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:

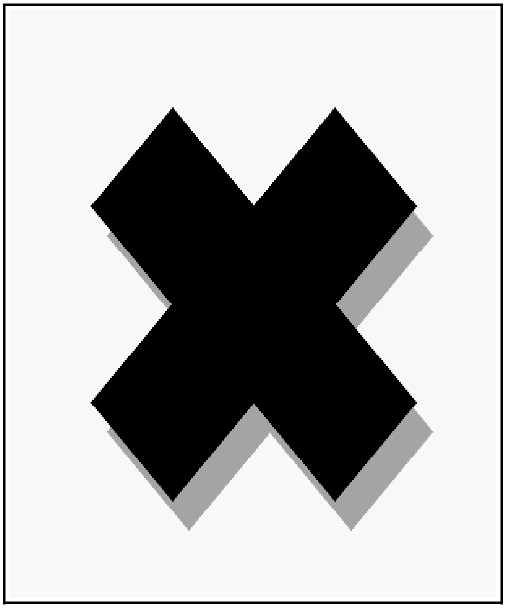
In Chapter 8 we also saw the use of lists for creating polygons and stamps. Arrays may also be used to draw stamps, polygons, and sprites. This may help simplify your code by allowing the same shape to be defined once, stored in an array, and used in various places in your program.

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



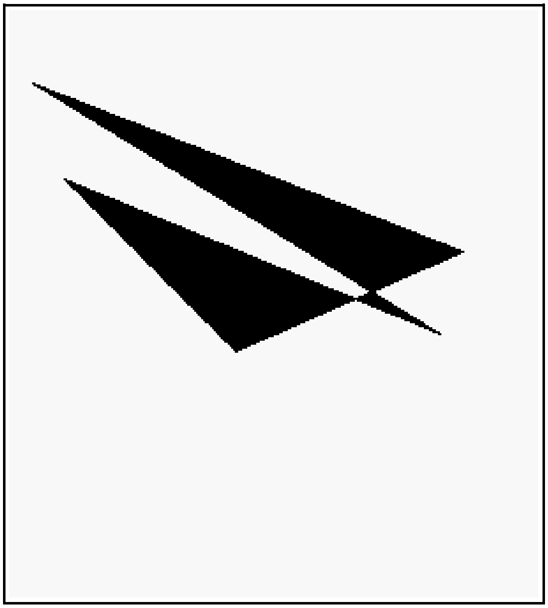
Sample Output 94: Shadow Stamp

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


```
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.


```
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

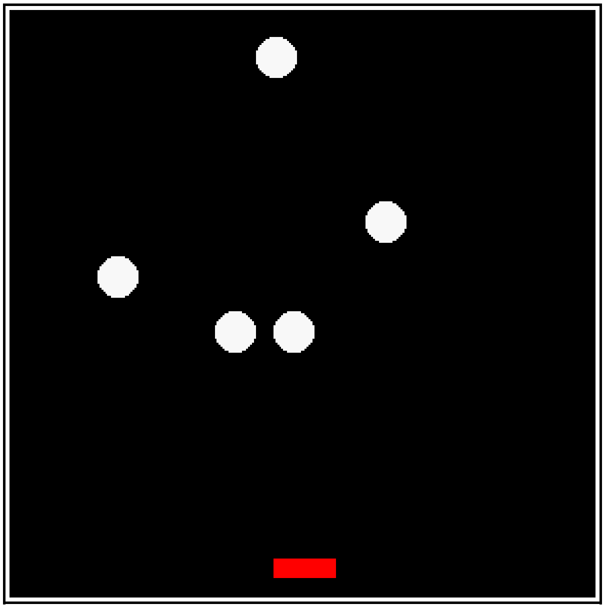
The "Big Program" for this chapter uses three numeric arrays to store the positions and speed of falling space debris. You are not playing pong but you are trying to avoid all of them to score points.

```
# 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

expression1 % expression2

The Modulo (%) operator performs integer division of *expression1* divided by *expression2* and 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 *expression1* to 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 $a = floor(b + 0.5)$.

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.

New Concept	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.

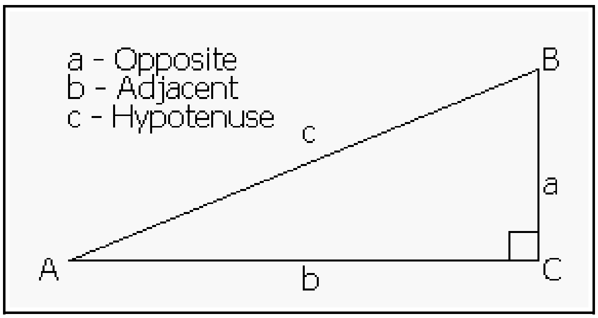


Illustration 24: Right Triangle

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.

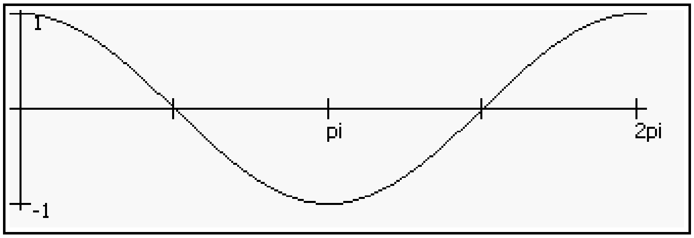


Illustration 25: Cos() Function

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.

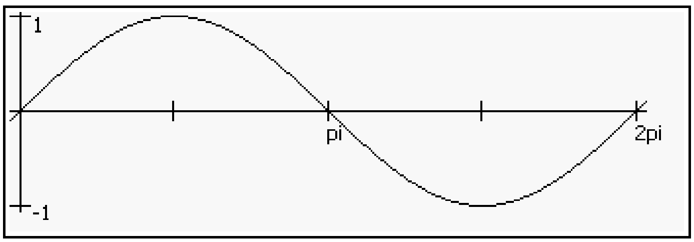


Illustration 26: Sin() Function

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 -∞ 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.

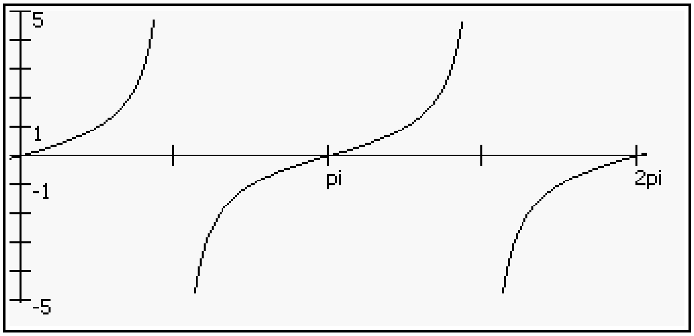


Illustration 27: Tan() Function

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.

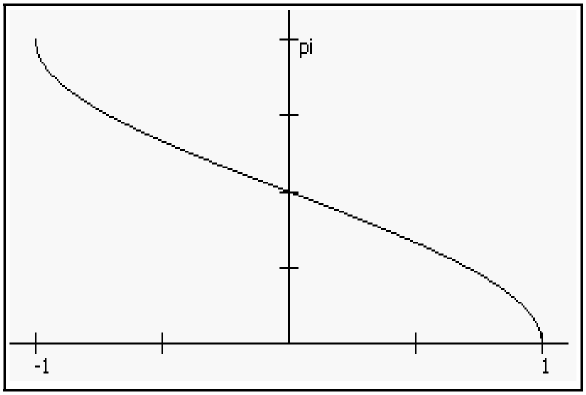


Illustration 28: Acos() Function

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.

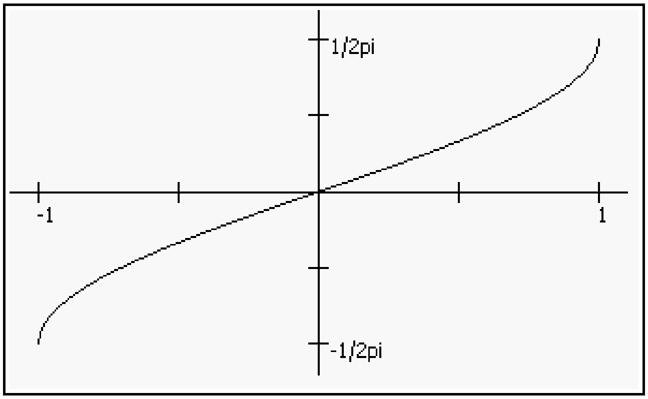


Illustration 29: Asin() Function

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.

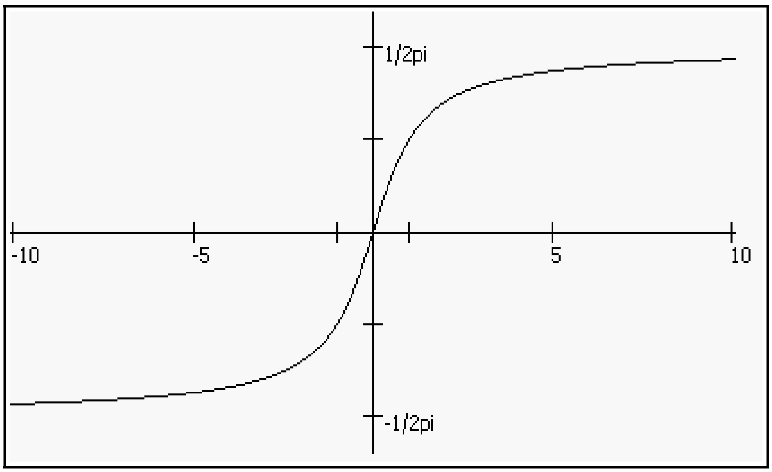


Illustration 30: Atan() Function

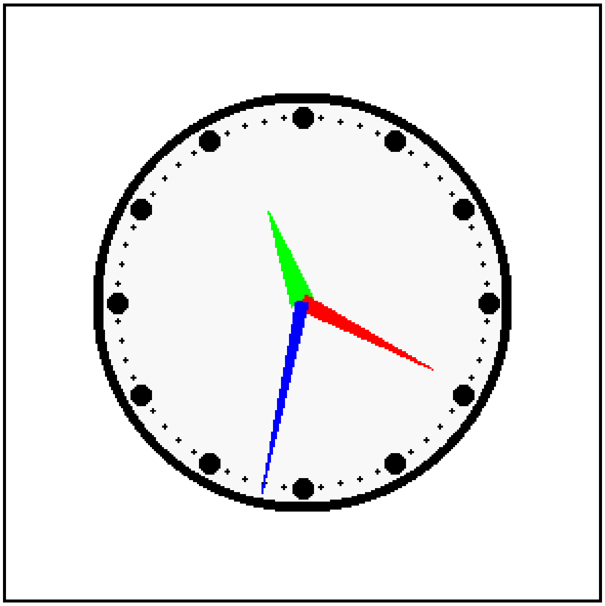
Big Program

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

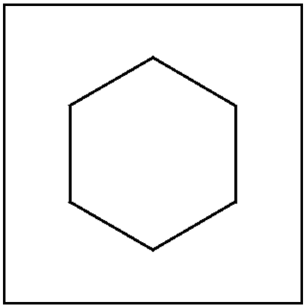
Problems

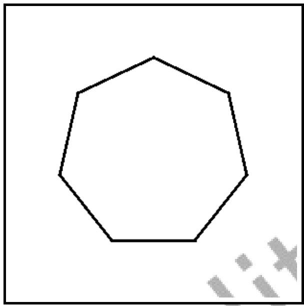
1. Have the user input a decimal number. Display the number it as a whole number and the closest faction over 1000 that is possible.

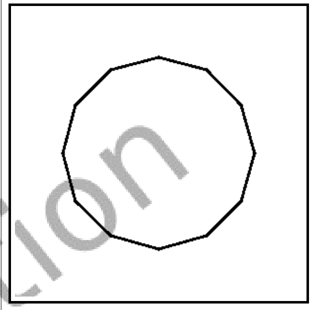
2. Take the program from Problem 1 and use a loop to reduce the fraction by dividing the numerator and denominator by common factors.

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