e i z l m d e l e
r x e e i l e o r o
e e r t i k s y e f
t k e n z e l j a d
b r e w r i t e d n
```

close, delimiter, directory, eof, exists, file, open, read, readline, reset, seek, size, token, write, writeline
words

Problems

1. Create a file in the directory where you save your programs named "numbers.txt". Open it with a text editor, like Notepad in Windows or gEdit in LINUX, and type in a list of decimal numbers. Put each one on a separate line.

Now write a program to read the numbers from the file, one line at a time. Calculate the total of the numbers in the file and the average.

Remember to use the **float()** function to convert the string you read from the file to a numeric value before you add it to the running total.

2. Create a file in the directory where you save your programs named "people.txt". Open it with a text editor, like Notepad in

Windows or gEdit in LINUX, and type in the data below.

```
Jim,M,47
Mary,F,23
Bob,M,67
John,M,13
Sue,F,18
```

Write a program that will read in the data from the people file. Use string handling functions from Chapter 16 to break each line into three parts: 1) name, 2) gender, and 3) age. Tally the total of the ages, the number of people, and the number of males as you read the file. Once you have read all the records display the percentage of males and the average age of the people in the file.

3. Create a file in the directory where you save your programs named "assignments.txt". Open it with a text editor, like Notepad in Windows or gEdit in LINUX, and type in the data below.

```
Jim,88,45
Joe,90,33
Mary,54,29
Maury, 57,30
```

Write a program that will read in the data from the assignments file and write out a new file named "finalgrade.txt" with the student's name, a comma, and their course grade. Calculate the course grade for each student based on the two assignment grades. The first assignment was worth 100 points and the second assignment was worth 50 points.

The output should look something like:

```
Jim,88
…
```

Chapter 19: Stacks, Queues, Lists, and Sorting

This chapter introduces a few advanced topics that are commonly covered in the first Computer Science class at the University level. The first three topics (Stack, Queue, and Linked List) are very common ways that information is stored in a computer system. The last two are algorithms for sorting information.

Stack:

A stack is one of the common data structures used by programmers to do many tasks. A stack works like the "discard pile" when you play the card game "crazy-eights". When you add a piece of data to a stack it is done on the top (called a "push") and these items stack upon each other. When you want a piece of information you take the top one off the stack and reveal the next one down (called a "pop"). Illustration 31 shows a graphical example.

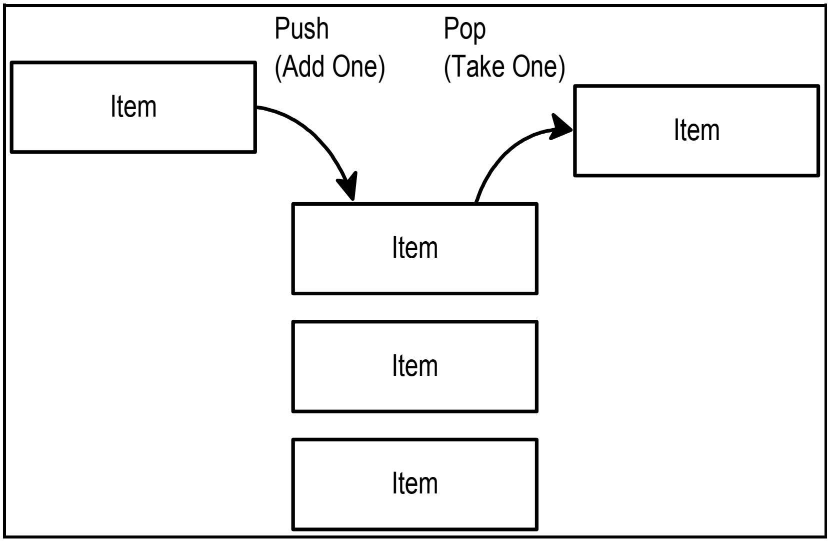


Illustration 31: What is a Stack

The operation of a stack can also be described as "last-in, first-out" or LIFO for short. The most recent item added will be the next item removed. Program 116 implements a stack using an array and a pointer to the most recently added item. In the "`push`" subroutine you will see array logic that will re-dimension the array to make sure there is enough room available in the stack for virtually any number of items to be added.

```
# stack.kbs
# implementing a stack using an array

dim stack(1) # array to hold stack with initial size
nstack = 0 # number of elements on stack
global stack, nstack

call push(1)
call push(2)
call push(3)
call push(4)
call push(5)

```

```
while not empty()
print pop()
end while

end

function empty()
# return true if the start is empty
return nstack=0
end function

function pop()
# get the top number from stack and return it
# or print a message and return -1
if nstack = 0 then
print "stack empty"
return -1
end if
nstack = nstack - 1
value = stack[nstack]
return value
end function

subroutine push(value)
# push the number in the variable value onto the stack
# make the stack larger if it is full
if nstack = stack[?] then redim stack(stack[?] + 5)
stack[nstack] = value
nstack = nstack + 1
end subroutine
```

Program 116: Stack

```
5
4
3
2
1
```

Sample Output 116: Stack


```
global variable
global variable, variable...
```

Global tells BASIC-256 that these variables can be seen by the entire program (both inside and outside the functions/subroutines). Using global variables is typically not encouraged, but when there is the need to share several values or arrays it may be appropriate.

Queue:

The queue (pronounced like the letter Q) is another very common data structure. The queue, in its simplest form, is like the lunch line at school. The first one in the line is the first one to get to eat. Illustration 32 shows a block diagram of a queue.

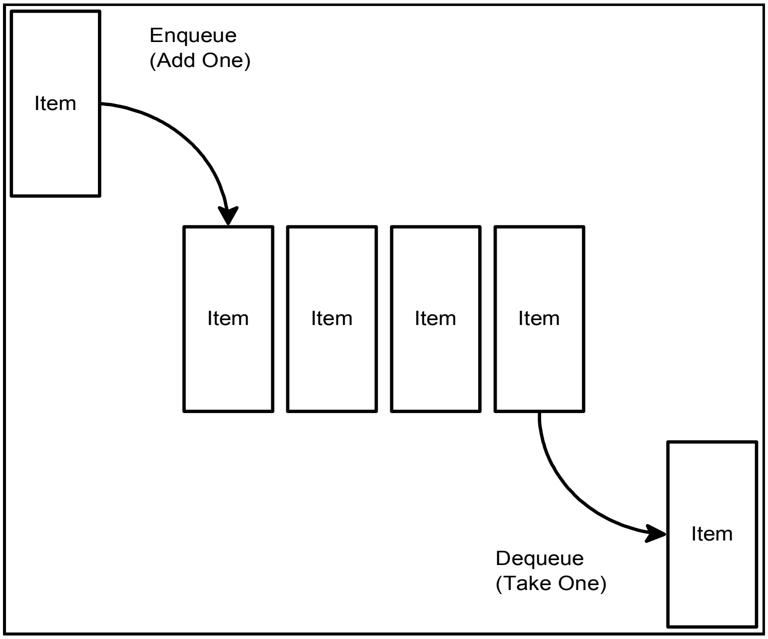


Illustration 32: What is a Queue

The terms enqueue (pronounced in-q) and dequeue (pronounced dee-q) are the names we use to describe adding a new item to the end of the line (tail) or removing an item from the front of the line (head). Sometimes this is described as a "first-in, first-out" or FIFO. The example in Program 117 uses an array and two pointers that keep track of the head of the line and the tail of the line.

```
# queue.kbs
# implementing a queue using an array

global queuesize, queue, queuetail, queuehead, inqueue

call createqueue(5)

call enqueue(1)
call enqueue(2)
```

```

print dequeue()
print

call enqueue(3)
call enqueue(4)

print dequeue()
print dequeue()
print

call enqueue(5)
call enqueue(6)
call enqueue(7)

# empty everybody from the queue
while inqueue > 0
   print dequeue()
end while

end

subroutine createqueue(z)
   # maximum number of entries in the queue at any one time
   queuesize = z
   # array to hold queue with initial size
   dim queue(z)
   # location in queue of next new entry
   queuetail = 0
   # location in queue of next entry to be returned (served)
   queuehead = 0
   # number of entries in queue
   inqueue = 0
end subroutine

function dequeue()
   if inqueue = 0 then
```

```
      print "queue is empty"
      value = -1
   else
      value = queue[queuehead]
      inqueue--
      queuehead++
      if queuehead = queuesize then queuehead = 0
   end if
   return value
end function

subroutine enqueue(value)
   if inqueue = queuesize then
      print "queue is full"
   else
      queue[queuetail] = value
      inqueue++
      queuetail++
      if queuetail = queuesize then queuetail = 0
   end if
end subroutine
```

Program 117: Queue

```
1

2
3

4
5
6
7
```

Sample Output 117: Queue

Linked List:

In most books the discussion of this material starts with the linked list. Because BASIC-256 handles memory differently than many other languages this discussion was saved after introducing stacks and queues.

A linked list is a sequence of nodes that contains data and a pointer or index to the next node in the list. In addition to the nodes with their information we also need a pointer to the first node. We call the first node the "Head". Take a look at Illustration 33 and you will see how each node points to another.

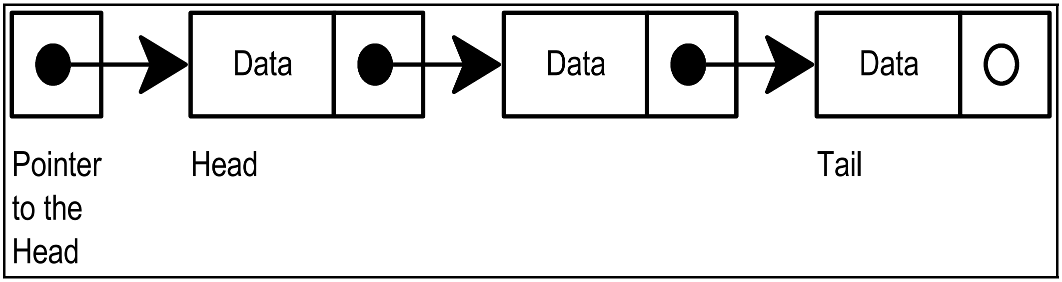


Illustration 33: Linked List

An advantage to the linked list, over an array, is the ease of inserting or deleting a node. To delete a node all you need to do is change the pointer on the previous node (Illustration 34) and release the discarded node so that it may be reused.

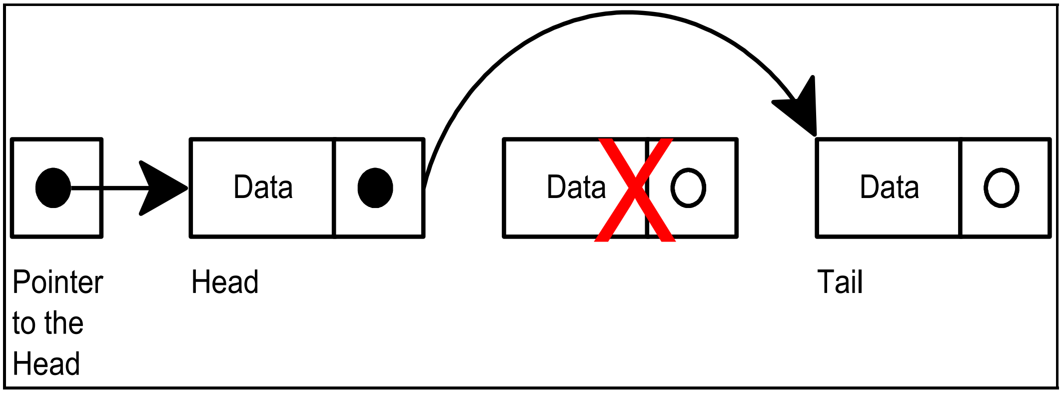


Illustration 34: Deleting an Item from a Linked List

Inserting a new node is also as simple as creating the new node, linking the new node to the next node, and linking the previous node to the first node. Illustration 35 Shows inserting a new node into the second position.

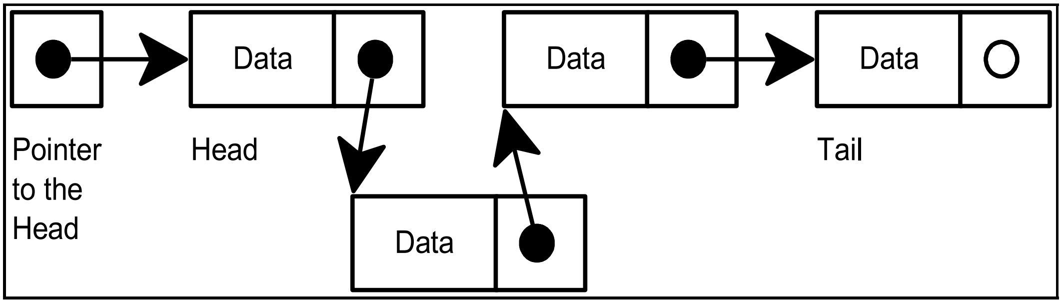


Illustration 35: Inserting an Item into a Linked List

Linked lists are commonly thought of as the simplest data structures. In the BASIC language we can't allocate memory like in most languages so we will simulate this behavior using arrays. In Program 118 we use the data array to store the text in the list, the nextitem array to contain the index to the next node, and the freeitem array to contain a stack of free (unused) array indexes.

```
# linkedlist.kbs

# create a linked list using arrays

# data is an array coitaining the data strings in the list
# nextitem is an array with pointers to the next data item
# if nextitem is -2 it is free or -1 it is the end

global head, data, nextitem
call initialize(6)

# list of 3 people
call append("Bob")
call append("Sue")
call append("Guido")
call displaylist()
call displayarrays()
call wait()

print "delete person 2"
call delete(2)
call displaylist()
call displayarrays()
call wait()

print "insert Mary into the front of the list (#1)"
call insert("Mary",1)
call displaylist()
call displayarrays()
call wait()

print "insert John at position 2"
call insert("John",2)
call displaylist()
call displayarrays()
call wait()

```

```
print "delete person 1"
call delete(1)
call displaylist()
call displayarrays()
call wait()

end

subroutine wait()
   input "press enter to continue> ",foo
   print
end subroutine

subroutine initialize(n)
   head = -1    # start of list (-1 pointer to nowhere)
   dim data(n)
   dim nextitem(n)
   # initialize items as free
   for t = 0 to data[?]-1
      call freeitem(t)
   next t
end subroutine

subroutine freeitem(i)
   # free element at array index i
   data[i] = ""
   nextitem[i] = -2
end subroutine

function findfree()
   # find a free item (an item pointing to -2)
   for t = 0 to data[?]-1
      if nextitem[t] = -2 then return t
   next t
   print 'no free elements to allocate'
   end
end function

```

```
function createitem(text)
   # create a new item on the list
   # and return index to new location
   i = findfree()
   data[i] = text
   nextitem[i] = -1
   return i
end function

subroutine displaylist()
   # showlist by following the linked list
   print "list..."
   k = 0
   i = head
   do
      k = k + 1
      print k + " ";
      print data[i]
      i = nextitem[i]
   until i = -1
end subroutine

subroutine displayarrays()
   # show data actually stored and how
   print "arrays..."
   for i = 0 to data[?]-1
      print i + " " + data[i] + " >" + nextitem[i] ;
      if head = i then print " <<head";
      print
   next i
end subroutine

subroutine insert(text, n)
   # insert text at position n
   index = createitem(text)
   if n = 1 then
      nextitem[index] = head
      head = index
   else
```

```
      k = 2
      i = head
      while i <> -1 and k <> n
         k = k + 1
         i = nextitem[i]
      end while
      if i <> -1 then
         nextitem[index] = nextitem[i]
         nextitem[i] = index
      else
         print "can't insert beyond end of list"
      end if
   end if
end subroutine

subroutine delete(n)
   # delete element n from linked list
   if n = 1 then
      # delete head - make second element the new head
      index = head
      head = nextitem[index]
      call freeitem(index)
   else
      k = 2
      i = head
      while i <> -1 and k <> n
         k = k + 1
         i = nextitem[i]
      end while
      if i <> -1 then
         index = nextitem[i]
         nextitem[i] = nextitem[nextitem[i]]
         call freeitem(index)
      else
         print "can't delete beyond end of list"
      end if
   end if
end subroutine
```

```
subroutine append(text)
   # append  text to end of linked list
   index = createitem(text)
   if head = -1 then
      # no head yet - make item the head
      head = index
   else
      # move to the end of the list and add new item
      i = head
      while nextitem[i] <> -1
         i = nextitem[i]
      end while
      nextitem[i] = index
   endif
end subroutine
```

Program 118: Linked List

Explore	Re-write Program 118 to implement a stack and a queue using a linked list.

Slow and Inefficient Sort - Bubble Sort:

The "Bubble Sort" is probably the worst algorithm ever devised to sort a list of values. It is very slow and inefficient except for small sets of items. This is a classic example of a bad algorithm.

The only real positive thing that can be said about this algorithm is that it is simple to explain and to implement. Illustration 36 shows a flow-chart of the algorithm. The bubble sort goes through the array over and over again swapping the order of adjacent items until the sort is complete,

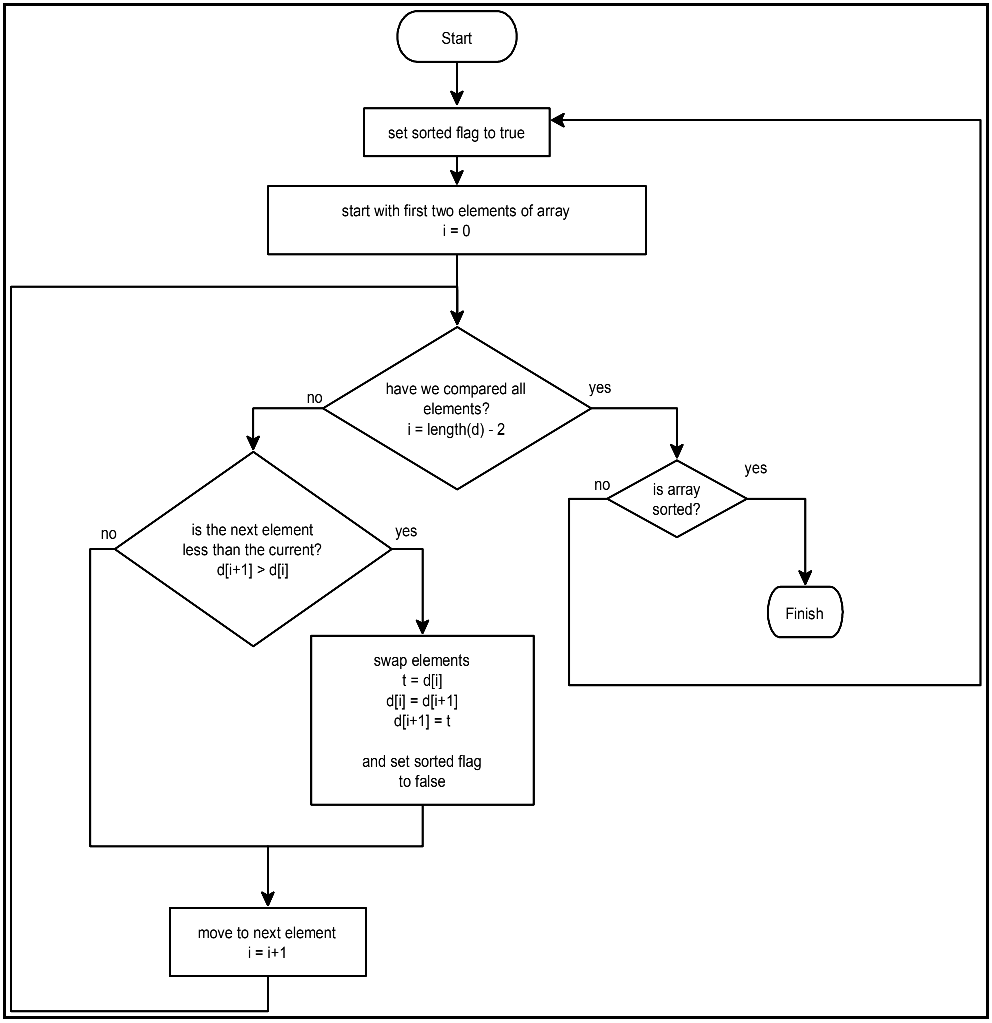


Illustration 36: Bubble Sort - Flowchart

```
# bubblesortf.kbs
# implementing a simple sort

# a bubble sort is one of the SLOWEST algorithms
```

```
# for sorting but it is the easiest to implement
# and understand.
#
# The algorithm for a bubble sort is
# 1. Go through the array swaping adjacent values
#    so that lower value comes first.
# 2. Do step 1 over and over until there have
#    been no swaps (the array is sorted)
#

dim d(20)

# fill array with unsorted numbers
for i = 0 to d[?]-1
   d[i] = int(rand * 1000)
next i

print "*** Un-Sorted ***"

call displayarray(ref(d))
call bubblesort(ref(d))

print "*** Sorted ***"
call displayarray(ref(d))
end

subroutine displayarray(ref(array))
   # print out the array's values
   for i = 0 to array[?]-1
      print array[i] + " ";
   next i
   print
end subroutine

subroutine bubblesort(ref(array))
   do
      sorted = true
      for i = 0 to array[?] - 2
         if array[i] > array[i+1] then
```

```
                sorted = false
                temp = array[i+1]
                array[i+1] = array[i]
                array[i] = temp
            end if
        next i
    until sorted
end subroutine
```

Program 119: Bubble Sort

```
*** Un-Sorted ***
878 95 746 345 750 232 355 472 649 678 758 424
653 698 482 154 91 69 895 414
*** Sorted ***
69 91 95 154 232 345 355 414 424 472 482 649
653 678 698 746 750 758 878 895
```

Sample Output 119: Bubble Sort

Better Sort – Insertion Sort:

The insertion sort is another algorithm for sorting a list of items. It is usually faster than the bubble sort, but in the worst case case could take as long.

The insertion sort gets it's name from how it works. The sort goes through the elements of the array (index = 1 to length -1) and inserts the value in the correct location in the previous array elements. Illustration 37 shows a step-by-step example.

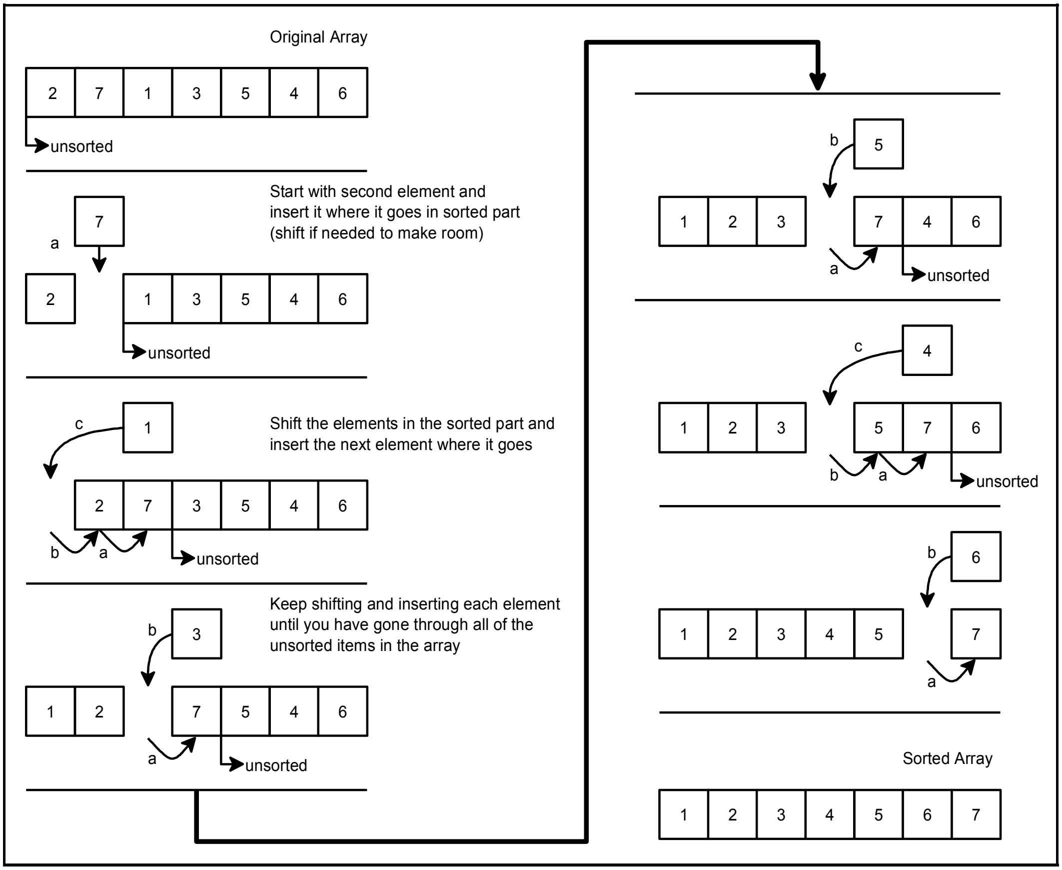


Illustration 37: Insertion Sort - Step-by-step

```
# insertionsort.kbs
# implementing an efficient sort

# The insertion sort loops through the items
# starting at the second element.

# takes current element and inserts it
# in the the correct sorted place in
# the previously sorted elements

# moving from backward from the current
# location and sliding elements with a
```

```
# larger value forward to make room for
# the current value in the correct
# place (in the partially sorted array)

dim d(20)

# fill array with unsorted numbers
for i = 0 to d[?]-1
   d[i] = int(rand * 1000)
next i

print "*** Un-Sorted ***"
call displayarray(ref(d))

call insertionsort(ref(d))

print "*** Sorted ***"
call displayarray(ref(d))
end

subroutine displayarray(ref(a))
   # print out the array's values
   for i = 0 to a[?]-1
      print a[i] + " ";
   next i
   print
end subroutine

subroutine insertionsort(ref(a))
   for i = 1 to a[?] - 1
      currentvalue = a[i]
      j = i - 1
      done = false
      do
         if a[j] > currentvalue then
            a[j+1] = a[j]
            j = j - 1
            if j < 0 then done = true
         else
```

```
                done = true
            endif
        until done
        a[j+1] = currentvalue
    next i
end subroutine
```

Program 120: Insertion Sort

```
*** Un-Sorted ***
913 401 178 844 574 289 583 806 332 835 439 52
140 802 365 972 898 737 297 65
*** Sorted ***
52 65 140 178 289 297 332 365 401 439 574 583
737 802 806 835 844 898 913 972
```

Sample Output 120: Insertion Sort

Exercises:

Word Search

```
k f i f o e q i q h m t o
n o f i l u x q q y e r b
i h p v e o d t q y u o d
l m p u f d s r c t e s e
v o e k x v m o i s u n u
p g f c i l e s a i q o e
q l f a u h m e l l n i u
v o i t q s o l l i e t q
i b c s z u r b o d t r e
z a i v e p y b c s z e d
d l e y d j h u a r o s p
z y n g o v c b t y l n q
m x t s n y i t e i q i b
```

allocate, bubblesort, dequeue, efficient, enqueue, fifo, global, insertionsort, lifo, link, list, memory, node, pop, push, queue, stack

Problems

1. Rewrite the “Bubble Sort” function to sort strings, not numbers. Add a second true/false argument to make the sort case-sensitive/insensitive.

2. Implement the “Insertion Sort” using the linked-list functions so that items are moved logically and not physically moved.

3. Develop a function to do the “Merge Sort” (http://en.wikipedia.org/wiki/Merge_sort) on an array of numbers. Create arrays of random numbers of varying lengths ans sotrt them using the “Bubble Sort”, the “Insertion Sort”, and your new “Merge Sort”. Which is the slowest? Fastest?

Chapter 20 – Runtime Error Trapping

As you have worked through the examples and created your own programs you have seen errors that happen while the program is running. These errors are called "runtime errors". BASIC-256 includes a group of special commands that allow your program to recover from or handle these errors.

You may already have seen programs that throw or display errors when they are running. They often occur when an invalid mathematical operation happens or when an unassigned variable is used. In Program 121 you see a program that works most of the time but will error and quit running if the denominator is zero.

```
# divider.kbs
# simple division

print "divide two numbers"
while true
   input "numerator?", n
   input "denominator?", d
   q = n/d
   print "quotient is  " + q
end while
```

Program 121: Simple Division Program That May Error

```
divide two numbers
numerator?6
denominator?9
quotient is  0.6666667
numerator?5
denominator?2
quotient is  2.5
numerator?9
```

```
denominator?0
ERROR on line 8: Division by zero.
```

Sample Output 121: Simple Division Program That May Error

Try a Statement and Catch an Error:

The **try/catch/end try** block is structured so that if a trappable runtime error occurs in the code between the **try** and the **catch**, the code immediately following the **catch** will be executed. The following example shows the simple division program now catching the division by zero error.

```
# trycatch.kbs
# simple try catch

print "divide two numbers"
while true
   input "numerator?", n
   input "denominator?", d
   try
      q = n/d
      print "quotient is  " + q
   catch
      print "I can't divide " + d + " into " + n
   end try
end while
```

Program 122: Simple Division Program That Catches Error

```
divide two numbers
numerator?5
denominator?6
quotient is  0.8333333
numerator?99
denominator?0
I can't divide 0 into 99
numerator?4
```

```
denominator?3
quotient is  1.3333333
numerator?
```

Sample Output 122: Simple Division Program That Catches Error


```
try
    statement(s) to try
catch
    statement(s) to execute if an error occurs
end try
```

The **try/catch/end try** ...

Trapping errors, when you do not mean too, can cause problems and mask other problems with your programs. Error trapping should only be used when needed and disabled when not.

Finding Out Which Error:

Sometimes just knowing that an error happened is not enough. There are functions that will return the error number (**lasterror**), the line where the error happened in the program (**lasterrorline**), a text message describing the error (**lasterrormessage**), and extra command specific error messages (**lasterrorextra**).

```
# trap.kbs
# error trapping with reporting

try
   print "z = " + z
```

```
catch
   print "Caught Error"
   print "   Error = " + lasterror
   print "   On Line = " + lasterrorline
   print "   Message = " + lasterrormessage
end try
print "Still running after error"
```

Program 123: Try/Catch - With Messages

```
Caught Error
   Error = 12
   On Line = 4
   Message = Unknown variable z
Still running after error
```

Sample Output 123: Try/Catch - With Messages

New Concept

```
lasterror or lasterror()
lasterrorline or lasterrorline()
lasterrormessage or lasterrormessage()
lasterrorextra or lasterrorextra()
```

The four "last error" functions will return information about the last trapped error. These values will remain unchanged until another error is encountered.

lasterror	Returns the number of the last trapped error. If no errors have been trapped this function will return a zero. See Appendix G: Errors and Warnings for a complete list of trappable errors.
lasterrorline	Returns the line number, of the program, where the last error was trapped.
l**asterrormessage**	Returns a string describing the last error.
lasterrorextra	Returns a string with additional error information. For most errors this function will not return any information.

Type Conversion Errors

BASIC-256 by default will return a zero when it is unable to convert a string to a number. You may have seen this previously when using the **inputinteger** and **inputfloat** statements. This will also happen when the **int()** and **float()** functions convert a string to a number.

You may optionally tell BASIC-256 to display a trappable warning or throw an error that stops execution of your program. You can change this setting in the “Preferences” dialog, on the User tab.

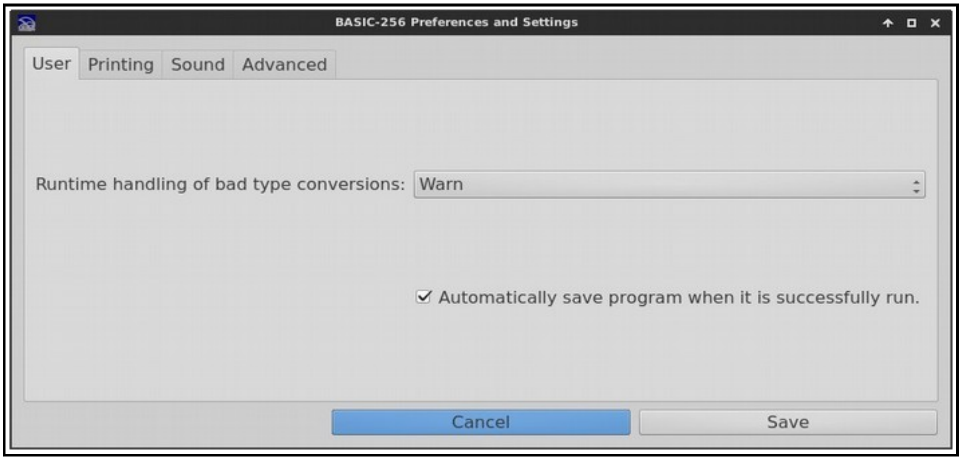


Illustration 38: Preferences - Type Conversion Ignore/Warn/Error

```
# inputnumber.kbs

input "enter a number> ",a
print a
```

Program 124: Type Conversion Error

Program run with the errors "Ignored".

```
enter a number> foo
0
```

Sample Output 124: Type Conversion Error - Ignored (Deafult)

Program run with the "Warning" enabled. Notice that the program continues running but displays a message. The **try**/**catch**/**end try** statements will catch the warning so that you may display a custom message or do special proccessing.

```
enter a number> sdfsdf
```

```
WARNING on line 3: Unable to convert string to
number, zero used.
0
```

Sample Output 124: Type Conversion Error - Warning

This third example had the property set to “Error”. When an invalid type conversion happens an error is displayed and program execution stops. This error is trappable with the **try**/**catch**/**end try** statements.

```
enter a number> abcd
ERROR on line 3: Unable to convert string to
number.
```

Sample Output 124: Type Conversion Error - Error

Creating An Error Trapping Routine:

There is a second way to trap run-time errors, by using an error trapping subroutine. When this type of error trapping is turned on, with the **onerror** statement, the program will call a specified subroutine when an error occurs. When the error trap returns the program will automatically continue with the next line in the program.

If we look at Program 125 we will see that the program calls the subroutine when it tries to read the value of z (an undefined variable). If we try to run the same program with line one commented out or removed the program will terminate when the error happens.

```
1	# simpletrap.kbs
2	# simple error trapping
3
4	onerror trap
5
6	print "z = " + z
```

```
print "Still running after error"
end

subroutine trap()
   print "I trapped an error."
end subroutine
```

Program 125: Simple Runtime Error Trap

```
I trapped an error.
z = 0
Still running after error
```

Sample Output 125: Simple Runtime Error Trap


```
onerror label
```

Create an error trap that will automatically jump to the subroutine at the specified label when an error occurs.

You may use the **lasterror**, **lasterrorline**, **lasterrormessage**, and **lasterrorextra** functions within your error trap subroutine to display any messages or do any processing you wish to do. Additionally you may not define an **onerror** trap inside a **try/catch**.

Turning Off Error Trapping Routine:

Sometimes in a program we will want to trap errors during part of the program and not trap other errors. The **offerror** statement turns error trapping off. This causes all errors encountered to stop the program.

```
# trapoff.kbs
# error trapping with reporting

onerror errortrap
print "z = " + z
print "Still running after first error"

offerror
print "z = " + z
print "Still running after second error"
end

subroutine errortrap()
   print "Error Trap - Activated"
end subroutine
```

Program 126: Turning Off the Trap

```
Error Trap - Activated
z = 0
Still running after first error
ERROR on line 6: Unknown variable
```

Sample Output 126: Turning Off the Trap

Exercises:

Word Search

```
e u q r l w f e p j x s p w n c
p g u b i r r h f j w w w o c p
b l a s t e r r o r e x t r a p
q e e s v w j l p g a m w l o q
t a n n s r q o i i t m r a n o
r f x i d e o u c a t c h t e y
y h z r l t m r f k o s k v r i
q o i b m r r r r s i e f b r f
x l f x o z o y o e l b b i o a
y k m f z o r r q r t s k e r a
z a h l e i r y r p r s f g y m
i l i l n r e j f e p e a n r l
a q c m t q r k o g t l t l u u
r e u k z b b o u f l s g s t j
m s u h l a r x r m v w a q a l
u b z r l h a l k p a r t l n l
```

catch, endtry, error, lasterror, lasterrorextra, lasterrorline, lasterrormessage, offerror, onerror, trap, try

Problems

1. Set the “runtime handling of bad type conversion” “Preference” to “warn” or “Error” and write a simple program that asks the user to enter a number. If the user enters something that is not a number, trap the warning/error and ask again.

```
enter a number> gdf2345
bad entry. try again.
enter a number> fdg545
bad entry. try again.
enter a number> 43fdgdf
bad entry. try again.
```

	`enter a number> 22`
	`You entered 22`
	
	2. Take the logic you just developed in Problem 1 and create a function that takes one argument, the prompt message, repeatedly asks the user for a number until they enter one, and returns the user's numeric entry.
	
	3. Write a program that causes many errors to occur, trap and them. Be sure to check out Appendix G: Errors and Warnings for a complete list

Chapter 21: Database Programming

This chapter will show how BASIC-256 can connect to a simple relational database and use it to store and retrieve useful information.

What is a Database:

A database is simply an organized collection of numbers, string, and other types of information. The most common type of database is the "Relational Database". Relational Databases are made up of four major parts: tables, rows, columns, and relationships (see Table 9).

Table	A table consists of a predefined number or columns any any number of rows with information about a specific object or subject. Also known as a relation.
Row	Also called a tuple.
Column	This can also be referred to as an attribute.
Relationship	A reference of the key of one table as a column of another table. This creates a connection between tables.

Table 9: Major Components of a Relational Database

The SQL Language:

Most relational databases, today, use a language called SQL to actually extract and manipulate data. SQL is actually an acronym for Structured Query Language. The original SQL language was developed by IBM in the 1970s and has become the primary language used by relational databases.

SQL is a very powerful language and has been implemented by dozens of software companies, over the years. Because of this complexity there are many different dialects of SQL in use. BASIC-256 uses the SQLite database engine. Please see the SQLite web-page at http://www.sqlite.org for more information about the dialect of SQL shown in these examples.

Creating and Adding Data to a Database:

The SQLite library does not require the installation of a database sever or the setting up of a complex system. The database and all of its parts are stored in a simple file on your computer. This file can even be copied to another computer and used, without problem.

The first program (Program 127: Create a Database) creates a new sample database file and tables. The tables are represented by the Entity Relationship Diagram (ERD) as shown in Illustration 39.

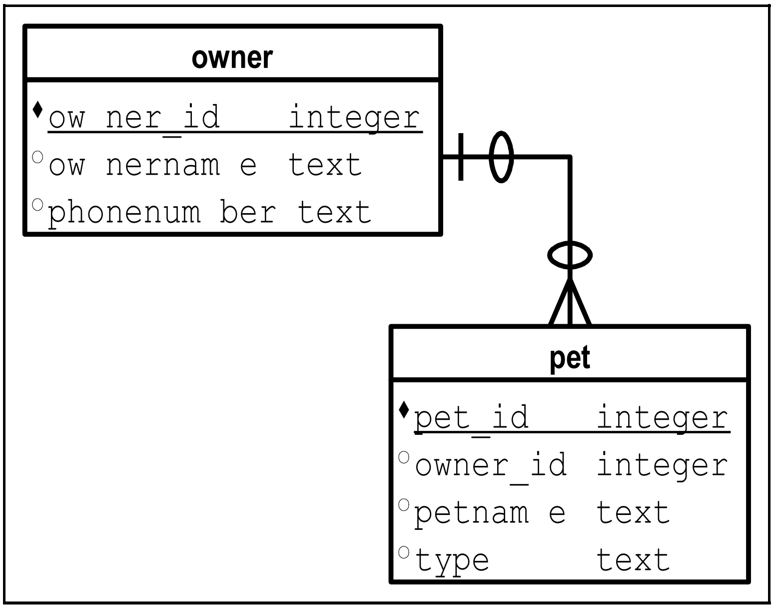


Illustration 39: Entity Relationship Diagram of Chapter Database

```
# dbcreate.kbs - create the pets database and tables

# delete old database and create a database with two tables
file = "pets.sqlite3"
if exists(file) then kill(file)
dbopen file

stmt =  "CREATE TABLE owner (owner_id INTEGER, ownername TEXT, phonenumber TEXT, PRIMARY KEY (owner_id));"
call executeSQL(stmt)

stmt =  "CREATE TABLE pet (pet_id INTEGER, owner_id INTEGER, petname TEXT, type TEXT, PRIMARY KEY (pet_id), FOREIGN KEY (owner_id) REFERENCES owner (owner_id));"
call executeSQL(stmt)

# wrap everything up
dbclose
print file + " created."
end

subroutine executeSQL(stmt)
     print stmt
     try
          dbexecute stmt
     catch
          print "Caught Error"
          print "   Error = " + lasterror
          print "   On Line = " + lasterrorline
          print "   Message = " + lasterrormessage
     endtry
end subroutine
```

Program 127: Create a Database

```
CREATE TABLE owner (owner_id INTEGER, ownername
TEXT, phonenumber TEXT, PRIMARY KEY
(owner_id));
CREATE TABLE pet (pet_id INTEGER, owner_id
INTEGER, petname TEXT, type TEXT, PRIMARY KEY
(pet_id), FOREIGN KEY (owner_id) REFERENCES
owner (owner_id));
pets.sqlite3 created.
```

Sample Output 127: Create a Database

So far you have seen three new database statements: **dbopen** – will open a database file and create it if it does not exist, **dbexecute** – will execute an SQL statement on the open database, and **dbclose** – closes the open database file.


```
dbopen filename
```

Open an SQLite database file. If the database does not exist then create a new empty database file.

`dbexecute sqlstatement`

Perform the SQL statement on the currently open SQLite database file. No value will be returned but a trappable runtime error will occur if there were any problems executing the statement on the database.

`dbclose`

Close the currently open SQLite database file. This statement insures that all data is written out to the database file.

These same three statements can also be used to execute other SQL statements. The INSERT INTO statement (Program 128) adds new rows of data to the tables and the UPDATE statement (Program 129) will change an existing row's information.

When you are building a SQL statement that may contain informtion typed in by the user, you must be very careful and handle quotation marks that they might type in. Malicious users may try to do something called an SQL-Injection where they will embed a harmful SQL statement into what they have entered into the program. Data may be lost or compromised if care is not taken.

The following examples use a function called "quote" that will quote a string containing quotation marks correctly and should eliminate this risk for simple programs.

The "quote" function will place single quotation marks around a string and return the string with the quotes. If a string contains single quotations within it, they will be doubled and handled correctly by SQLite.

```
# quote.kbs - quote a string for SQLite
# SAVE IT AS quote.kbs
#
# wrap a string in single quotes (for a sql statement)
# if it contains a single quote double it
function quote(a)
    return "'" + replace(a,"'","''") + "'"
end function
```

```
# dbinsert.kbs - add rows to the database

include "quote.kbs"

file = "pets.sqlite3"
dbopen file

call addowner(1, "Jim", "555-3434")
```

```
call addpet(1, 1, "Spot", "Cat")
call addpet(2, 1, "Fred", "Cat")
call addpet(3, 1, "Elvis", "Cat")

call addowner(2, "Sue", "555-8764")
call addpet(4, 2, "Alfred", "Dog")
call addpet(5, 2, "Fido", "Cat")

call addowner(3, "Amy", "555-4321")
call addpet(6, 3, "Bones", "Dog")

call addowner(4, "Dee", "555-9659")
call addpet(7, 4, "Sam", "Goat")

# wrap everything up
dbclose
end

subroutine addowner(owner_id, ownername, phonenumber)
    stmt = "INSERT INTO owner (owner_id, ownername, phonenumber) VALUES (" + owner_id + "," + quote(ownername) + "," + quote(phonenumber) + ");"
    print stmt
    try
        dbexecute stmt
    catch
        print "Unbale to add owner " + owner_id + " " + lasterrorextra
    end try
end subroutine

subroutine addpet(pet_id, owner_id, petname, type)
    stmt = "INSERT INTO pet (pet_id, owner_id, petname, type) VALUES (" + pet_id + "," + owner_id + "," + quote(petname) + "," + quote(type) + ");"
    print stmt
    try
        dbexecute stmt
    catch
```

```
            print "Unbale to add pet " + pet_id + " " + 
lasterrorextra
        end try
endsubroutine
```

Program 128: Insert Rows into Database

```
INSERT INTO owner (owner_id, ownername, 
phonenumber) VALUES (1,'Jim','555-3434');
INSERT INTO pet (pet_id, owner_id, petname, 
type) VALUES (1,1,'Spot','Cat');
INSERT INTO pet (pet_id, owner_id, petname, 
type) VALUES (2,1,'Fred','Cat');
INSERT INTO pet (pet_id, owner_id, petname, 
type) VALUES (3,1,'Elvis','Cat');
INSERT INTO owner (owner_id, ownername, 
phonenumber) VALUES (2,'Sue','555-8764');
INSERT INTO pet (pet_id, owner_id, petname, 
type) VALUES (4,2,'Alfred','Dog');
INSERT INTO pet (pet_id, owner_id, petname, 
type) VALUES (5,2,'Fido','Cat');
INSERT INTO owner (owner_id, ownername, 
phonenumber) VALUES (3,'Amy','555-4321');
INSERT INTO pet (pet_id, owner_id, petname, 
type) VALUES (6,3,'Bones','Dog');
INSERT INTO owner (owner_id, ownername, 
phonenumber) VALUES (4,'Dee','555-9659');
INSERT INTO pet (pet_id, owner_id, petname, 
type) VALUES (7,4,'Sam','Goat');
```

Sample Output 128: Insert Rows into Database

```
# dbupdate.kbs - update a database row

include "quote.kbs"

dbopen "pets.sqlite3"
```

```
s$ =  "UPDATE owner SET phonenumber = " + quote("555-5555") + " where owner_id = 1;"
print s$
dbexecute s$
dbclose
```

Program 129: Update Row in a Database

```
UPDATE owner SET phonenumber = '555-5555' where owner_id = 1;
```

Sample Output 129: Update Row in a Database

Retrieving Information from a Database:

So far we have seen how to open, close, and execute a SQL statement that does not return any values. A database would be pretty useless if we could not get information out of it.

The SELECT statement, in the SQL language, allows us to retrieve the desired data. After a SELECT is executed a "record set" is created that contains the rows and columns of data that was extracted from the database. Program 130 shows three different SELECT statements and how the data is read into your BASIC-256 program.

```
# showpetsdb.kbs
# display data from the pets database

dbopen "pets.sqlite3"

# show owners and their phone numbers
print "Owners and Phone Numbers"
dbopenset "SELECT ownername, phonenumber FROM owner ORDER BY ownername;"
while dbrow()
```

```
     print dbstring(0) + " " + dbstring(1)
end while
dbcloseset

print

# show owners and their pets
print "Owners with Pets"
dbopenset "SELECT owner.ownername, pet.pet_id, pet.petname, pet.type FROM owner JOIN pet ON pet.owner_id = owner.owner_id ORDER BY ownername, petname;"
while dbrow()
     print dbstring(0) + " " + dbint(1) + " " + dbstring(2) + " " + dbstring(3)
end while
dbcloseset

print

# show average number of pets
print "Average Number of Pets"
dbopenset "SELECT AVG(c) FROM (SELECT COUNT(*) AS c FROM owner JOIN pet ON pet.owner_id = owner.owner_id GROUP BY owner.owner_id) AS numpets;"
while dbrow()
     print dbfloat(0)
end while
dbcloseset

# wrap everything up
dbclose
```

Program 130: Selecting Sets of Data from a Database

```
Owners and Phone Numbers
Amy 555-9932
Dee 555-4433
```

```
Jim 555-5555
Sue 555-8764

Owners with Pets
Amy 6 Bones Dog
Dee 7 Sam Goat
Jim 3 Elvis Cat
Jim 2 Fred Cat
Jim 1 Spot Cat
Sue 4 Alfred Cat
Sue 5 Fido Dog

Average Number of Pets
1.75
```

Sample Output 130: Selecting Sets of Data from a Database

New Concept

`dbopenset sqlstatement`

Execute a SELECT statement on the database and create a "record set" to allow the program to read in the result. The "record set" may contain 0 or more rows as extracted by the SELECT.

New Concept

`dbrow  or  dbrow ()`

Function to advance the result of the last **dbopenset** to the next row. Returns false if we are at the end of the selected data.

You need to advance to the first row, using **dbrow**, after a **dbopenset** statement before you can read any data.


```
dbint ( column )
dbfloat ( column )
dbstring ( column )
```

These functions will return data from the current row of the record set. You must know the zero based numeric column number of the desired data.

dbint	Return the cell data as an integer.
dbfloat	Return the cell data as a floating-point number.
dbstring	Return the cell data as a string.


```
dbcloseset
```

Close and discard the results of the last **dbopenset** statement.

Big Program

The big program this chapter creates a single program that creates, maintains, and lists phone numbers stored in a database file.

Pay special attention to the quote function used in creating the SQL statements. It wraps all strings in the statements in single quotes after changing the single quotes in a string to a pair of them. This doubling of quotes inside quotes is how to insert a quotation mark in an SQL statement.

```
# rolofile.kbs
# a database example to keep track of phone numbers

include "quote.kbs"

dbopen "rolofile.sqlite3"
call createtables()

do
     print
     print "rolofile - phone numbers"
     print "1-add person"
     print "2-list people"
     print "3-add phone"
     print "4-list phones"
     input "0-exit >", choice
     print

     if choice=1 then call addperson()
     if choice=2 then call listpeople()
     if choice=3 then call addphone()
     if choice=4 then call listphone()
until choice = 0
dbclose
end

function inputphonetype()
```

```
     do
          input "Phone Type (h-home, c-cell, f-fax, w-work) > ", type
     until type = "h" or type = "c" or type = "f" or type = "w"
     return type
end function

subroutine createtables()
     # includes the IF NOT EXISTS clause to not error if the
     # table already exists
     dbexecute "CREATE TABLE IF NOT EXISTS person (person_id TEXT PRIMARY KEY, name TEXT);"
     dbexecute "CREATE TABLE IF NOT EXISTS phone (person_id TEXT, phone TEXT, type TEXT, PRIMARY KEY (person_id, phone));"
end subroutine

subroutine addperson()
     print "add person"
     input "person id > ", person_id
     person_id = upper(person_id)
     if ispersononfile(person_id) or person_id = "" then
          print "person already on file or empty"
     else
          inputstring "person name > ", person_name
          if person_name = "" then
               print "please enter name"
          else
               dbexecute "INSERT INTO person (person_id, name) VALUES (" + quote(person_id) + "," + quote(person_name) + ");"
               print person_id + " added."
          end if
     end if
end subroutine

```

```
subroutine addphone()
    print "add phone number"
    input "person id > ", person_id
    person_id = upper(person_id)
    if not ispersononfile(person_id) then
        print "person not on file"
    else
        inputstring "phone number > ", phone
        if phone = "" then
            print "please enter a phone number"
        else
            type = inputphonetype()
            dbexecute "INSERT INTO phone (person_id, phone, type) values (" + quote(person_id) + "," + quote(phone) + "," + quote(type) + ");"
            print phone + " added."
        end if
    end if
end subroutine

function ispersononfile(person_id)
    # return true/false whether the person is on the person table
    onfile = false
    dbopenset "select person_id from person where person_id = " + quote(person_id)
    if dbrow() then onfile = true
    dbcloseset
    return onfile
end function

subroutine listpeople()
    dbopenset "select person_id, name from person order by person_id"
    while dbrow()
        print dbstring("person_id") + " " + dbstring("name")
    end while
    dbcloseset
```

```
end subroutine

subroutine listphone()
    input "person id to list (return for all) > ", person_id
    person_id = upper(person_id)
    stmt = "SELECT person.person_id, person.name, phone.phone, phone.type FROM person LEFT JOIN phone ON person.person_id = phone.person_id"
    if person_id <> "" then stmt += " WHERE person.person_id = " + quote(person_id)
    stmt += " ORDER BY person.person_id"
    dbopenset stmt
    while dbrow()
        print dbstring("person_id") + " " + dbstring("name") + " " + dbstring("phone") + " " + dbstring("type")
    end while
    dbcloseset
end subroutine
```

Exercises:

Word Search

y	p	z	t	c	e	l	e	s	o	x	x	d
e	l	i	b	a	m	l	n	a	x	x	t	b
t	q	x	h	o	o	e	t	g	n	e	d	i
a	s	t	p	s	t	l	n	e	s	t	f	n
e	e	a	e	a	n	i	f	n	t	t	s	t
r	q	t	d	s	r	o	e	b	m	a	d	r
c	n	p	u	t	e	p	i	n	d	b	m	e
i	u	e	s	c	o	s	m	t	c	l	u	s
d	q	b	p	b	e	u	o	l	a	e	y	n
b	d	u	d	o	l	x	o	l	z	l	f	i
r	m	o	e	o	b	s	e	p	c	w	e	m
o	x	h	c	r	e	d	t	b	o	b	y	r
w	c	g	h	t	y	j	c	r	d	s	d	m

	column, create, dbclose, dbcloseset, dbexecute, dbfloat, dbint, dbopen, dbopenset, dbrow, dbstring, insert, query, relationship, row, select, sql, table, update

Problems	1. Take the “Big Program” from this chapter and modify it to create an application to keep track of a student's grades for several classes. You will need the following menu options to allow the user to: • Enter a class code, assignment name, possible points, score on an assignment and store this information into a database table. • Create a way for the student to see all of the grades for a single class after they enter the class code. • Create an option to see a list of all classes with total points possible, total points scored, and percentage of scored vs. possible.

Chapter 22: Connecting with a Network

This chapter discusses how to use the BASIC-256 networking statements. Networking in BASIC-256 will allow for a simple "socket" connection using TCP (Transmission Control Protocol). This chapter is not meant to be a full introduction to TCP/IP socket programming.

Socket Connection:

TCP stream sockets create a connection between two computers or programs. Packets of information may be sent and received in a bi-directional (or two way) manner over the connection.

To start a connection we need one computer or program to act as a server (to wait for the incoming telephone call) and the other to be a client (to make the telephone call). Illustration 40 shows graphically how a stream connection is made.

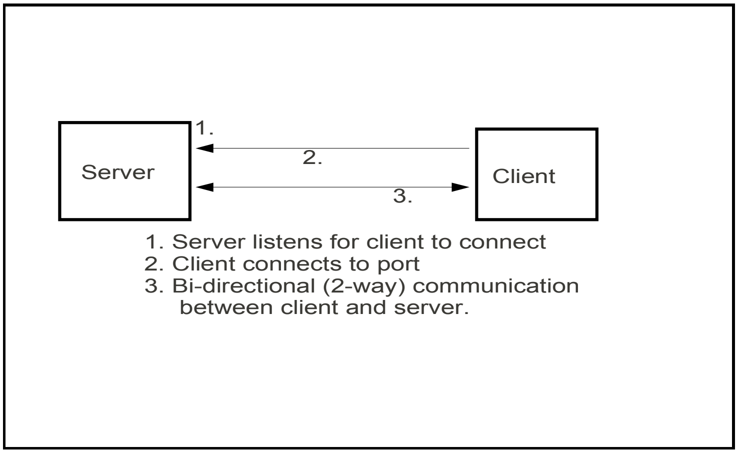


Illustration 40: Socket Communication

Just like with a telephone call, the person making the call (client) needs to know the phone number of the person they are calling (server). We call that number an IP address. BASIC-256 uses IP version 4 addresses that are usually expressed as four numbers separated by periods (A.B.C.D) where A, B, C, and D are integer values from 0 to 255.

In addition to having the IP address for the server, the client and server must also talk to each-other over a port. You can think of the port as a telephone extension in a large company. A person is assigned an extension (port) to answer (server) and if you want to talk to that person you (client) call that extension.

The port number may be between 0 and 65535 but various Internet and other applications have been reserved ports in the range of 0-1023. It is recommended that you avoid using these ports.

A Simple Server and Client:

```
# simpleserver.kbs
# send a message to the client on port 999

print "listening to port 9999 on " + netaddress()
NetListen 9999
NetWrite "The simple server sent this message."
NetClose
```

Program 131: Simple Network Server

```
# simpleclient.kbs
# connect to simple server and get the message
#
input "What is the address of the simple_server?", addr
if addr = "" then addr = "127.0.0.1"
#
```

```
NetConnect addr, 9999
print NetRead
NetClose
```

Program 132: Simple Network Client

```
listening to port 9999 on xx.xx.xx.xx
```

Sample Output 131: Simple Network Server

```
What is the address of the simple_server?
The simple server sent this message.
```

Sample Output 132: Simple Network Client

New Concept

netaddress
netaddress ( )

Function that returns a string containing the numeric IPv4 network address for this machine.

New Concept

netlisten portnumber
netlisten (portnumbrer)
netlisten socketnumber, portnumber
netlisten (socketnumber, portnumber)

Open up a network connection (server) on a specific port address and wait for another program to connect. If socketnumber is not specified socket number zero (0) will be used.

New Concept

```
netclose
netclose ( )
netclose socketnumber
netclose ( socketnumber )
```

Close the specified network connection (socket). If socketnumber is not specified socket number zero (0) will be closed.

New Concept

```
netwrite string
netwrite ( string )
netwrite socketnumber, string
netwrite ( socketnumber, string )
```

Send a string to the specified open network connection. If socketnumberis not specified socket number zero (0) will be written to.

New Concept

```
netconnect servername, portnumber
netconnect ( servername, portnumber )
netconnect socketnumber, servername, portnumber
netconnect ( socketnumber, servername, portnumber
     )
```

Open a network connection (client) to a server. The IP address or host name of a server are specified in the servernameargument, and the specific network port number. If socketnumberis not specified socket number zero (0) will be used for the connection.

New Concept

```
netread
netread ( )
netread ( socketnumber )
```

Read data from the specified network connection and return it as a string. This function is blocking (it will wait until data is received). If socketnumberis not specified socket number zero (0) will be read from.

Network Chat:

This example adds one new function (**netdata**) to the networking statements we have already introduced. Use of this new function will allow our network clients to process other events, like keystrokes, and then read network data only when there is data to be read.

The network chat program (Error: Reference source not found) combines the client and server program into one. If you start the application and it is unable to connect to a server the error is trapped and the program then becomes a server. This is one of many possible methods to allow a single program to fill both roles.

```
# chat.kbs
# use port 9999 for simple chat

input "Chat to address (return for server or local host)?”, addr
if addr = "" then addr = "127.0.0.1"
#
# try to connect to server - if there is not one become one
try
   NetConnect addr, 9999
catch
```

```
      print "starting server - waiting for chat client"
      NetListen 9999
   end try
   print "connected"

   while true
      # get key pressed and send it
      k = key
      if k <> 0 then
         call show(k)
         netwrite string(k)
      end if
      # get key from network and show it
      if NetData() then
         k = int(NetRead())
         call show(k)
      end if
      pause .01
   end while
   end

   subroutine show(keyvalue)
      if keyvalue=16777220 then
         print
      else
         print chr(keyvalue);
      end if
   end subroutine
```

Program 133: Network Chat

The following is observed when the user on the client types the message "HI SERVER" and then the user on the server types "HI CLIENT".

```
Chat to address (return for server or local
host)?
starting server - waiting for chat client
```

```
connected
HI SERVER
HI CLIENT
```

Sample Output 133.1: Network Chat (Server)

```
Chat to address (return for server or local
host)?
connected
HI SERVER
HI CLIENT
```

Sample Output 133.2: Network Chat (Client)

New Concept

```
netdata or netdata()
netdata ( socketnumbr )
```

Returns true if there is network data waiting to be read. This allows for the program to continue operations without waiting for a network packet to arrive.

Big Program

The big program this chapter creates a two player networked tank battle game. Each player is the white tank on their screen and the other player is the black tank. Use the arrow keys to rotate and move. Shoot with the space bar.

```
# battle.kbs
```

```
# uses port 9998 for server

spritedim 4
call tanksprite(0,white) # me
call tanksprite(1,black) # opponent
call projectilesprite(2,blue) # my shot
call projectilesprite(3,red) # opponent shot

kspace = 32
kleft = 16777234
kright = 16777236
kup = 16777235
kdown = 16777237

dr = pi / 20   # direction change
dxy = 2.5 # move speed
shotdxy = 5    # shot move speed
port = 9998    # port to communicate on

print "Tank Battle - You are the white tank."
print "Your mission is to shoot and kill the"
print "black one. Use arrows to move and"
print "space to shoot."
print

input "Are you the server? (y or n)", mode
if mode = "y" then
   print "You are the server. Waiting for a client to connect."
   NetListen port
else
   input "Server Address to connect to (return for local host)?", addr
   if addr = "" then addr = "127.0.0.1"
   NetConnect addr, port
end if

# set my default position and send to my opponent
x = 100
```

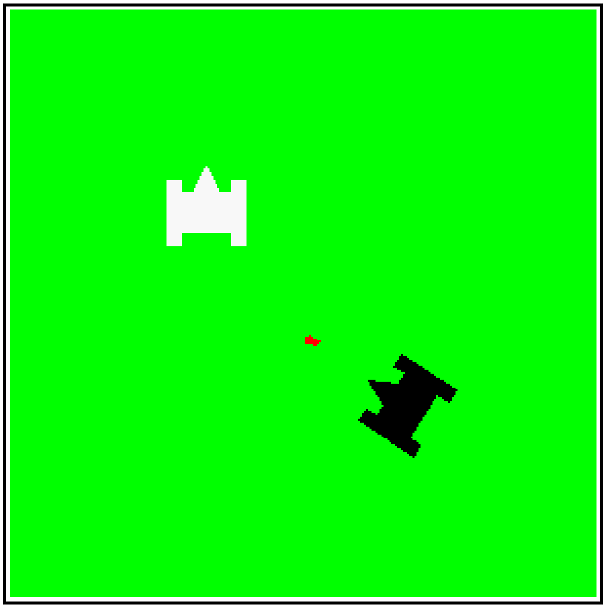
```
y = 100
r = 0
# projectile position direction and visible
p = false
px = 0
py = 0
pr = 0
call writeposition(x,y,r,p,px,py,pr)


# update the screen
color green
rect 0, 0, graphwidth, graphheight
spriteshow 0
spriteshow 1
spriteplace 0, x, y, 1, r
while true
   # get key pressed and move tank on the screen
   k = key
   if k <> 0 then
      if k = kup then
         x = ( graphwidth + x + sin(r) * dxy ) % graphwidth
         y = ( graphheight + y - cos(r) * dxy ) % graphheight
      end if
      if k = kdown then
         x = ( graphwidth + x - sin(r) * dxy ) % graphwidth
         y = ( graphheight + y + cos(r) * dxy ) % graphheight
      end if
      if k = kleft then r = r - dr
      if k = kright then r = r + dr
      if k = kspace then
         pr = r
         px = x
         py = y
         p = true
```

```
         spriteshow 2
      end if
      spriteplace 0, x, y, 1, r
      call writeposition( x, y, r, p, px, py, pr )
      if spritecollide( 0, 1 ) then
         netwrite "F"
         print "You just ran into the other tank and you both died. Game Over."
         end
      end if
   end if
   # move my projectile (if there is one)
   if p then
      px = px + sin( pr ) * shotdxy
      py = py - cos( pr ) * shotdxy
      spriteplace 2, px, py, 1, pr
      if spritecollide( 1, 2 ) then
         NetWrite "W"
         print "You killed your opponent. Game over."
         end
      end if
      if px < 0 or px > graphwidth or py < 0 or py > graphheight then
         p = false
         spritehide 2
      end if
      call writeposition( x, y, r, p, px, py, pr )
   end if
   #
   # get position from network and
   # set location variables for the opponent
   # flip the coordinates as we decode
   while NetData()
      position = NetRead()
      while position <> ""
         if left(position,1) = "W" then
            print "You Died. - Game Over"
            end
         end if
```

```
         if left(position,1) = "F" then
            print "You were hit and you both died. - Game Over"
            end
         end if
         op_x = graphwidth - unpad( ref( position ), 3)
         op_y = graphheight - unpad( ref( position ), 3)
         op_r = pi + unpad( ref( position ), 5)
         op_p = unpad( ref( position ), 1)
         op_px = graphwidth - unpad( ref( position ), 3)
         op_py = graphheight - unpad( ref( position ), 3)
         op_pr = pi + unpad( ref( position ), 5)
         # display opponent
         spriteplace 1, op_x, op_y, 1, op_r
         if op_p then
            spriteshow 3
            spriteplace 3, op_px, op_py, 1, op_pr
         else
            spritehide 3
         end if
      end while
   end while
   #
   pause .05
end while

subroutine writeposition(x,y,r,p,px,py,pr)
   position =  lpad( int( x ), 3 ) + lpad( int( y ),
3 ) + lpad( r, 5 ) + lpad( p, 1 ) + lpad( int( px ),
3 ) + lpad( int( py ), 3 ) + lpad( pr, 5 )
   NetWrite position
end subroutine

function lpad( n, l )
   # return a number left padded in spaces
```

```
   s = left( n, l )
   while length( s ) < l
      s = " " + s
   end while
   return s
end function

function unpad( ref( l ), l )
   # return a number at the begining padded in l spaces
   # and shorten the string by l that we just pulled off
   n = float( left( l, l ) )
   if length( l ) > l then
      l = mid( l, l + 1, 99999 )
   else
      l = ""
   end if
   return n
end function

subroutine tanksprite( spritenumber , c )
   color c
   spritepoly spritenumber, {0,0, 7,0, 7,7, 14,7, 20,0, 26,7, 33,7, 33,0, 40,0, 40,40, 33,40, 33,33, 7,33, 7,40, 0,40}
end subroutine

subroutine projectilesprite( spritenumber, c)
   color c
   spritepoly spritenumber, {3,0, 3,8, 0,8}
end subroutine
```

Program 134: Network Tank Battle



Sample Output 45: Adding Machine - Using Exit While

Exercises:

m	r	d	t	n	s	i	p	n	n
j	r	f	d	o	c	k	e	e	e
v	v	r	c	l	r	t	g	s	t
p	h	k	i	o	s	d	e	o	c
k	e	e	w	i	a	r	k	l	o
t	n	t	l	e	v	v	c	c	n
t	e	t	r	e	t	t	n	t	n
n	e	t	r	c	x	g	o	e	e
n	e	x	p	o	r	t	m	n	c
n	e	t	i	r	w	t	e	n	t

client, listen, netclose, netconnect, netlisten, netread, network, netwrite, port, server, socket, tcp

Problems

1. Modify Problem 4 from the keyboard control chapter to create a network client/server 2 player "ping-pong" game.

2. Write a simple server/client rock-paper-scissors game where two players will compete.

3. Write a complex network chat server that can connect to several clients at once. You will need a server process to assign each client a different port on the server for the actual chat traffic.

Appendix A: Loading BASIC-256 on your Windows PC

This chapter will walk you step by step through downloading and installing BASIC-256 on your Microsoft Windows PC. The instructions are written for Windows XP with Firefox as your Web browser. Your specific configuration and installation may be different but the general steps should be similar.

1 – Download:

Connect to the Internet and navigate to the Web site http://www.basic256.org and follow the download link. Once you are at the Sourceforge project page click on the green "Download Now!"button (Illustration 41) to start the download process.

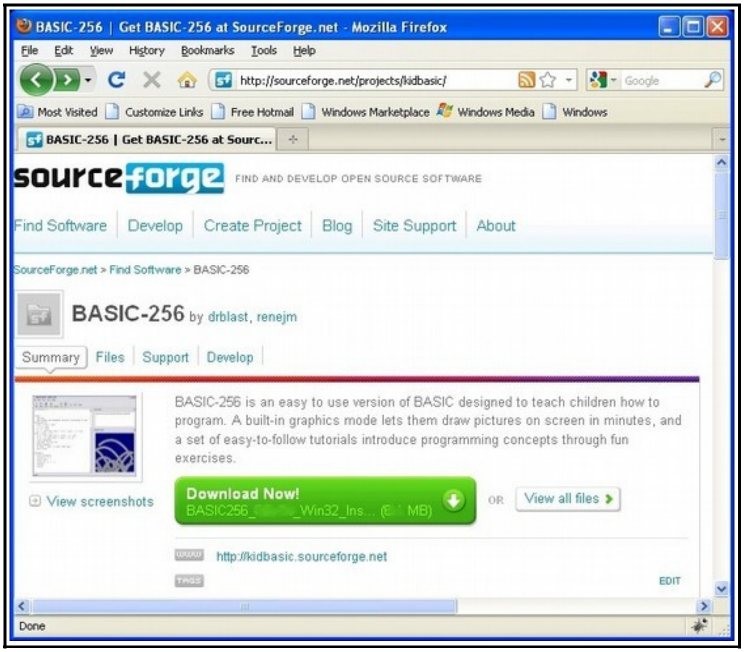


Illustration 41: BASIC-256 on Sourceforge

The download process may ask you what you want to do with the file. Click the "Save File" button (Illustration 42).

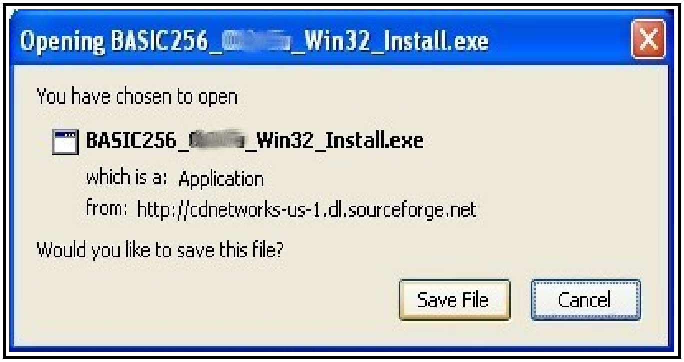


Illustration 42: Saving Install File

Firefox should display the "Downloads" window and actually download the BASIC-256 installer. When it is finished it should look like Illustration 43. Do not close this window quite yet, you will need it to start the Installation.

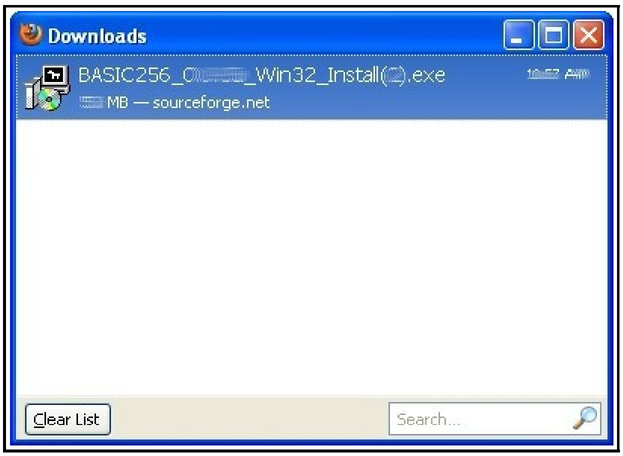


Illustration 43: File Downloaded

2 – Installing:

Once the file has finished downloading (Illustration 43) use your mouse and click on the file from the download list. You will then see one or two dialogs asking if you really want to execute this file (Illustration 44) (Illustration 45). You need to click the "OK" or "Run" buttons on these dialogs.

Illustration 44: Open File Warning

Illustration 45: Open File Security Warning

After the security warnings are cleared you will see the actual BASIC-256 Installer application. Click the "Next>" button on the first screen (Illustration 46).

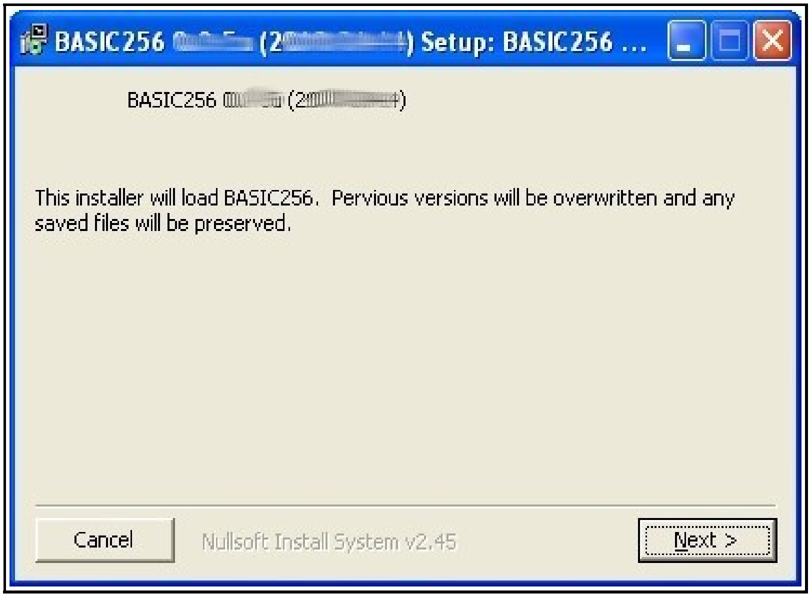


Illustration 46: Installer - Welcome Screen

Read and agree to the GNU GPL software license and click on "I Agree" (Illustration 47). The GNU GPL license is one of the most commonly used "Open Source" and"Free" license to software. You have the right to use, give away, and modify the programs released under the GPL. This license only relates to the BASIC-256 software and not the contents of this book.

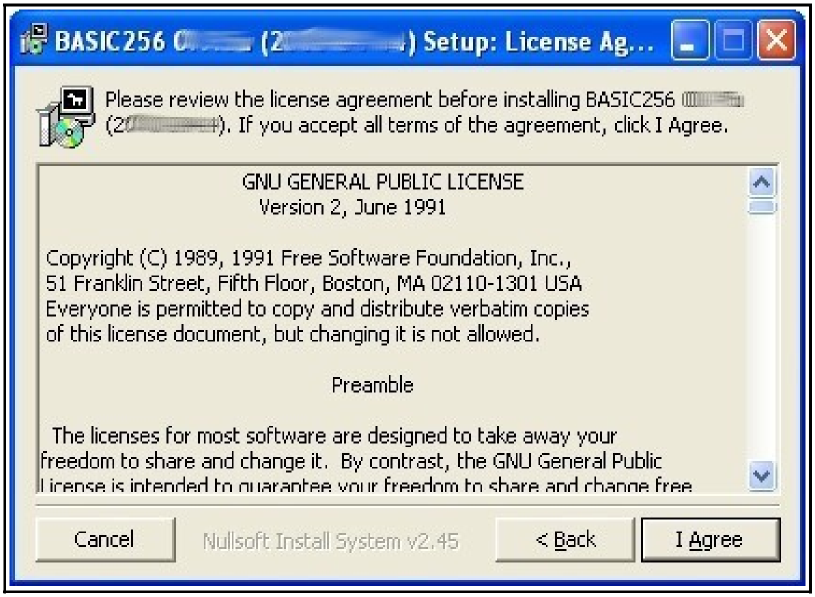


Illustration 47: Installer - GPL License Screen

The next Installer screen asks you what you want to install (Illustration 48). If you are installing BASIC-256 to a USB or other type of removable drive then it is suggested that you un-check the "Start Menu Shortcuts". For most users who are installing to a hard drive, should do a complete install. Click "Next>".

Illustration 48: Installer - What to Install

Illustration 49 shows the last screen before the install begins. This screen asks you what folder to install the BASIC-256 executable files into. If you are installing to your hard drive then you should accept the default path.

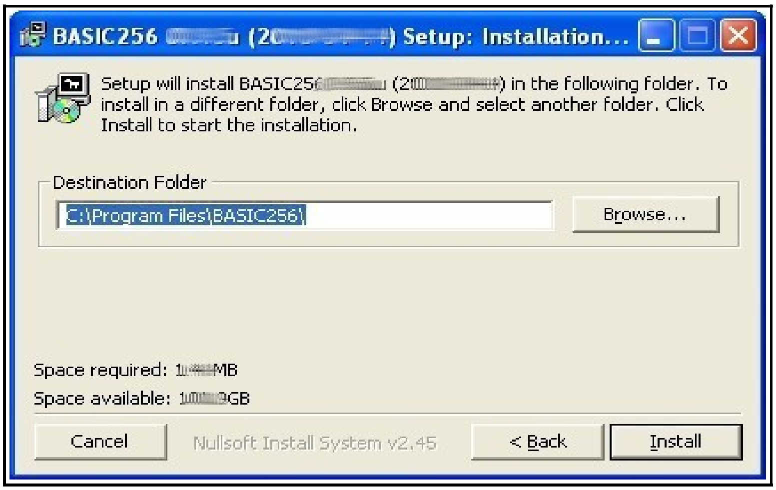


Illustration 49: Installer - Where to Install

The installation is complete when you see this screen (Illustration 50). Click "Close".

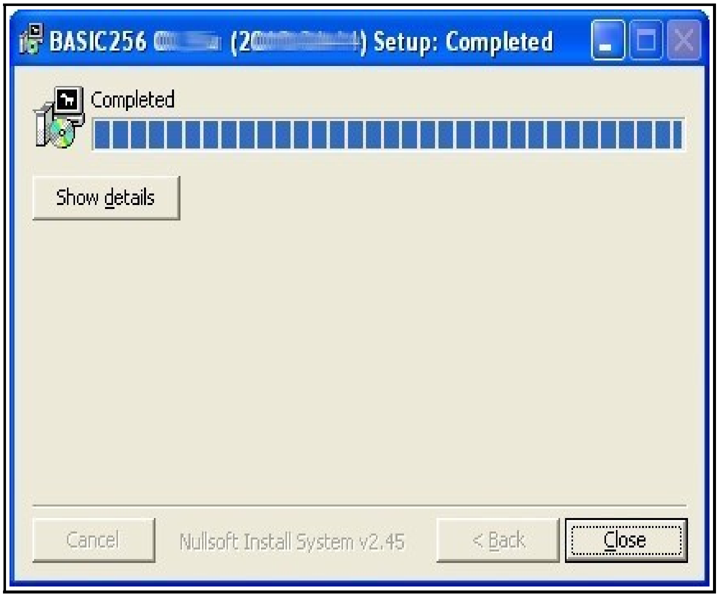


Illustration 50: Installer - Complete

3 – Starting BASIC-256

The installation is complete. You may now click on the Windows "Start" button and then "All Programs >" (Illustration 51).

Illustration 51: XP Start Button

You will then see a menu for BASIC-256. You may open the program by clicking on it, uninstall it, or view the documentation from this menu (Illustration 52).

Illustration 52: BASIC-256 Menu from All Programs

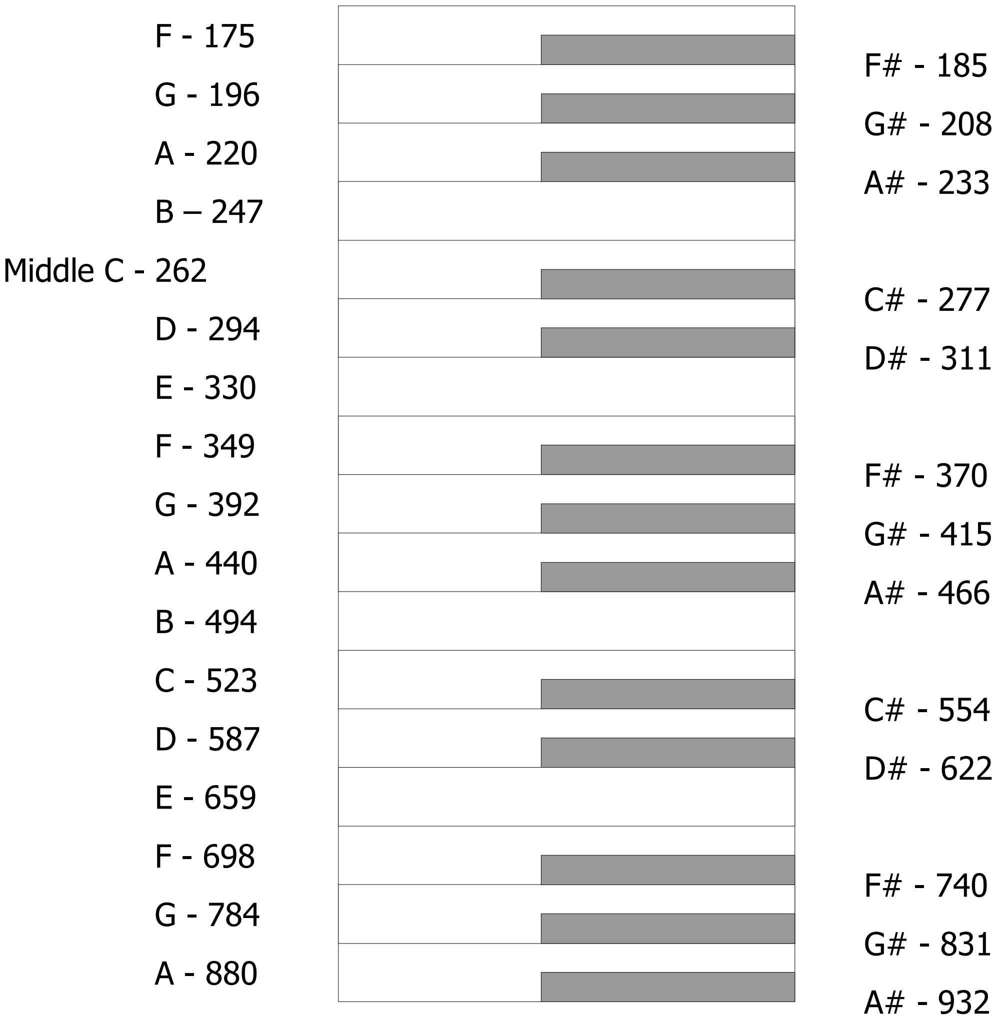
Appendix B: Color Names and Numbers

Listing of standard color names used in the colorstatement. The corresponding RGB values are also listed.

Color	RGB Values	Swatch
black	0, 0, 0	
white	255, 255, 255	
red	255, 0, 0	
darkred	128, 0, 0	
green	0, 255, 0	
darkgreen	0, 128, 0	
blue	0, 0, 255	
darkblue	0, 0, 128	
cyan	0, 255, 255	
darkcyan	0, 128, 128	
purple	255, 0, 255	
darkpurple	128, 0, 128	
yellow	255, 255, 0	
darkyellow	128, 128, 0	
orange	255, 102, 0	
darkorange	176, 61, 0	
gray /grey	160, 160, 160	
darkgray / darkgrey	128, 128, 128	
clear		

Appendix C: Musical Tones

This chart will help you in converting the keys on a piano into frequencies to use in the **sound** statement.



Appendix D: Key Values

Key values are returned by the key()function and represent the last keyboard key pressed since the key was last read. This table lists the commonly used key values for the standard English keyboard. Other key values exist.

English (EN) Keyboard Codes										
Key	**#**		**Key**	**#**		**Key**	**#**		**Key**	**#**
Space	32		A	65		L	76		W	87
0	48		B	66		M	77		X	88
1	49		C	67		N	78		Y	89
2	50		D	68		O	79		Z	90
3	51		E	69		P	80		ESC	16777216
4	52		F	70		Q	81		Backspace	16777219
5	53		G	71		R	82		Enter	16777220
6	54		H	72		S	83		Left Arrow	16777234
7	55		I	73		T	84		Up Arrow	16777235
8	56		J	74		U	85		Right Arrow	16777236
9	57		K	75		V	86		Down Arrow	16777237

Appendix E: Unicode Character Values – Latin (English)

This table shows the Unicode character values for standard Latin (English) letters and symbols. These values correspond with the ASCII values that have been used since the 1960's. Additional character sets are available at http://www.unicode.org.

CHR	#	CHR	#	CHR	#	CHR	#	CHR	#	CHR	#
NUL	0	SYN	22	,	44	B	66	X	88	n	110
SOH	1	ETB	23	-	45	C	67	Y	89	o	111
STX	2	CAN	24	.	46	D	68	Z	90	p	112
ETX	3	EM	25	/	47	E	69	[	91	q	113
ET	4	SUB	26	0	48	F	70	\	92	r	114
ENQ	5	ESC	27	1	49	G	71	]	93	s	115
ACK	6	FS	28	2	50	H	72	^	94	t	116
BEL	7	GS	28	3	51	I	73	_	95	u	117
BS	8	RS	30	4	52	J	74	`	96	v	118
HT	9	US	31	5	53	K	75	a	97	w	119
LF	10	Space	32	6	54	L	76	b	98	x	120
VT	11	!	33	7	55	M	77	c	99	y	121
FF	12	"	34	8	56	N	78	d	100	z	122
CR	13	#	35	9	57	O	79	e	101	{	123
SO	14	$	36	:	58	P	80	f	102	|	124
SI	15	%	37	;	59	Q	81	g	103	}	125
DLE	16	&	38	<	60	R	82	h	104	~	126
DC1	17	'	39	=	61	S	83	i	105	DEL	127
DC2	18	(	40	>	62	T	84	j	106		
DC3	19	)	41	?	63	U	85	k	107		
DC4	20	*	42	@	64	V	86	l	108		
NAK	21	+	43	A	65	W	87	m	109		

0-31 and 127 are non-printable.
Adapted from the Unicode Standard 5.2

Appendix F: Reserved Words

These are the words that the BASIC-256 language uses to perform various tasks. You may not use any of these words for variable names or labels for the GOTO and GOSUB statements

```
#
abs
acos
and
arc
asc
asin
atan
black
blue
call
catch
ceil
changedir
chord
chr
circle
clear
clg
clickb
clickclear
clickx
clicky
close
cls
color
colour
confirm
continue
continuedo
continuefor
continuewhile
cos
count
countx
currentdir
cyan
dark
darkblue
darkcyan
darkgeeen
darkgray
darkgrey
darkorange
darkpurple
darkred
darkyellow
day
dbclose
dbcloseset
dbexecute
dbfloat
dbint
dbnull
dbopen
dbopenset
dbrow
dbstring
debuginfo
degrees
dim
dir
do
editvisible
else
end
endfunction
endif
endsubroutine
endtry
endwhile
eof
```

error_arrayindex
error_arrayindexmissing
error_arraysizelarge
error_arraysizesmall
error_byref
error_byreftype
error_colornumber
error_dbcolno
error_dbconnnumber
error_dbnotopen
error_dbnotset
error_dbnotsetrow
error_dbopen
error_dbquery
error_dbsetnumber
error_divzero
error_filenotopen
error_filenumber
error_fileopen
error_filereset
error_filewrite
error_folder
error_fontsize
error_fontweight
error_for1
error_for2
error_freedb
error_freedbset
error_freefile
error_freenet
error_imagefile
error_imagesavetype
error_imagescale
error_infinity
error_logrange
error_netaccept
error_netbind
error_netconn
error_nethost
error_netnone
error_netread
error_netsock
error_netsocknumber
error_netsockopt
error_netwrite
error_none
error_nonnumeric
error_nosuchvariable
error_notanumber
error_notimplemented
error_penwidth
error_permission
error_polyarray
error_polypoints
error_printernotoff
error_printernoton
error_printeropen
error_putbitformat
error_radix
error_radixstring
error_rgb
error_spritena
error_spritenumber
error_spriteslice
error_strend
error_stringmaxlen
error_strneglen
error_strstart
exists
exitdo
exitfor
exitwhile
exp
explode
explodex
false
fastgraphics
float
floor
font
for
freedb
freedbset
freefile
freenet
frombinary
fromhex
fromoctal
fromradix
getbrushcolor

getcolor
getpenwidth
getsetting
getslice
global
gosub
goto
graphheight
graphsize
graphwidth
gray
green
grey
hour
if
imgload
imgsave
implode
include
input
instr
instrx
int
key
kill
lasterror
lasterrorextra
lasterrorline
lasterrormessage
left
length
line
log
log10
lower
md5
mid
minute
month
mouseb
mousex
mousey
msec
netaddress
netclose
netconnect
netdata
netlisten
netread
netwritenext
next
not
offerror
onerror
open
openb
or
orange
ostype
outputvisible
pause
penwidth
pi
pie
pixel
plot
poly
portin
portout
print
printercancel
printeroff
printeron
printerpage
purple
putslice
radians
rand
read
readbyte
readline
rect
red
redim
ref
refresh
rem
replace
replacex
reset

```
return
rgb
right
say
second
seek
setsetting
sin
size
sound
spritecollide
spritedim
spriteh
spritehide
spriteload
spritemove
spriteplace
spritepoly
spriteshow
spriteslice
spritev
spritew
spritex
spritey
sqr
stamp
step
string
system
tan
text
textheight
textwidth
then
throwerror
to
tobinary
tohex
tooctal
toradix
true
try
until
upper
version
volume
wavplay
wavstop
wavwait
while
white
write
writebyte
writeline
xor
year
yellow
```

Appendix G: Errors and Warnings

Error #		Error Description (EN)
0	ERROR_NONE	
2	ERROR_FOR1	"Illegal FOR – start number > end number"
3	ERROR_FOR2	"Illegal FOR – start number < end number"
5	ERROR_FILENUMBER	"Invalid File Number"
6	ERROR_FILEOPEN	"Unable to open file"
7	ERROR_FILENOTOPEN	"File not open."
8	ERROR_FILEWRITE	"Unable to write to file"
9	ERROR_FILERESET	"Unable to reset file"
10	ERROR_ARRAYSIZELARGE	"Array dimension too large"
11	ERROR_ARRAYSIZESMALL	"Array dimension too small"
12	ERROR_NOSUCHVARIABLE	"Unknown variable"
15	ERROR_ARRAYINDEX	"Array index out of bounds"
16	ERROR_STRNEGLEN	"Substring length less that zero"
17	ERROR_STRSTART	"Starting position less than zero"
18	ERROR_STREND	"String not long enough for given starting character"
19	ERROR_NONNUMERIC	"Non-numeric value in numeric expression"
20	ERROR_RGB	"RGB Color values must be in the range of 0 to 255."
21	ERROR_PUTBITFORMAT	"String input to putbit incorrect."
22	ERROR_POLYARRAY	"Argument not an array for poly()/stamp()"
23	ERROR_POLYPOINTS	"Not enough points in array for poly()/stamp()"
24	ERROR_IMAGEFILE	"Unable to load image file."
25	ERROR_SPRITENUMBER	"Sprite number out of range."
26	ERROR_SPRITENA	"Sprite has not been assigned."
27	ERROR_SPRITESLICE	"Unable to slice image."
28	ERROR_FOLDER	"Invalid directory name."

29	ERROR_INFINITY	"Operation returned infinity."
30	ERROR_DBOPEN	"Unable to open SQLITE database."
31	ERROR_DBQUERY	"Database query error (message follows)."
32	ERROR_DBNOTOPEN	"Database must be opened first."
33	ERROR_DBCOLNO	"Column number out of range."
34	ERROR_DBNOTSET	"Record set must be opened first."
35	ERROR_TYPECONV	“Unable to convert string to number.”
36	ERROR_NETSOCK	"Error opening network socket."
37	ERROR_NETHOST	"Error finding network host."
38	ERROR_NETCONN	"Unable to connect to network host."
39	ERROR_NETREAD	"Unable to read from network connection."
40	ERROR_NETNONE	"Network connection has not been opened."
41	ERROR_NETWRITE	"Unable to write to network connection."
42	ERROR_NETSOCKOPT	"Unable to set network socket options."
43	ERROR_NETBIND	"Unable to bind network socket."
44	ERROR_NETACCEPT	"Unable to accept network connection."
45	ERROR_NETSOCKNUMBER	"Invalid Socket Number"
46	ERROR_PERMISSION	"You do not have permission to use this statement/function."
47	ERROR_IMAGESAVETYPE	"Invalid image save type."
50	ERROR_DIVZERO	"Division by zero"
51	ERROR_BYREF	"Function/Subroutine expecting variable reference in call"
52	ERROR_BYREFTYPE	"Function/Subroutine variable incorrect reference type in call"
53	ERROR_FREEFILE	"There are no free file numbers to allocate"
54	ERROR_FREENET	"There are no free network connections to allocate"
55	ERROR_FREEDB	"There are no free database connections to allocate"
56	ERROR_DBCONNNUMBER	"Invalid Database Connection Number"
57	ERROR_FREEDBSET	"There are no free data sets to allocate for that

		database connection"
58	ERROR_DBSETNUMBER	"Invalid data set number"
59	ERROR_DBNOTSETROW	"You must advance the data set using DBROW before you can read data from it"
60	ERROR_PENWIDTH	"Drawing pen width must be a non-negative number"
61	ERROR_COLORNUMBER	"Color values must be in the range of -1 to 16,777,215"
62	ERROR_ARRAYINDEXMISSING	"Array variable %VARNAME% has no value without an index"
63	ERROR_IMAGESCALE	"Image scale must be greater than or equal to zero"
64	ERROR_FONTSIZE	"Font size, in points, must be greater than or equal to zero"
65	ERROR_FONTWEIGHT	"Font weight must be greater than or equal to zero"
66	ERROR_RADIXSTRING	"Unable to convert radix string back to a decimal number"
67	ERROR_RADIX	"Radix conversion base muse be between 2 and 36"
68	ERROR_LOGRANGE	"Unable to calculate the logarithm or root of a negative number"
69	ERROR_STRINGMAXLEN	"String exceeds maximum length of 16,777,216 characters"
70	ERROR_NOTANUMBER	"Mathematical operation returned an undefined value"
71	ERROR_PRINTERNOTON	"Printer is not on."
72	ERROR_PRINTERNOTOFF	"Printing is already on."
73	ERROR_PRINTEROPEN	"Unable to open printer."
65535	ERROR_NOTIMPLEMENTED	"Feature not implemented in this environment."

WARNING #		**Error Description (EN)**
65537	WARNING_TYPECONV	"Unable to convert string to number, zero used"

Appendix H: Glossary

Glossary of terms used in this book.

algorithm – A step-by-step process for solving a problem.

angle – An angle is formed when two line segments (or rays) start at the same point on a plane. An angle's measurement is the amount of rotation from one ray to another on the plane and is typically expressed in radians or degrees.

argument – A data value included in a statement or function call used to pass information. In BASIC-256 argument values are not changed by the statement or function.

array – A collection of data, stored in the computer's memory, that is accessed by using one or more integer indexes. See also numeric array, one dimensional array, string array, and two dimensional array.

ASCII – (acronym for American Standard Code for Information Interchange) Defines a numeric code used to represent letters and symbols used in the English Language. See also Unicode.

asynchronous – Process or statements happening at one after the other.

Boolean Algebra – The algebra of true/false values created by Charles Boole over 150 years ago.

Cartesian Coordinate System – Uniquely identify a point on a plane by a pair of distances from the origin (0,0). The two distances are measured on perpendicular axes.

column (database) – defines a single piece of information that will be common to all rows of a database table.

constant – A value that can not be changed.

data structure – is a way to store and use information efficiently in a computer system

database – An organized collection of data. Most databases are computerized and consist of tables of similar information that are broken into rows and columns. See also: column, row, SQL, and table.

degrees – A unit of angular measure. Angles on a plane can have measures in degrees of 0 to 360. A right angle is 90 degrees. See also angle and radians.

empty string – A string with no characters and a length of zero (0). Represented by two quotation marks (""). See also string.

false – Boolean value representing not true. In BASIC-256 it is actually short hand for the integer zero (0). See also Boolean Algebra and true.

floating-point number – A numeric value that may or may not contain a decimal point. Typically floating-point numbers have a range of $\pm 1.7 \times 10^{\pm 308}$ with 15 digits of precision.

font – A style of drawing letters.

frequency – The number of occurrences of an event over a specific period of time. See also hertz.

function – A special type of statement in BASIC-256 that may take zero or more values, make calculations, and return information to your program.

graphics output area – The area on the screen where drawing is displayed.

hertz (hz) – Measure of frequency in cycles per second. Named for German physicist Heinrich Hertz. See also frequency.

integer – A numeric value with no decimal point. A whole number. Typically has a range of –2,147,483,648 to 2,147,483,647.

IP address – Short for Internet Protocol address. An IP address is a numeric label assigned to a device on a network.

label – A name associated with a specific place in the program. Used for jumping to with the **goto** and **gosub** statements.

list – A collection of values that can be used to assign arrays and in some statements. In BASIC-256 lists are represented as comma (,) separated values inside a set of curly-braces ({}).

logical error – An error that causes the program to not perform as expected.

named constant – A value that is represented by a name but can not be changed.

numeric array – An array of numbers.

one dimensional array - A structure in memory that holds a list of data that is addressed by a single index. See also array.

operator – Acts upon one or two pieces of data to perform an action.

pixel – Smallest addressable point on a computer display screen.

point – Measurement of text – 1 point = 1/72". A character set in 12 point will be 12/72" or 1/6" tall.

port – A software endpoint number used to create and communicate on a socket.

pseudocode – Description of what a program needs to do in a natural (non-computer) language. This word contains the prefix "pseudo" which means false and "code" for programming text.

radian - A unit of angular measure. Angles on a plane can have measures in radians of 0 to 2π. A right angle is $\pi/2$ degrees. See also angle and degrees.

radius – Distance from a circle to it's center. Also, ½ of a circle's diameter.

RGB – Acronym for Red Green Blue. Light is made up of these three colors.

row (database) – Also called a record or tuple. A row can be thought of as a single member of a table.

socket – A software endpoint that allows for bi-directional (2 way) network communications between two process on a single computer or two computers.

sprite – An image that is integrated into a graphical scene.

SQL – Acronym for Structured Query Language. SQL is the most widely used language to manipulate data in a relational database.

statement – A single complete action. Statements perform something and do not return a value.

string – A sequence of characters (letters, numbers, and symbols). String constants are surrounded by double quotation marks (").

string array – An array of strings.

sub-string – Part of a larger string.

subroutine – A block of code or portion of a larger program that performs a task independently from the rest of the program. A piece that can be used and re-used by many parts of a program.

syntax error – An error with the structure of a statement so that the program will not execute.

synchronous – Happening at the same time.

table (database) – Data organized into rows and columns. A table has a specific number of defined columns and zero or more rows.

transparent – Able to see through.

text output area – The area of the screen where plain text and errors is displayed.

true – Boolean value representing not false. In BASIC-256 it is actually short hand for the integer one (1). See also Boolean Algebra and false.

two dimensional array – A structure in memory that will hold rows and columns of data. See also array.

Unicode – The modern standard used to represent characters and symbols of all the world's languages as integer numbers.

variable – A named storage location in the computer's memory that can be changed or varied. A variable can store an integer, floating-point number, string, or an array.