

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.

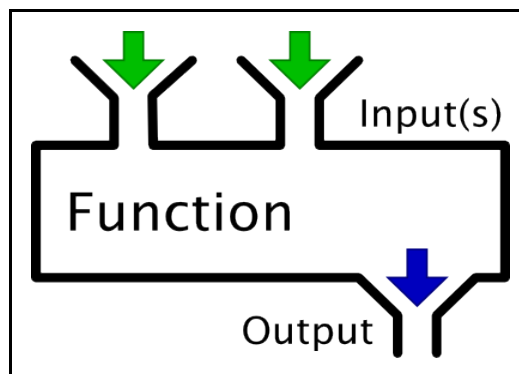


Illustration 22: Block Diagram of a Function



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).

```
1  # minimum.kbs
2  # minimum function
3
4  inputfloat "enter a number ", a
5  inputfloat "enter a second number ", b
6
7  print "the smaller one is ";
8  print minimum(a,b)
9  end
10
11 function minimum(x,y)
12 # return the smallest of the two numbers passed
13     if x<y then return x
14     return y
15 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

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

```

3
4   print "die roller"
5   s = get("sides on the die",6)
6   n = get("number of die", 2)
7   total = 0
8   for x = 1 to n
9       d = die(s)
10      print d
11      total = total + d
12  next x
13  print "total "+ total
14  end
15
16  function get(message, default)
17      # get an integer number
18      # if they press enter or type in a non integer
19      then default to another value
20      input message + " (default " + default + ") ?" ,
21      n
22      if typeof(n) <> 1 then n = default
23      return n
24  end function
25
26  function die(sides)
27      # roll a die and return 1 to sides
28      return int(rand*sides)+1
29  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 it returns a string.

```
1 # repeatstring.kbs
2 # simple string function - make copies
3
4 a = "hi"
5 b = repeat(a,20)
6 print a
7 print b
8 end
9
10 function repeat(word,numberoftimes)
11     result = ""
12     for t = 1 to numberoftimes
13         result := word
14     next t
15     return result
16 end function
```

Program 60: Repeating String Function

```
hi  
hiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
```

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.




New Concept


```
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.

 New Concept	<i>Call subroutinename (value(s))</i>
	The Call statement tells BASIC-256 to transfer program control to the subroutine and pass the values to the subroutine for processing.

 New Concept	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.

```
1  # subroutineclock.kbs
2  # display a comple ticking clock
3
4  fastgraphics
5  font "Tahoma", 20, 100
6  color blue
7  rect 0, 0, 300, 300
8  color yellow
9  text 0, 0, "My Clock."
10
11 while true
12     call displaytime()
13     pause 1.0
14 end while
15
16 end
```

```
17
18 subroutine displaytime()
19     color blue
20     rect 100, 100, 200, 100
21     color yellow
22     text 100, 100, padtwo(hour) + ":" +
        padtwo(minute) + ":" + padtwo(second)
23     refresh
24 end subroutine
25
26 function padtwo(x)
27     # if x is a single digit then prepend a zero
28     if x < 10 then x = "0"+x
29     return x
30 end function
```

Program 61: Subroutine Clock



Sample Output 61: Subroutine Clock



New Concept

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.

```

1  ## subroutine clockimproved.kbs
2  # better ticking clock
3
4  fastgraphics
5  font "Tahoma", 20, 100
6  clg blue
7
8  call displaydate()
9  while true
10     call displaytime()
11     pause 1.0

```

```
12     end while
13
14     end
15
16     subroutine displaydate()
17         # draw over old date
18         color blue
19         rect 50,50, 200, 100
20         # draw new date
21         color yellow
22         text 50,50, padnumber(month) + "/" +
padnumber(day) + "/" + padnumber(year)
23         refresh
24     end subroutine
25
26     subroutine displaytime()
27         # draw over old time
28         color blue
29         rect 50,100, 200, 100
30         #draw new time
31         color yellow
32         text 50, 100, padnumber(hour) + ":" +
padnumber(minute) + ":" + padnumber(second)
33         refresh
34     end subroutine
35
36     function padnumber(n)
37         if n < 10 then n = "0" + n
38         return n
39     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.

```
1  # gamerollerinclude.kbs
2  # Game Dice Roller
3
4  include "diefunction.kbs"
5  include "getintegerfunction.kbs"
6
7  print "die roller with included functions"
8  s = getinteger("sides on the die",6)
9  n = getinteger("number of die",2)
10 total = 0
11
12 for x = 1 to n
13     d = die(s)
```

```
14     print d
15     total = total + d
16 next x
17 print "total "+ total
18 end
```

Program 63: Game Dice Roller – With Included Functions

```
1 # diefunction.kbs
2 # function to roll a N sided die
3
4 function die(sides)
5     return int(rand*sides)+1
6 end function
```

Program 64: Game Dice Roller – die Function

```
1 # getintegerfunction.kbs
2 # get an integer number
3 # if they press enter or type in a non integer then
  default to another value
4
5 function getinteger(message, default)
6     input message + " (default " + default + ") ?" ,
  n
7     if typeof(n) <> TYPE_INT then n = default
8     return n
9 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.

```
1  # addingmachine.kbs
2  # create a nice adding machine
3
4  include "getintegerfunction.kbs"
5
6  print "adding machine"
7  print "press stop to end"
8
9  total = 0
10 while true
11     a = getinteger("+ ",0)
12     total = total + a
13     print total
14 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



New Concept

```
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 *goto* statement.


```
1 # goto.kbs
2 top:
3 print "hi"
4 goto top
```

Program 67: Goto With a Label

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

Sample Output 67: Goto With a Label

 <p>New Concept</p>	<p><i>label:</i></p> <p>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.</p> <p>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.</p> <p>Examples of valid labels include: top:, far999:, and About:.</p>
---	--

 <p>New Concept</p>	<p><i>goto label</i></p> <p>The goto statement causes the execution to jump to the statement directly following the label.</p>
--	--

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.

```
1  # gosub.kbs
2  # a simple gosub
3
4  a = 10
5  for t = 1 to 3
6      print "a equals " + a
7      gosub showline
8  next t
9  end
10
11 showline:
12 print "-----"
13 a = a * 2
14 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*.



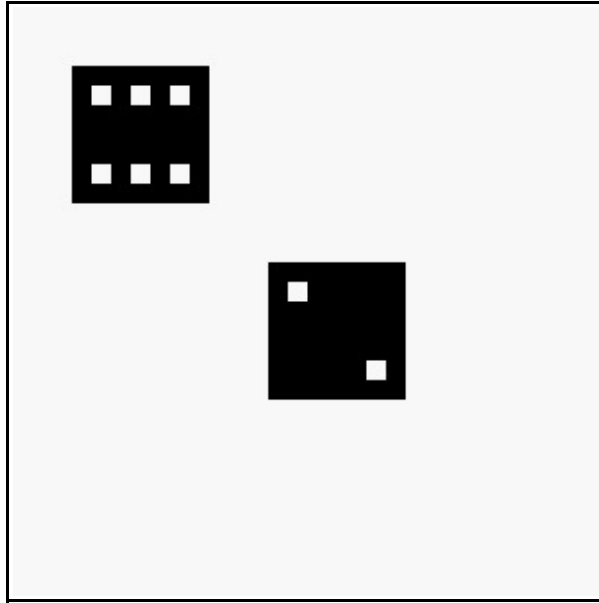
Big Program

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.

```
1  # rollgraphicaldice.kbs
2  # roll two dice graphically
3
4  include "diefunction.kbs"
5
6  clg
7  total = 0
8
9  roll = die(6)
10 total = total + roll
11 call drawdie(30,30, roll)
12
13 roll = die(6)
14 total = total + roll
15 call drawdie(130,130, roll)
16
17 print "you rolled " + total + "."
```

```
18     end
19
20     subroutine drawdie(x,y,n)
21         # draw 70x70 with dots 10x10 pixels
22         # set x,y for top left and n for number of dots
23         color black
24         rect x,y,70,70
25         color white
26         # top row
27         if n <> 1 then rect x + 10, y + 10, 10, 10
28         if n = 6 then rect x + 30, y + 10, 10, 10
29         if n >= 4 and n <= 6 then rect x + 50, y + 10,
30         10, 10
31         # middle
32         if n = 1 or n = 3 or n = 5 then rect x + 30, y +
33         30, 10, 10
34         # bottom row
35         if n >= 4 and n <= 6 then rect x + 10, y + 50,
36         10, 10
37         if n <> 1 then rect x + 50, y + 50, 10, 10
38         if n = 6 then rect x + 30, y + 50, 10, 10
39     end subroutine
```


Program 69: Big Program - Roll Two Dice Graphically




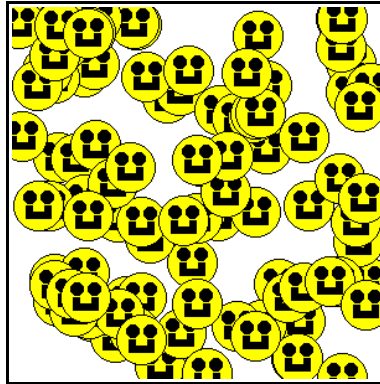
Sample Output 69: Big Program - Roll Two Dice Graphically

Free eBook

Exercises:

 <p>Word Search</p>	<pre> 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 </pre> <p>argument, call, day, end, file, function, gosub, goto, hour, include, label, minute, month, parenthesis, return, second, subroutine, terminate, year</p>
---	--

 <p>Problems</p>	<ol style="list-style-type: none"> 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 x_1, y_1 and x_2, y_2 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 $\frac{y_1 - y_2}{x_1 - x_2}$. 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 $n * (n-1) * (n-2) * \dots * 3 * 2 * 1$ 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:

```
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
```

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.