Rexx Programming – Best Practices

by Howard Fosdick © updated 2022

Introduction--

Rexx is one of the most flexible, capable programming languages around. Its range of options and power means that some stylistic “rules of thumb” can be helpful. This list offers some.

The list includes “common errors” as well. Remember them to reduce your debugging time and write error-free code.

Optimal “rules of thumb” can vary somewhat according to whether you’re writing a short script or a large programming system. I note rules of thumb that are specially recommended practices for large systems. They are collected towards the bottom of the list.

Rexx Rules of Thumb--

Write Readable Code, Not “Fortune-Cookie” Code

Some programmers like to write clever, “fortune-cookie” code. Their code doesn’t live long because it can not be maintained. Strive to write the most readable, maintainable code, not the shortest or most clever code. This first rule is the basis for many that follow.

Capitalize on Capitalization

Take advantage of Rexx’s case-insensitive nature to make code more readable.

Use Good Variable Names

Since you can use long variable names, do so for readability. Include embedded underscores in names too --

Good	**->**	Social_Security_Number
Poor	->	sno

Use Spacing and Indentation

Take advantage of Rexx’s free-format nature. Indent according to the nesting level of **IF** and **DO** statements. Line up **ENDs** as appropriate.

Code whose horizontal alignment matches its logic is much easier to read and maintain.

Limit Expression Nesting

You can nest expressions to any depth, but readability is better served by breaking up nested expressions into separate lines of code. Ever had to maintain someone else’s highly-nested expressions? It’s *very* difficult.

Comment Code

Rexx's free-form comments allow you to comment your code wherever and however you like. Well-commented code is way easier to maintain than code without explanation.

Maintain Code Comments

When you maintain code, you must update the code comments also. Otherwise they will become misleading rather than an aid.

Handle Errors

Good code is robust and fail-safe. Handle all errors from—

- Command results
- Interpreter-raised error conditions
- Return codes from interfaces
- I/O results

Use Rexx's Built-in Error and Exception Handling to Manage Errors

While there are some cases where you will not want out-of-line error handling, you should normally employ Rexx's built-in exception handlers.

Check All Return Codes

Good programs check **all** return codes to ensure their robustness and viability.

Explicitly Initialize Subscripts

Explicitly initialize a subscript at the top of the **DO** loop at which it is used.

Do Not Alter Subscripts Inside Loops

Do not explicitly alter a subscript variable while within a loop. Let the **DO** instruction increment it for you naturally.

Use Top-Driven Loops Only

Use only top-driven **DO** loops. Bottom-driven loops do not conform to the rules of structured programming.

Don't Exit From Inside a Loop

Do not early force-exit from within a **DO** loop. Structured programming only allows for normal exit at the loop's top-driven condition check.

Use a Loop Subscript Only for a Loop

If you use a variable like **I** for subscripting a **DO** loop, do not also use it to index a table from within that loop!

If You Nest Loops, Use a Different Subscript for Each

If you don't follow this principle, your logic won't work properly. When nesting **DO** loops, you've got to use a different subscript for each **DO**.

Build OS Commands in Simple Steps

Build strings that represent operating system commands in several simple steps, not in a single huge, complicated statement.

Build OS Commands in a Variable

Build strings that represent operating system commands within a variable. You can then display the variable to ensure the command string is properly created.

Avoid Unnecessary Quotation Marks in Building OS Commands

One sometimes sees double-quoting when building operating system commands for mainframe environments. This is not necessary and hurts the readability of the code. Use as few quotes as possible when concatenating command strings.

Use Rexx-Aware Editors

Using a Rexx-aware editor can reduce your error rate. Example products include—

- THE (The Hessling Editor) for Windows, Linux, Unix, etc.
- The REXX Text Editor (RexxEd), distributed with Reginald for Windows
- The Interactive System Productivity Facility (ISPF) on mainframes
- XEDIT on mainframes

Publish Site Style Standards

This renders code at a site similar regardless of the author. Consider what your enforcement mechanism should be.

Use Automated Tools to Ensure Standards Compliance

Many sites buy or develop tools to ensure standards compliance and the similarity of code across authors. ***Templates*** that programmers fill out for documentation purposes are one good example.

Consider Code Reviews

Code reviews are another technique to ensure common code style and conformance to site standards.

Don't Forget to End a Comment

Every comment must be ended by ***/**. Otherwise your program becomes one long comment!

Don't Forget to End a Literal String or Parenthetical Expression

Same as ending a comment… you *must* do it!

Don't Forget RETURN or EXIT

Remember that functions *must* return a string to the caller; subroutines may optionally do so. Be sure to code **RETURNs** and **EXITs** where required.

Don’t Overlook Automatic Upper-case Conversion

Remember that instructions like **PULL** and **ARG** auto-convert to upper-case. If you do not want auto-conversion, use **PARSE PULL** and **PARSE ARG**.

This can be especially important if your program reads in file names. File names are case-sensitive in Linux and Unix, but not in Windows.

Left Parentheses Must Immediately Follow Functions

Don’t forget to immediately follow a function by a left parenthesis. No intervening space is allowed.

```
Incorrect   ->   rc =  function_name    (argument1)
Correct     ->   rc =  function_name(argument1)
```

CALLs Do Not Use Parentheses

Remember that **CALL** statements do not use parentheses to enclose arguments.

```
Incorrect   ->   call  subroutine(argument1, argument2)
Correct     ->   call  subroutine  argument1,  argument2
```

Don’t Forget that Functions Return Strings

Don’t forget that a function returns a string. You must provide a place for the string to go.

```
Incorrect   ->   function_name(argument1)
Correct     ->   feedback = function_name(argument1)
Correct     ->   call   function_name  argument1
```

Watch Statement Continuations

Remember that the *comma* is Rexx’s line continuation character. Surround comma’s used for statement continuation with blanks for readability. Line them up if multiple ones occur across several lines.

Use Strict Comparisons When Required

If you want to compare a string including its preceding or following blanks, be sure to use Rexx’s strict comparison operators.

Don’t Strict-Compare Numbers

Strict comparisons are for strings, not for numeric values.

Don’t Use DO UNTIL

Structured programming doesn’t permit bottom-driven loops. Any such loop can be structured properly by using the **DO-WHILE** instead.

Don’t Use SIGNAL as a GOTO

Structured programming does not allow GO TO’s. Use structured control constructs instead (**DO**, **IF**, etc).   Set a control flag if necessary to convert your GO TO’s to **IF’s** and **DO’s**.

How to Return More than One Value from a Function

Rexx functions return a single value only. If you want to return more than one value you can—

- Concatenate the values into one returnable string, then use PARSE in the caller to get the individual values
- Write values to a file, which you use as a global passing mechanism

How to Iterate Over all Tails in a Stem Variable

This is not part of ANSI-standard classic Rexx.   But many Rexx interpreters have this built-in as an extra (for example, Open Object Rexx, roo!, Reginald, and BRexx). Or you can use any of several add-on packages that support this (for example, REXXUTIL).

Leverage Rexx’s Interactive Trace Facility

The big advantage to interpreters is interactive debugging. Leverage Rexx’s built-in debugging facilities to greatly reduce your debugging time.

Maintain a Site-wide Function or Object Library

Code re-use can *only* happen if you put the proper mechanisms in place to facilitate and encourage it.

Consistency is the Mother of All Virtues -- especially for large programs!

Whatever coding style you pick, apply it consistently.   Consistency may be the “hob-gobblin of small minds,” but in coding it is a key virtue. If you’ve ever had to read and maintain a large set of programs written by someone else, you’ll know what I mean.

Write Structured Code   -- especially for large programs!

Structured programming has been established for over twenty years ago as a superior programming methodology, especially for large programs. Use it if you wish to write maintainable code.

Write Modular Code   -- especially for large programs!

Modular principles complement structured programming principles. Both have long been proven superior coding techniques. If you don’t believe me, you might read Yourdon and Constantine.

Declare All Variables   -- especially for large programs!

Not declaring variables is easy and convenient for small scripts. But declare all variables for larger systems!   Otherwise you’ll likely experience errors based on uninitialized variables. Use the **NOVALUE** built-in exception handler to trap any variable that has not been properly assigned a value.

Don't Use Global Variables ***-- especially for large programs!***

Programmers often use global variables because they're easier than passing arguments. But for large programs, the most common maintenance errors are due to global variables. I recommend you not use global variables, especially on larger projects. Sometimes a configuration or **ini** file works well for storing global variables, or even a single routine that is effectively a just memory definition or "control block" definition.

[This material was extracted and abbreviated from the book *Rexx Programmer's Reference*. The book provides fuller explanations and examples of the rules briefly listed here.]