Subroutines, Functions, and Modularity

By Howard Fosdick

Overview

Rexx fully supports structured programming. It encourages *modularity* — breaking up large, complex programs into a set of small, simple, interacting components or pieces. These components feature well-defined interfaces that render their interaction clear. Modularity underlies good program structure. Modularity means more easily understood and maintained programs than ill-designed "spaghetti" code, which can quickly become unmaintainable on large programming projects. Structured programming practices and modularity together reduce error rates and produce more reliable code.

Rexx provides the full range of techniques to invoke other programs and to create subroutines and functions. The basic concept is that there should be ways to link together any code you create, buy, or reuse. This is one of the fundamental advantages to using a "glue" language like Rexx.

With Rexx, you can develop large, modular programs that invoke routines written in Rexx or other languages, which issue operating system commands and utilize functions packaged in external function libraries. This article describes the basic ways in which one writes modular Rexx programs.

This article investigates how to write internal subroutines and functions, and how to call them from within the main program. Passing arguments or values into subroutines is an important issue, as is the ability to pass changed values back to the calling program. Variable *scoping* refers to the span of code from within which variables can be changed. This article explores the rules of scoping and how they affect the manner in which scripts are coded. Finally, we introduce the idea of *recursion*, a routine that calls itself as its own subroutine. While this may at first seem confusing, in fact it is a simple technique that clearly expresses certain kinds of algorithms. Not all programming languages support recursion; Rexx does. The article includes a brief script that illustrates how recursion operates.

[Excerpted from *The Rexx Programmer's Reference*, *(*Wiley, 2005) by Howard Fosdick]

Chapter 8

The Building Blocks

As Figure 8-1 shows, any Rexx script can invoke either *internal* or *external* routines. *Internal* means that the code resides in the same file as the script that *calls* or invokes the routines. Those routines that are *external* reside in some file other than that of the invoking script.

How Rexx Supports Modularity

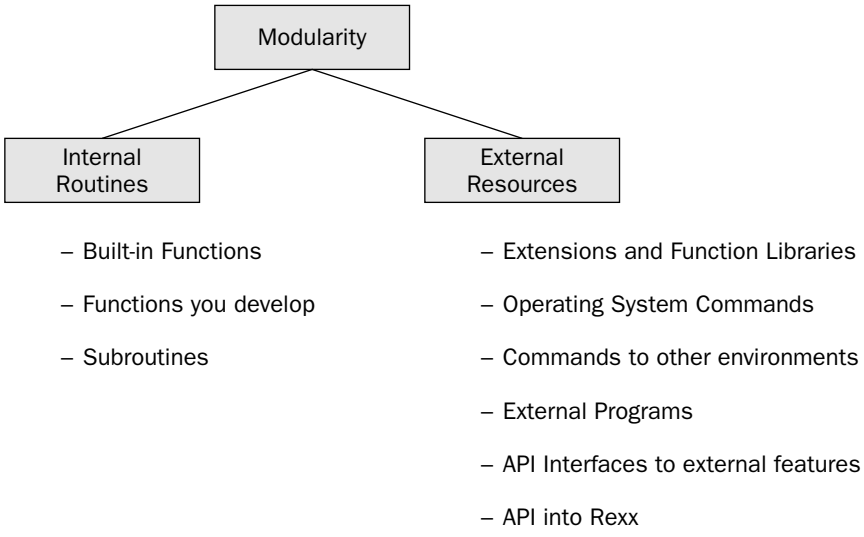

Figure 8-1

Internal routines are classified as either *functions* or *subroutines*. Functions include those that are provided as part of the Rexx language (the *built-in functions*) and those that you write yourself (*user-defined functions*). Functions are distinct from subroutines in that functions *must* return a single result string to the caller through the `return` instruction with which they end. Rexx replaces the function code in any statement with the returned value from the function. Subroutines may or may not send back a value to their caller via their `return` instruction. The returned value from a subroutine, if there is one, is placed into the special variable named `result`.

External routines can be functions, too. Often, these come in the form of a package designed to support a particular functionality and are called *extensions* or *function libraries.* External routines might also be the equivalent of internal subroutines, written in Rexx, except that they reside in a different file than that of the caller.

Rexx makes it easy to invoke external programs from your script, regardless of the language in which they are written. If the Rexx interpreter encounters a string in a script that does not correspond to its instruction set, it evaluates that expression and then passes it to the operating system for execution. So, it is simple to run operating system commands or other programs from a Rexx script. Chapter 14 illustrates how to do this. One of Rexx's great strengths is its role in issuing, controlling, and coordinating operating system commands. It is also easy to direct commands to other outside "environments" such as

text editors or other tools. Rexx is called a *macro language* because it is often used to provide programmability for various tools. For example, on mainframes Rexx is used as the macro language to program the widely used editors, XEDIT and the ISPF Editor.

There are a large variety of Rexx extensions and packages. For example, the open-source *Rexx/SQL* package provides an interface to a variety of relational databases from within Rexx scripts. Other examples include interfaces to *curses*, the text-screen control package; to *RexxXML*, for XML programming; to *ISAM*, the indexed sequential access method; to *TK* and *DW*, for easy GUI programming; to *gd*, for graphics images; *RxSock*, for TCP/IP sockets, and many other interfaces. Chapters 15 through 18 discuss and demonstrate some of these free and open-source packages. Chapter 29 discusses a few of the many interfaces to mainframe Rexx and how Rexx offers a high-level macro and interface language for mainframe interfaces and facilities. Appendix H lists several dozen of the many free and open-source interfaces that are available and tells how to locate them for downloading.

Internal Functions and Subroutines

Functions must always return exactly one result to the caller. Use the `return` instruction to do this. *Subroutines* may or may not send a result back to the caller via `return`, but they, too, end with the `return` instruction.

Functions may be invoked in either of two ways. One method codes the function name, immediately followed by arguments, wherever one might encode an expression:

```
returned_string  =  function_name(parameter_1, parameter_2)
```

The function is resolved and the string it returns is plunked right into the expression where it was coded. In this case, the assignment statement then moves that value to the variable `returned_string`. Since you can code a function anywhere you can code an expression, nesting the function within an `if` or `do` instruction is common:

```
if ( balanced_parentheses(string_in) ) = 'YES' then
```

Here the call to the function `balanced_parentheses` is nested within an `if` instruction to provide a result for the comparison. After the function `balanced_parentheses` has been run, its result is plunked right where it was encoded in the `if` instruction.

You can nest functions within functions, as shown in this `return` instruction from one of the sample scripts we discuss later in this chapter:

```
return substr(string,length(string),1) || ,
reverse(substr(string,1,length(string)-1))
```

Recall that the comma is the *line continuation character*. So, both of these lines constitute a single statement.

This `return` instruction features a complex expression that returns a single character string result to the caller. The first part of the expression nests the `length` function within the `substr` function; the second part nests `length` within `substr` within `reverse`. Yikes! Nesting is very powerful, but for the sake of clarity we don't recommend getting too fancy with it. Deeply nested expressions may show cleverness

but they become unintelligible if too complex. When complex code is developed for corporate, governmental, or educational institutions, the value of that code drops the moment the programmer who wrote it leaves the organization.

The second basic way to invoke a function is through the `call` instruction:

```
call  function_name   parameter_1, parameter_2
```

For example, to duplicate the code we looked at earlier where the invocation of the `balanced_parentheses` routine was nested within an `if` statement, we could have alternatively coded:

```
call  balanced_parentheses  string_in
if result = 'YES' then   /* inspect the result returned from the function call */
```

The result string from the function is automatically placed into the special variable named `result` and may be accessed from there.

Special variable `result` will be set to uninitialized if not set by a subroutine. In this case its value will be its own name in capitals: `RESULT`.

Subroutines may only be invoked by the `call` instruction. Encode this in the exact same manner as the second method for invoking functions:

```
call subroutine_name  parameter_1,  parameter_2
```

The special variable `result` contains a value *if* the subroutine passed back a value on its `return` instruction. Otherwise `result` will be set to uninitialized (the value `RESULT`). All uninitialized variables are their own names set to uppercase, so use this test to see if `result` was not set:

```
if result = 'RESULT'  then say 'RESULT was not set by the subroutine.'
```

The built-in function `symbol` can also be used to see if any variable is uninitialized or whether it has been assigned a value. It returns the character string `VAR` if a variable has a value or the string `LIT` otherwise. We can apply it to see if `result` was assigned a value:

```
if symbol('RESULT')  == 'VAR'  then say 'A result was returned'
if symbol('RESULT')  == 'LIT'  then say 'No result was returned'
```

To summarize, here's a code snippet that shows how to organize a main routine (or *driver*) and its subroutine. The code shows that the `call` to the internal subroutine did not set special variable `result`:

```
/* Show whether RESULT was set by the CALL */

call  subroutine_name

if result = 'RESULT'
   then  say 'No RESULT was returned'
   else  say 'A RESULT was returned'

if symbol('RESULT')  == 'VAR'
   then say 'A RESULT was returned'
```

```
if symbol('RESULT')  == 'LIT'
   then say 'No RESULT was returned'

exit 0

subroutine_name:
   return
```

The `return` instruction ends the subroutine, but does not include an operand or string to send back to the calling routine. The code snippet displays these messages when it returns from the subroutine:

```
No RESULT was returned
No RESULT was returned
```

Now change the last statement in the code, the `return` instruction in the subroutine, to something like this:

```
return 'result_string'
```

Or, change it to this:

```
return 0
```

Either encoding means that the special variable `result` is set to the string returned. After invoking the internal routine, the code snippet now displays:

```
A RESULT was returned
A RESULT was returned
```

When encoding subroutine(s) and/or functions after the main routine or driver, code an `exit` instruction at the end of the code for the main routine. This prevents the flow of control from rolling right off the end of the main routine and going into the subroutines.

Here is another example that is the exact same as that seen in the preceding example. However, we have coded it incorrectly by commenting out the `exit` instruction that follows the main routine. We have also added a statement inside the subroutine that displays the message: `Subroutine has been entered`.

Here's the code:

```
/* Show whether RESULT was set by the CALL */

call  subroutine_name

if result = 'RESULT'
   then  say 'No RESULT was returned'
   else  say 'A RESULT was returned'

if symbol('RESULT')  == 'VAR'
   then say 'A RESULT was returned'
if symbol('RESULT')  == 'LIT'
   then say 'No RESULT was returned'
```

```
/* exit 0 */                                /* now commented out */

subroutine_name:
   say 'Subroutine has been entered'        /* new line of code  */
   return 0
```

This script displays this output:

```
Subroutine has been entered
A RESULT was returned
A RESULT was returned
Subroutine has been entered     <=  this line results from no EXIT instruction!
```

This shows you must code an `exit` instruction at the end of the main routine if it is followed by one or more subroutines or functions. The last line in the sample output shows that the subroutine was entered incorrectly because an `exit` instruction was not coded at the end of the main routine. As with the subroutine's `return` instruction, it is optional whether or not to code a return string on the `exit` statement. In the preceding example, the `exit` instruction passed a return code of `0` to the environment.

What if we place the code of subroutines *prior* to that of the main routine? Here we located the code of the subroutine prior to the driver:

```
/* Shows why subroutines should FOLLOW the main routine */

subroutine_name:
   say 'Subroutine has been entered'
   return 0

call  subroutine_name

if result = 'RESULT'
   then  say 'No RESULT was returned'
   else  say 'A RESULT was returned'

if symbol('RESULT')  == 'VAR'
   then say 'A RESULT was returned'
if symbol('RESULT')  == 'LIT'
   then say 'No RESULT was returned'
exit 0
```

Running this script displays just one line:

```
Subroutine has been entered
```

What happened was that Rexx starts at the top of the file and proceeds to interpret and execute the code, line by line. Since the subroutine is first in the file, it executes first. Its instruction `return  0` caused exit from the program before we ever got to the main routine! Oops. Always place the code for any internal subroutines or functions *after* the main routine or driver.

We'll cover program structure in more detail later. For now, here are some basic rules of thumb:

- End each subroutine or function with the `return` instruction.
- Every function *must* have an operand on its `return` instruction.
- Subroutines may optionally have a result on their `return` instruction.
- Encode the `exit` instruction at the end of the code of the main routine or driver.
- Place subroutines and functions after the main routine or driver.

We saw that Rexx uninitializes special variable `result` when a called subroutine does not pass back a result string. If you ever need to uninitialized a Rexx variable yourself, code the `drop` instruction:

```
drop  my_variable
```

This sets a variable you may have used back to its uninitialized state. It is now equal to its own name in all uppercase.

You can drop multiple variables in one instruction:

```
drop   my_variable_1   my_variable_2   my_variable_3
```

Passing Parameters into a Script from the Command Line

Passing data into a script is important because this provides programs with flexibility. For example, a script that processes a file can retrieve the name of the file to process from the user. You can pass data elements into scripts by coding them on the same command line by which you run the script. Let's explore how this is accomplished.

Data passed into a script when it is invoked are called *command-line arguments* or *input parameters*. To invoke a Rexx script and pass it command-line arguments or parameters, enter something like this:

```
c:\Regina\pgms> script_name  parameter_1  2  parameter_3
```

The script reads these three input strings `parameter_1`, `2`, and `parameter_3` with the `arg` instruction. `arg` automatically translates the input parms to uppercase. It is the equivalent of the instruction `parse upper arg`. If no uppercase translation is desired, use `parse arg.` Remember that a period following either of these instructions discards any more variables than are encoded on the `arg` or `parse arg` instruction. This example discards any arguments beyond the third one, if any are entered:

```
arg input_1 input_2 input_3  .    /* read 3 arguments, translate to capitals   */
```

Here is the same example coded with the parse arg instruction:

```
parse arg input_1 input_2 input_3 . /* read 3 arguments, no upper translation  */
```

By default, the `arg` and `parse arg` instructions splice the input parameters into pieces based on their separation by one or more intervening spaces. If you ran the program like this:

```
c:\Regina\pgms> script_name  parameter_1  2  parameter   _3
```

You'd want to code this statement in the script to pick up the input arguments:

```
parse arg  input_1  input_2  input_3  input_4  .
```

The resulting variable values would be:

```
input_1  =   parameter_1
input_2  =   2
input_3  =   parameter
input_4  =   _3
```

As per the basic rules of parsing, encoding too many input parameters puts all the overflow either into the *placeholder variable* (the period) or into the last specified input variable on the `parse arg` instruction.

Entering too few input parameters to match the `parse arg` statement means that the extra variables on the `parse arg` will be set to uninitialized. As always, an uninitialized variable is equal to its own name in uppercase.

Passing Parameters into Subroutines and Functions

Say that our sample script needs to run a subroutine or function, passing it the same three input parameters. Code the subroutine or function call as:

```
call  sub_routine  input_1, input_2, input_3
```

Code a comma between each of the parameters in the `call` instruction. The string (if any) sent back from the `call` will be available in the special variable named `result`.

Code a function `call` just like the `call` to the previous subroutine. Or encode it wherever you would an expression, as illustrated earlier, in the form:

```
result_string =  function_name(input_1, input_2, input_3)
```

Inside the function or subroutine, use either `arg` or `parse arg` to retrieve the arguments. The function or subroutine picking up the input parameters should encode commas that parallel those of the `call` in its `arg` or `parse arg` instruction:

```
arg input_1,  input_2,  input_3  .
```

or

```
parse arg  input_1,  input_2,  input_3  .
```

The period or placeholder variable is optional. Presumably, the subroutine or function knows how many input parameters to expect and does not need it.

These examples illustrate the `arg` *instruction* retrieving the argument string passed to a script and splicing it apart into its individual pieces. There is also an `arg` *built-in function*. The `arg` function returns information about input arguments to the routine. For scripts called as functions or subroutines, the `arg` function either:

- Tells how many argument strings were passed in
- Tells whether a specific-numbered argument was supplied
- Supplies a specified argument

Let's look at a few examples. To learn how many arguments were passed in, code:

```
number_of_arguments  =  arg()
```

To retrieve a specific argument, say the third one, code:

```
get_third_argument  =  arg(3)
```

To see if the third argument exists (was passed or encoded in the call), write:

```
if (arg(3) == '')  then say 'No third argument was passed'
```

or

```
if arg(3,'O')  then say 'No third argument was passed'
```

The first of the two sample lines show that an input argument read by an internal routine will be the null string if it is not supplied to the routine. This differs from a command-line input argument that is read but not supplied, which is set to uninitialized (its own name in uppercase).

The second sample line shows one of the two options that can be used with the `arg` function:

- `E` (Exists) — Returns 1 if the *n*th argument exists. Otherwise returns 0.
- `O` (Omitted) — Returns 1 if the *n*th argument was Omitted. Otherwise returns 0.

The `arg` function *only* supplies this information for scripts that are called as functions or subroutines. For scripts invoked from the operating system's command line, the `arg` function will always show only `0` or `1` argument strings. In this respect Rexx scripts invoked as commands from the operating system behave differently than scripts invoked as internal routines (functions or subroutines). This is one of the very few Rexx inconsistencies you'll have to remember: the `arg` function tells how many arguments are passed into an internal routine, but applied to the command-line arguments coming into a script, it always returns either `0` or `1`.

Chapter 8

A Sample Program

To see how parameters are passed into programs, and how code can be modularized, let's look at a couple sample programs. The first sample program consists of a brief script that reads information from the command line. This main routine or "driver" then turns around and calls a subroutine that performs the real work of the program. Then the driver displays the result from the subroutine on the user's screen.

Of course, the driver could actually be part of a larger application. For example, it might be a "service routine" shared among programs in the application. Whatever its use, the important principles to grasp are how code can be modularized and how information can be passed between modules.

The first sample program tells whether parentheses in a string are *balanced*. A string is said to be balanced if:

- Every left parenthesis has a corresponding closing right parenthesis
- No right parenthesis occurs in the string prior to a corresponding left parenthesis

Here are some examples. These input strings meet the two criteria and so are considered balanced:

```
(())
() () ()
return (qiu(slk) ())
((((((()))))))
if (substr(length(string,1,2)))
```

These are unbalanced strings. Either the numbers of left and right parentheses are unequal, or a right parenthesis occurs prior to its corresponding left parenthesis:

```
)alkjdsfkl(                    /* right paren occurs before its left paren */
((akljlkfd)                    /* 2 left parens, only 1 right paren        */
if (substr(length(string,1,2))  /* 3 left parens, only 2 right parens       */
```

The last example shows that a script like this could be useful as a syntax-checker, or as a module in a language interpreter. You can actually use it to verify that your scripts possess properly encoded, balanced sets of parentheses.

To run the program, enter the string to verify as a command-line argument. Results appear on the next line:

```
C:\Regina\pgms> call_bal.rexx   if(substr(length(string,1,2))
Parentheses are NOT balanced
```

Try again, this time adding one last right parenthesis to the input string:

```
C:\Regina\pgms> call_bal.rexx   if(substr(length(string,1,2)))
Parentheses are balanced!
```

Here's the code for the caller. All it does is read the user's command-line input parameter and pass that character string to a function named `balanced_parens` that does the work. The function `balanced_parens` may be either internal or external — no change is required to its coding regardless of

where you place it. (However, you must be sure the operating system knows where to locate external functions. This often requires setting an environmental variable or the operating system's search path for called routines. We'll discuss this in detail later.)

```
/*  CALL BAL:                                                       */
/*                                                                  */
/*     Determines if the parentheses in a string are balanced.      */

arg  string .        /* the string to inspect                      */

if  balanced_parens(string) = 'Y' then   /* get answer from function */
    say 'Parentheses are balanced!'      /* write GOOD message ..or..*/
else
    say 'Parentheses are NOT balanced'   /* write INVALID message    */

exit 0
```

Here's the internal or external function that figures out if the parentheses are balanced. The algorithm keeps track of the parentheses simply by adding 1 to a counter for any left parenthesis it encounters, and subtracting 1 from that counter for any right parenthesis it reads. A final counter (`ctr`) equal to 0 means the parentheses are balanced — that there are an equal number of left and right parentheses in the input string. If at any time the counter goes negative, this indicates that a right parenthesis was found prior to any possible matching left parenthesis. This represents another case in which the input string is invalid.

```
/*  BALANCED PARENS:                                                */
/*                                                                  */
/*     Returns Y if parentheses in input string are balanced,       */
/*     N if they are not balanced.                                  */

balanced_parens:

arg  string .        /* the string to inspect                      */

ctr = 0                 /* identifies right paren BEFORE a left one  */
valid = 1
endstring = length(string)             /* get length of input string */

do j=1  to endstring  while (valid)
   char = substr(string,j,1)           /* inspect each character     */
   if char = '(' then ctr = ctr + 1
   if char = ')' then ctr = ctr - 1
   if ctr < 0   then valid = 0
end

if ctr = 0 then return 'Y'
           else return 'N'
```

Another way to code this problem is for the subroutine to return 1 for a string with balanced parentheses, and 0 if they are unbalanced. Then you could code this in the caller:

```
if balanced_parens(string)  then
   say  'Parentheses are balanced!'
else
   say  'Parentheses are NOT balanced'
```

This allows coding the function as an *operatorless condition test* in a manner popular in programming in languages like C, C++, or C#. But remember that the expression in an `if` instruction must evaluate to 1 (TRUE) or 0 (FALSE) in Rexx, so the function *must* return one of these two values. A nonzero, positive integer other than 1 will not work in Rexx, unlike languages in the C family. A positive value other than 1 results in a syntax error in Rexx (we note, though, that there are a few Rexx interpreters that are extended to allow safe coding of operatorless condition tests).

Coding operatorless condition tests also runs counter to the general principle that a function or subroutine returns 0 for success and 1 for failure. Wouldn't balanced parentheses be considered "success"? This coding works fine but contravenes the informal coding convention.

The Function Search Order

Given that Rexx supports built-in functions, internal functions, and external functions, an important issue is how Rexx locates functions referred to by scripts. For example, if you write an internal function with the same name as a built-in function, it is vital to understand which of the two functions Rexx invokes when some other routine refers to that function name.

This issue is common to many programming languages and is called the *function search order*. In Rexx the function search order is:

1. *Internal function* — The label exists in the current script file.
2. *Built-in function* — Rexx sees if the function is one of its own built-in functions.
3. *External function* — Rexx seeks an external function with the name. It may be written in Rexx or any language conforming to the system-dependent interface that Rexx uses to invoke it and pass the parameter(s).

Where Rexx looks for external functions is operating-system-dependent. You can normally place external functions in the same directory as the caller and Rexx will find them. On many platforms, you must set an *environmental variable* or a *search path* parameter to tell the operating system where to look for external functions and subroutines.

The function search order means that you could code an internal function with the same name as a Rexx built-in function and Rexx will use *your* function. You can thus replace, or override, Rexx's built-in functions.

If you want to avoid this, code the function reference as an uppercase string in quotation marks. The quotation marks mean Rexx skips Step 1 and *only* looks for built-in or external functions. Uppercase is important because built-in functions have uppercase names.

With this knowledge, you can override Rexx functions with your own, while still invoking the built-in functions when you like. You can manage Rexx's search order to get the best of both worlds.

Recursion

A *recursive* function or routine is one that calls itself. Any recursive function could be coded in traditional nonrecursive fashion (or *iteratively*), but sometimes recursion offers a better problem solution. Not all programming languages support recursion; Rexx does.

Since a recursive function invokes itself, there must be some end test by which the routine knows to stop *recursing* (invoking itself). If there is no such end test, the program recurses forever, and you have effectively coded an "endless loop!"

Figure 8-2 pictorially represents recursion.

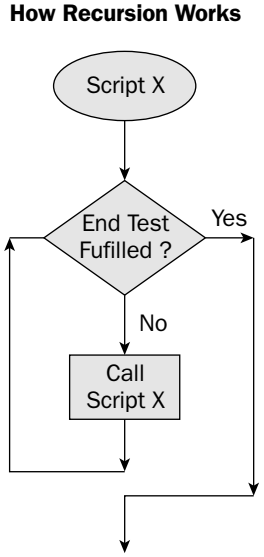

Figure 8-2

This sample recursive function reverses the characters within a given string—just like Rexx's `reverse` built-in function. If you feed it the character string `abc`, it returns the string `cba`.

The function calls itself to process each character in the input string and finds its "end test" when there are no more characters left in the string to process. Each time the function is entered, it returns the last character in the string and recurses to process the remaining string.

```
/*  REVERSE:                                                        */
/*                                                                  */
/*     Recursive routine that reverses the characters in a string.  */

reverse: procedure

parse arg string        /* read the string to reverse              */

if string == ''         /* here's the 'end recursion' condition    */
```

```
      then return ''
   else
      return substr(string,length(string),1) || ,
             reverse(substr(string,1,length(string)-1))
```

The `reverse` function uses the *strictly equal* operator (`==`). This is required because the regular "equals" operator pads item with blanks for comparisons, something that might not work in this function. The line that uses the strictly equal operator compares the input string to the *null string*, the string that contains no characters, represented by two back-to-back quotation marks (`''`). This is the "end test" that tells the function to return, because it has processed all the characters in the original input string:

```
if string == ''         /* here's the 'end recursion' condition      */
   then return ''
```

The last two lines of the function show how to continue a statement across lines. Just code a comma (`,`) and the `return` instruction's expression spans into the next line. The comma is Rexx's *line continuation character*. Code it at any natural breakpoint in the statement. Between parts of a statement is fine; within the middle of a character string literal would not work. This is valid:

```
say  'Hi '   ,
     'there!'              /*  valid line continuation */
```

But this will fail with a syntax error, because the line continuation character appears in the middle of a quoted literal:

```
say  'Hi     ,
     there!'               /* invalid line continuation, syntax error! */
```

Of course, the trick to this program to reverse character strings is this one, heavily nested line of code:

```
      return substr(string,length(string),1) || ,
             reverse(substr(string,1,length(string)-1))
```

The first portion of this statement always returns the last character in the substring being inspected:

```
substr(string,length(string),1)
```

An alternative way to code this is to use the `right` function, as in: `right(string, 1)`.

The second portion of the `return` statement recursively invokes the `reverse` function with the remaining substring to process. This is the original string passed in, minus the last character (which was just returned to the caller):

```
reverse(substr(string,1,length(string)-1))
```

To test a program like this, you need a simple *driver* or some "scaffolding" to initially invoke the new `reverse` function. Fortunately, the rapid development that Rexx enables makes this easy. Coding a driver to test the new `reverse` function is as simple as coding these few lines:

```
/*   Simple "test driver" for the REVERSE function.                */

parse arg string .
call reverse string                  /* call the REVERSE function   */
say 'The reversed string is:' result  /* display the RESULT          */
exit 0
```

This code reads an input string from the user as an input command-line argument. It invokes the recursive, user-written `reverse` function and displays the result to the user.

The `say` instruction in this code uses the special variable `result` to display the string returned from the `reverse` function on the user's display screen:

```
say 'The reversed string is:' result  /* display the RESULT          */
```

Our new `reverse` function has the same name and functionality as Rexx's own, built-in `reverse` function. Which will Rexx run? The *function search order* tells us. Assuming that the `reverse` function we coded is internal, Rexx invokes it, because user-written internal functions have priority over Rexx's built-in functions in the function search order. If we want to use the built-in Rexx reverse function instead, we would code the name of the function in quoted uppercase letters. These two lines show the difference. This line invokes our own `reverse` function:

```
call reverse   string     /* call our own REVERSE function           */
```

In contrast, this statement runs Rexx's built-in `reverse` function:

```
call 'REVERSE' string     /* use the Rexx built-in REVERSE function  */
```

More on Scoping

Developers place internal functions and subroutines after the main routine or driver in the script file. Here's the basic prototype for script structure where the main script has subroutines and/or functions:

```
main_routine:
   call  my_function     parameter_in
   call  my_subroutine   parameter_in
   exit 0

my_function: procedure
   return  result_string

my_subroutine: procedure
   return
```

Rexx does not require any label for the main routine or driving portion of the script, but we recommend it as a good programming practice. A Rexx *label* is simply a name terminated with a colon. In this script, we've identified the driver routine with the label `main_routine:` . This is good programming practice in very large programs because it may not always be obvious where the logic of the driver really starts. In other words, if there is a long list of variable declarations or lots of initialization at the top of a script, identifying where the "real" work of the main routine begins can sometimes be helpful.

A key issue in any large program is *scoping* — which of the caller's variables are available for reading and/or updating by a called function or subroutine. In Rexx, the `procedure` instruction is the basic tool for managing variable scoping. `procedure` is encoded as the first instruction following the label in any function or subroutine for which it's used.

The `procedure` instruction protects all existing variables by making them unknown to any instructions that follow. It ensures that the subroutine or function for which it is encoded cannot access or change any of its caller's variables. For example, in the `reverse` function, we coded this first line:

```
reverse: procedure
```

This means the `reverse` routine cannot read or update any variables from its caller — they are protected by the `procedure` instruction. This is a good start on proper modularity, but of course, we need a way to give the `reverse` routine access to those variables it *does* need to access. One approach is to pass them in as arguments or parameters, as we did in calling the `reverse` function, with this general structure:

```
calling routine:
   parse arg  parm_1  parm_2  .  /* get command-line arguments from the user    */
   call function_name parm_1, parm_2      /* pass them to the internal routine  */
   say 'The function result is:'  result  /* retrieve RESULT from the routine   */
   exit 0

function_name: procedure
   parse arg  parm_1,  parm_2             /* get parameters from the caller     */
   return  result_string                  /* return result to caller            */
```

The `procedure` instruction protects all variables from the function or subroutine. This function cannot even read any of the caller's variables. It knows only about those passed in as input parameters, `parm_1` and `parm_2`. It can read the variables that are passed in via `arg`, and it sends back *one* result string via the `return` instruction. *It cannot change the value of any of the* `arg` *variables in the caller*. These are passed in on a read-only basis to the function or subroutine, which can only pass back one string value by a `return` instruction.

Another approach to passing data items between routines is to specify *exposed variables* on the `procedure` instruction. These variables are available for both reading *and updating* by the invoked routine:

```
function_name: procedure  expose  variable_1  array_element.1
```

In this case the function or subroutine can read and manipulate the variable `variable_1` and the specific array element `array_element.1`. The function or subroutine has full read and update access to these two `expose`'d variables.

With this knowledge, here's an alternative way to structure the relationship between caller and called routine:

```
calling_routine:
   parse arg  parm_1  parm_2  .  /* get command-line arguments from the user    */
   call subroutine_name             /*  call  the subroutine (or function)      */
   say 'The function result is:'  result   /* retrieve RESULT from the routine  */
   say 'The changed variables are:' parm_1 parm_2  /* see if variables changed  */
```

```
   exit 0

subroutine_name: procedure  expose  parm_1  parm_2
   /* refer to and update the variables parm_1  and parm_2 as desired          */
   parm_1 = 'New value set by Sub. '
   parm_2 = '2nd new value set by Sub.'
   return  result_string                          /* return result to caller   */
```

The output from this code demonstrates that the subroutine changed the values the caller originally set for variables `parm_1` and `parm_2`:

```
The function result is: RESULT_STRING
The changed variables are: New value set by Sub. 2nd new value set by Sub.
```

The `procedure` instruction limits variable access in the called function or subroutine. Only those variables specifically named on the `procedure expose` instruction will be available to the called routine.

To summarize, there are two basic approaches to making caller variables available to the called routine. Either pass them in as input arguments, or code the `procedure expose` instruction followed by a variable list. The called function or subroutine cannot change input arguments — these are read-only values passed by the caller. In contrast, any variables listed on the `procedure expose` statement can be both read and updated by the called function or subroutine. The calling routine will, of course, "see" those updated variable values.

Two brief scripts illustrate these principles. This first demonstrates that the called routine is unable to change any variables owned by its caller because of the `procedure` instruction coded on the first line of the called routine:

```
/* This code shows that a PROCEDURE instruction (without an EXPOSE          */
/* keyword) prevents a called function or subroutine from reading           */
/* or updating any of the caller's variables.                               */
/*                                                                          */
/* Argument-passing and the ARG instruction gives the called                */
/* function or subroutine READ-ONLY access to parameters.                   */

calling_routine:

   variable_1 = 'main'
   variable_2 = 'main'

   call my_subrtn(variable_1)

   say 'main:' variable_1 variable_2            /* NOT changed by my_subrtn */
   exit 0

my_subrtn: procedure

   arg variable_1                              /* provides read-only access */

   say 'my_subrtn:' variable_1  variable_2     /* variable_2 is not set     */

   variable_1 = 'my_subrtn'
```

```
    variable_2 = 'my_subrtn'

    say 'my_subrtn:' variable_1  variable_2
    return
```

This is the output from this script:

```
my_subrtn: MAIN  VARIABLE_2
my_subrtn: my_subrtn  my_subrtn
main: main  main
```

The first output line shows that the subroutine was passed a value for `variable_1`, but `variable_2` was not passed in to it. The subroutine accessed the single value passed in to it by its `arg` instruction. The second line of the output shows that the called routine locally changed the values of variables `variable_1` and `variable_2` to the string value `my_subrtn`—but the last line shows that these assignments did not affect the variables of the same names in the caller. The subroutine could not change the caller's values for these two variables. This is so because the `procedure` instruction was encoded on the subroutine but it did not list any variables as `expose`'d.

This next script is similar but illustrates coding the `procedure expose` instruction to allow a called routine to manipulate the enumerated variables of its caller:

```
/*  This code shows that ONLY those variables listed after EXPOSE         */
/*  may be read and updated by the called function or subroutine.         */

calling_routine:

   variable_1      = 'main'
   array_name.  = 'main'              /* The called routine can update    */
   array_element.1 = 'main'              /* array elements if desired.    */
   not_exposed     = 'main'

   call my_subrtn                        /* don't pass parms, use EXPOSE  */

   say  'main:'  variable_1  array_name.4  array_element.1  not_exposed
   exit 0

my_subrtn: procedure expose  variable_1  array_name. array_element.1

   say 'my_subrtn:' variable_ 1 array_name.4  array_element.1  not_exposed

   variable_1   = 'my_subrtn'            /* These will be set back in the */
   array_name.4 = 'my_subrtn'            /* caller, since they were       */
   array_element.1 = 'my_subrtn'         /* on the PROCEDURE EXPOSE.      */

      say 'my_subrtn:' variable_1  array_name.4  array_element.1 not_exposed
   return
```

The output from this script is:

```
my_subrtn: main main main  NOT_EXPOSED
my_subrtn: my_subrtn  my_subrtn  my_subrtn  NOT_EXPOSED
main: my_subrtn  my_subrtn  my_subrtn  main
```

The first output line shows that the subroutine accessed the three caller's variables listed on the `procedure expose` instruction. This shows the three variables set to the string value `main`. The fourth variable shows up as `NOT_EXPOSED` because the subroutine did not list it in its `procedure expose` statement and cannot access it.

The second output line shows that the subroutine set the value of the three variables it can change to the value `my_subrtn`. This line was displayed from within the subroutine.

The last output line confirms that the three variables set by the subroutine were successfully passed back to and picked up by the caller. Since only three variables were passed to the subroutine, the fourth variable, originally set to the string value `main` by the caller, still retains that same value.

What about external routines? Invoke them just like internal routines, but the Rexx interpreter always assigns them an implicit `procedure` instruction so that all the caller's variables are hidden. You cannot code a `procedure expose` instruction at the start of the external routine. Pass information into the external routine through input arguments. Code a `return` instruction to return a string from the external routine. Or, you can code an `exit` instruction with a return value.

For internal routines, if you code them without the `procedure` instruction, *all* the caller's variables are available to the internal routines. All the caller's variables are effectively *global variables*. Global variables are values that can be changed from any internal routine. Global variables present an alternative to passing updatable values into subroutines and functions via the `procedure expose` instruction.

Developers sometimes like using global variables because coding can be faster and more convenient. One does not have to take the time to consider and encode the correct `procedure expose` instructions. But global variables are not considered a good programming practice because they violate one of the key principles of modularity — that variables are explicitly assigned for use in specific modules. So that you recognize this scenario when you have to maintain *someone else's* code, here is the general script structure for using global variables:

```
/*  Illustrate that Global Variables are accessible to ALL internal routines */
main_routine:
   a = 'this is a global variable!'
   call  my_subroutine
   say  'Prove subroutine changed the value:' a
   feedback = my_function()
   say 'Prove the function changed the value:' a
   exit 0

my_subroutine:
   /* all variables from MAIN_ROUTINE are available to this routine for
      read and or update */
   a = 'this setting will be seen by the caller'
   return

my_function:
   /* all variables from MAIN_ROUTINE are available to this routine for read
      and or update */
   a = 'this new value will be seen by the caller'
   return 0
```

The program output shows that the two internal routines are able to change any variable values in the calling routine at will. The two output lines are displayed by the driver. The latter portion of each line shows that the subroutine and function were able to change the value of the global variable named `a`:

```
Prove subroutine changed the value: this setting will be seen by the caller
Prove the function changed the value: this new value will be seen by the caller
```

All you have to do to use global variables is neglect to code the `procedure` instruction on subroutines on functions. This is convenient for the developer. But in large programs, it can be extremely difficult to track all the places in which variables are altered. *Side effects* are a real possibility, unexpected problems resulting from maintenance to code that does not follow the principles of structured programming and modularity.

To this point, we've discussed several ways to pass variables into and back from functions and subroutines. This chart summarizes the ways to pass information to and from called internal subroutines and functions:

Technique	Internal Routine's Variable Access	Comments
Pass arguments as input parameters	Read-only access to the passed variables *only*	Standard for passing in read-only values
`procedure expose`	Read and update access to `expose`'d variables *only*	Standard for updating some variables while hiding others
`procedure` (*without* `expose`)	Hides *all* the caller's variables	Standard for hiding all caller's variables
Global variables	Read and update access to *all* the caller's variables	Violates principles of modularity; works fine but not recommended
`return` expression	Send back one string to the caller	Standard for passing back one item of information

Whichever approach(es) you use, consistency is a virtue. This is especially the case for larger or more complex programming applications.

Another Sample Program

This next sample script illustrates a couple of the different ways to pass information into subroutines. One data element is passed in as an input argument to the routine, while the other data item is passed in via the `procedure expose` instruction.

This program searches a string and returns the rightmost occurrence of a specified character. It is a recursive function that duplicates functionality found in the built-in `lastpos` function. It shows how to pass data items to a called internal routine as input parameters and how to use the `procedure expose` instruction to pass in updateable items.

```
/*  RINDEX:                                                          */
/*                                                                   */
/*     Returns the rightmost position of a byte within a string.     */

rindex: procedure expose search_byte

parse arg string                                  /* read the string */

say string search_byte               /* show recursive trace for fun */

string_length   = length(string)     /* determine string length      */
string_length_1 = length(string) -1  /* determined string length - 1 */

if string == ''         /* here's the 'end recursion' condition      */
   then return 0
else do
   if substr(string,string_length,1) == search_byte then
      return string_length
   else
      new_string_to_search = substr(string,1,string_length_1)
   return rindex(new_string_to_search)
end
```

This script requires two inputs: a character string to inspect for the rightmost occurrence of a character, and the character or "search byte" to look for.

When invoked, the function looks to see if the last character in the string to search is the search character. If yes, it returns that position:

```
   if substr(string,string_length,1) == search_byte then
      return string_length
```

If the search character is not found, the routine calls itself with the remaining characters to search as the new string to search:

```
      new_string_to_search = substr(string,1,string_length_1)
      return rindex(new_string_to_search)
```

The end condition for recursion occurs when either the character has been found, or there are no more characters in the original string to search.

The function requires two pieces of input information: the string to inspect, and the character to find within that string. It reads the string to inspect as an input parameter, from the `parse arg` instruction:

```
parse arg string                                    /* read the string */
```

The first line in the function gives the program access to the character to locate in the string:

```
rindex: procedure expose search_byte
```

The two pieces of information are coming into this program in two different ways. In a way this makes sense, because the character to locate never changes (it is a global constant), but the string that the function searches is reduced by one character in each recursive invocation of this function. While this program works fine, it suggests that passing in information through different mechanisms could be confusing. This is especially the case when a large number of variables are involved.

For large programs, consistency in parameter passing is beneficial. Large programs become complicated when programmers mix internal routines that have `procedure expose` instructions with routines that do not include this instruction. Rexx allows this but we do not recommend it. Consistency underlies readable, maintainable code. Coding a `procedure` or `procedure expose` instruction for *every* internal routine conforms to best programming practice.

Summary

This chapter describes the basic mechanisms by which Rexx scripts are modularized. Modularity is a fundamental means by which large programs are rendered readable, reliable, and maintainable. Modularity means breaking up large, complex tasks into a series of smaller, discrete modules. The interfaces between modules (the variables passed between them) should be well defined and controlled to reduce complexity and error.

We covered the various ways to pass information into internal routines and how to pass information from those routines back to the caller. These included passing data elements as input arguments, the `procedure` instruction and its `expose` keyword, and using global variables. We discussed some of the advantages and disadvantages of the methods, and offered sample scripts to illustrate each approach. The first sample script read a command-line argument from its environment and passed this string as an input argument to its subroutine. The subroutine passed a single value back up to its caller by using the `return` instruction. The last sample script was recursive. It invoked itself as a subroutine and illustrated how the `procedure expose` instruction could be used to pass values in recursive code. This latter example also suggests that consistently encoding the `procedure expose` instruction on *every* routine is a good approach for large programming projects. This consistent approach reduces errors, especially those that might otherwise result from maintenance on large programs that use global variables.

Test Your Understanding

1. Why is modularity important? How does Rexx support it?
2. What's the difference between a subroutine and function? When should you use one versus the other?
3. What is the difference between internal and external subroutines? How is the `procedure` instruction used differently for each?
4. What is the function search order, and how do you override it?
5. What are the basic ways in which information is passed to/from a caller and its internal routines?
6. What happens if you code a `procedure` instruction without an `expose` keyword? What's the difference between parameters passed in to an internal subroutine and read by the `arg` instruction versus those that are exposed by the `procedure expose` instruction?
7. In condition testing, TRUE is 1 and FALSE is 0. What happens when you write an if instruction with a condition that evaluates to some nonzero integer other than `1`?