SUBROUTINES

Subroutine: a completely separate section of code that we `CALL` from our main program or another *subroutine*.

Until now, we've been too busy concentrating on the individual language elements to worry about *subroutines*, but it's time for us to improve the way we think about programs.

One of the best ways to begin a programming problem is to start with the overall program's goal then break it into several smaller modules. We should never lose sight of the overall goal of the program, but we should try to think of how the individual pieces fit together to accomplish that overall goal.

When we finally begin coding the problem, we need to continue to think in terms of how those pieces fit together. Don't approach a problem as though it were one giant program; rather, continue to write the small pieces individually.

This does not mean we should write separate programs to do everything. We can keep individual pieces of the overall program together if we utilize *subroutines* (sections of programs that we can execute repeatedly). Many good programmers write programs that consist solely of *subroutines*, even if the program is to execute one or more of the *subroutines* only once.

Subroutines format

```
SUB subroutine name [(p1, p2, … pn)] [STATIC]
     Block of QBasic statements
END SUB
```

(***note***: It is a common programming convention to indent the body of your *subroutines*.)

- `SUB`: this reserve word designating the beginning of a *subroutine*.
- *subroutine name*: The name of the *subroutine* – up to 40 characters long.
- (p_1, p_2, … p_n)*:* An optional list of ***parameters*** (variables), enclosed in parenthesis, and separated by commas, that show the number and type of ***arguments*** to be passed to the *subroutine* from the `CALL` statement.
- `STATIC`: optional: if present, the `STATIC` reserved word instructs the PC to retain the values of the local variables between calls. If `STATIC` is not present, then local variables are re-initialized to zero or a null strings each time the *subroutine* is called.
- `END SUB`: The reserved words that cause control to transfer from the *subroutine* back to the first executable statement immediately following the `CALL` statement in the main program or another *subroutine/function* that CALLED (invoked) the *subroutine*.

Using the QBasic editor to enter subroutines

The QBasic editor includes all the tools we need to enter and modify *subroutines*. To initiate a *subroutine*, we click on `Edit` on the menu bar and then click `New SUB`. QBasic displays a dialog box requesting that we enter the *subroutine's* name. A *subroutine's* name can be up to 40 characters in length and should follow the naming conventions applicable to variable names. (As a naming convention we should begin all *subroutine* names with an upper case letter). After entering the *subroutine's* name, we click `OK` or press the *Enter* key. QBasic automatically includes the `SUB` and `END SUB` statement. We can press the *Enter* key to insert a blank line, thus making room for our *subroutine* code.

A *subroutine* is run (invoked) by a `CALL` statement. The keyword `CALL` is followed immediately by the *subroutines* name, to which control is transferred. Once control transfers, the instructions inside the *subroutine* are executed one after another until the `END SUB` statement is encountered. The `END SUB` statement transfers control back to the next executable statement line following the corresponding `CALL` statement in the superior (CALLING) program or *subroutine*.

It is a common programming practice to create what are referred to as *stub subroutines* when we begin coding a program. A *stub subroutine* is a skeleton version of the final *subroutine*. It does not contain details or fulfill the necessary program task(s). It simply exists as the target of a `CALL` statement. As programmers we will often place a *dummy* `PRINT` statement inside a *stub subroutine* to begin with. Through the use of these *dummy* `PRINT` statements, *stub subroutines* help demonstrate that our program flows as intended. With *stub subroutines*, we can test and debug a program as it is being built. When the program is complete and functional, our *dummy* `PRINT` statement(s) are removed.

Editing Subroutines

After entering the main program and the associated *subroutine*, the view window will display the last *subroutine* entered. At this point, we may want to review and correct entries in the main program and the associated *subroutine*. QBasic allows us to display and edit the different units of code in three ways:

1. Click `View` on the menu bar, click `SUBs` and then double-click the name of the unit we want to display.
2. Press F2 to display the `SUBs` dialog box, and then double-click the name of the unit we want to display.
3. Press Shift+F2 to display the next *subroutine* in alphabetical order by name. Press Ctrl+F2 to display the previous *subroutine* in reverse alphabetical order by name.

Parameters and Arguments

Note: The list of variable names, found inside parenthesis, following the `CALL` keyword and the *subroutine's* name are referred to as ***arguments***.
The list of variable names, found inside parenthesis, following the `SUB NAME` in the *subroutine* are referred to as ***parameters***.

When a CALL is made, ***arguments*** are assigned to the corresponding ***parameters*** in the SUB statement. This means that the first ***argument*** is passed to the first ***parameter***, the second ***argument*** to the second ***parameter***, and so forth. The number of ***arguments*** in the CALL statement must agree exactly in type and number with the ***parameters*** in the SUB statement. This capability of passing only the required values to a *subroutine* is considered to be one of the major advantages over the SHARED and COMMON SHARRED statements.

A ***parameter*** in a SUB statement must be a variable, an ***argument*** in a CALL statement can be a constant, variable, expression, array element, or an entire array.

Passing No Values to a SUB

Not all *subroutines* require that the CALL statement pass ***arguments*** to a *subroutine*. It is perfectly acceptable to have an empty ***parameter*** list for a *subroutine* but, if the *subroutine's* ***parameter*** list is empty the CALL statement's ***argument*** list must be omitted or empty as well.

Passing Constants and Expressions

Both constants and expression can appear in the argument list of a CALL statement. Numeric constants and numeric expressions can be passed only to numeric variables. If an attempt is made to pass a numeric value to a string variable, the PC displays a dialog box with the error message Type mismatch before terminating execution of the program.

It is a rule in QBasic that ***arguments*** in the CALL statement must agree with the corresponding ***parameters*** in number and type. If an ***argument*** and a ***parameter*** are numeric, but do not agree in type, then the ***argument*** is forced to agree with the ***parameter***.

Passing arguments

Passing by Reference/Address (default)

By default, all QBasic variable ***arguments*** are *passed by reference*. Sometimes, this is called being *passed by address*. When an ***argument*** (a local variable) is *passed by address*, the variable's address in memory is sent to, and assigned to, the receiving routine's ***parameter*** list. (If more than one variable is *passed by address*, each address is sent to and assigned to the receiving *function's* corresponding ***parameter***.)

When you instruct QBasic to define a variable, you are telling QBasic to find an available spot in memory and to assign that memory's address to the variable name. When your program prints or uses the variable, QBasic knows to go to the variable's address and print what is there.

When a variable is *passed by reference*, the address of the variable is copied to the receiving routine. Any time a variable is *passed by reference* (as are all variables by default in QBasic), if you change the *local variable* in the receiving routine, the variable also changes in the *calling routine*.

Passing by Value

Although *passing* ***arguments*** *by value* is not the default in QBasic, the procedure is safer than *passing* ***arguments*** *by reference*. Sometimes *passing* ***arguments*** *by value* is called *passing by copy*. When we *pass* a variable *by value*, the receiving *subroutine* or *function* can change the local copy. When it does, however, the variable (memory location) is not changed in the calling procedure.

Any time we need to pass ***arguments*** to a ***function*** and return one value, we should consider *passing* the arguments *by value*, if the *function* needs to change the value in both the calling and receiving *functions*; we need to stick with the default *passing by reference*.

To *pass* a variable *by value*, we simply enclose it in parentheses inside the ***parameter*** list. For example:

```
CalcWages( (hours), (rate), (taxRate) )
```

Note: All QBasic intrinsic *functions*, such as `INT()` and `LEFT$()`..., assume that their values are *passed by value*, although you do not have to send them ***arguments*** individually enclosed in parentheses.

Passing Values between the Main Program and Subroutines or from one Subroutines to another Subroutine (this method should be avoided when possible)

An alternative to passing values between the Main Program and *subroutines* using the ***argument-parameter*** -list technique is to use the `SHARED` reserved word within the *subroutine* to share the values between the *subroutine* and the Main Program. A `SHARED` statement or a group of `SHARED` statements placed immediately after the `SUB` statement, at the beginning of a *subroutine,* can be used to declare a list of variables that we want to share in either direction between the Main Program and that specific *subroutine*.

In QBasic we cannot share values between *subroutines*; that is, the value of a variable created in one *subroutine* cannot be `SHARED` with another *subroutine*. This leads to the following rule:

The `SHARED` statement can be used only to share variables defined in the Main Program or the *subroutine* in which the `SHARED` statement resides. The `SHARED` statement cannot be used to share variables between two *subroutines*.

MAIN PROGRAM

```
DECLARE SUB EnterData()
CLS
DIM nums(1 TO 5) AS INTEGER
DIM max AS INTEGER
max = 4
CALL EnterData
END
```

SUB ROUTINE

```
SUB EnterData
     SHARED nums() AS INTEGER, max AS INTEGER
     DIM ctr AS INTEGER

     FOR ctr = 1 to max + 1
          INPUT "Enter a number: "; nums(ctr)
     NEXT ctr

END SUB
```

At first we may find the concept of passing variables a bit confusing. So, we might begin writing programs using what is known as the `COMMON SHARED` statement. However, *passing by value* makes a program more efficient and less prone to error because variables are created and stored only when needed.

Variable SCOPE (LOCAL and GLOBAL)

Variable scope (sometimes called visibility of variables) describes how variables are "seen" by our program. Some variables are ***global*** in scope so they can be seen from (and used by) every statement in the program. Some variables are ***local*** in scope so they can be seen from (and used by) only the code in which they were defined.

Note: If we use no *subroutine* or *function* procedures, the concept of ***local*** and ***global variable*** is moot (every variable would be local to the `MAIN` program).

With *subroutines* and *functions*, variables are by default ***local variables***; that is, available only to the *subroutine* that created them. To share variables between the main program and all associated *subroutines*, we might use the `SHARED` statement in the subroutine/function or the less common `COMMON SHARED` statement in the main program to create ***global variables***. In other words, the `COMMON SHARED` statement is used to identify ***global variables*** in our program, that is, variables that are to be shared with all associated *subroutines* (a risky proposition).

The `SHARED` statement works just like its counterpart `DIM`, without the `SHARED` keyword. `DIM` and `SHARED` both allocate memory for variables and arrays, as well as define these variables' types.

The **seldom used** `COMMON SHARED` statement works in a similar manner to the `SHARED` statement. Its purpose is to declare within the `MAIN` program, which variables are to be ***global*** (visible to the entire program). If we choose to put an array variable in a `COMMON SHARED` statement, we still must use `DIM` to declare the number of elements the array will use. We should never put subscripts in the parentheses of `COMMON SHARED` array variables. All `COMMON SHARED` statements must be located at the beginning of the main program before any executable lines of code.

Note: *constants*, defined with the `CONST` statement, in the `MAIN` program are always ***global*** in scope.

The proper method of making variables available to *subroutines/functions* is to *pass* the values of variables to the *subroutine/function* that needs them rather than making all variables ***global*** with the `COMMON SHARED` statement. Values are then passed to *subroutine/function* by assigning them to the ***arguments*** through the `CALL` statement.

User-Defined Functions

Functions are very similar to *subroutines* with one exception, *functions* always have a return value.

In addition to the intrinsic numeric and string *functions* provided by QBasic, QBasic allows us to define our own *functions* that relate to a particular application. This type of *function*, known as a *user-defined function*, is written directly into the program. A *user-defined function* is written as a series of statements similar to a *subroutine*. One way QBasic recognizes a *user-defined function* is by the `FUNCTION` and `END FUNCTION` statements. *Functions* are `CALL`ed by typing their name in an assignment statement, `PRINT` statement or an expression. If there are any ***arguments*** to pass to the *function*, they are passed in the same fashion as we passed them to *subroutines*. *Function* **arguments** can be simple variables, expressions, constants, array elements, or entire arrays.

Inside the *function* definition, at least ones statement must assign a value to the *function* name as a *return value*. Following are some rules concerning *functions*:

1. *Functions* are defined as a distinct unit of code in the same fashion as a *subroutine*.
2. *Function* names cannot begin with `FN`.
3. *Function* names must end with one of the 5 data type symbols.
4. Variable(s) declared inside a *function* are local unless `SHARED`.
5. *Function* **parameter** lists can receive values using the *pass-by-value* or *pass-by-reference* method.
6. A *Function* can reference itself (this is called *recursion*).
7. Single-line *functions* are not allowed in QBasic.

Functions differ from *subroutines* in that <u>*functions* always return a value to the CALLing statement</u>. The data type of the return value is determined by the data type of the *function* name. Therefore <u>*function* names should have one of the data type symbols appended to them</u>.

```
%    INTEGER
!    SINGLE
&    LONG INTEGER
#    DOUBLE PRECISION
$    STRING
```

The format of the FUNCTION statement is as follows:

```
FUNCTION name [ (parameter list ) ]
     Block of QBasic statements
     Function name = expression
END FUNCTION
```

Just as with *subroutines*, any variables that the *function* shares with the main program must be declared with a `SHARED` statement, or passed to the *function* as an argument. For *function* calls the `CALL` keyword is assumed by QBasic so all we need to do, in order to `CALL` a *function*, is to enter the *function*'s name and its parameter list. Therefore, we never need to use the `CALL` keyword to call a *function* in QBasic.

Remember that a *functions* return a value, and we must do something with that value. Therefore, we must do one of the following:

1. assign a *function* call to a variable
2. use it in an expression
3. print its results.

Using the QBasic Editor to enter Functions

We initiate a *function* by selecting the New `FUNCTION` command from the Edit menu. To edit a *function* that is not active in the view window, we use the same commands that we used for *subroutines*. That is, we select a specific *function* by using the `SUBs` command on the View menu.

Like *subroutines*, *functions* are saved to disk with the Main Program under a single file name. When we save the program, QBasic appends a `DECLARE` statement at the beginning of the Main Program for each associated *function*. When the Main Program is loaded from disk into main memory, any associated *functions* and *subroutines* are also loaded into main memory.