The Power of Associative Arrays

By Howard Fosdick, *Dr Dobbs Journal* July 25, 2006

Associative arrays, also referred to as maps, hashes, dictionaries, finite maps, lookup tables, and the like, are abstract data types composed of a collection of keys and values

Here's a problem for you. You need to match all the names in two lists, and create a third list consisting of names in the second list but not in the first. Only exact matches count. How do you do it?

You could read the names into two arrays (tables). Then read each name in one of the arrays, and search through the entire other array for a match. Copy names that don't match into a third array.

You might consider sorting the arrays to speed up the searching and matching process. And you might eliminate I/O time by processing only in memory.

Les Koehler faced this problem in the real world. He had an accounting program on his hands, written by a predecessor, that matched the accounting records they had accumulated against the accounting records that a vendor sent them daily. The format and field definitions varied between the two files. The program ran for several hours to process 25,000 records using the simple "walk the list" technique I just described. Les reduced the run time to *less than a minute* using associative arrays. In this article, I explain how he did it.

Associative Arrays

Conventional arrays or tables are simply lists of items indexed by numeric subscripts. For example, a simple list or one-dimensional array might look like this:

```
array_name.1
array_name.2
 ...etc...
```

Subscripts can be variables, but those variables must resolve to numeric values.

Associative arrays genericize this concept. They permit entry into the array by arbitrary strings. So entries in the array are referenced by their name, rather than by their location in the array. The array can be considered a collection of keys and related values. For example, you might have:

```
array_name.string_index_a
array_name.string_index_b
   ... etc ...
```

Subscripts, such as **string_index_a**, could either be a literal value or a variable that resolves into some value. That value could be anything--string or numeric.

The relationship between a key and its value is sometimes called a "mapping" or "binding." The most important associative operation is the simple lookup or "indexing."

A Solution Coded in Rexx

Most modern scripting languages support associative arrays, including Perl, Python, and Tcl/Tk. Les used Rexx for his solution. Rexx brings together readability and power--two characteristics few languages combine. It enjoys an ANSI standard and comes in procedural, object-oriented, and Java-compatible forms.

Object-oriented Rexx is a true super-set of classic procedural Rexx. It runs procedural Rexx programs without alteration. Open Object Rexx has garnered significant interest since its open-sourcing by IBM and subsequent hand-over to the Rexx Language Association.

Rexx is internationally popular and runs under virtually all operating systems. Open Object Rexx runs on Windows, Linux, and Unix. Classic procedural Rexx is the predominate scripting language on mainframes (z/OS, z/VM, and z/VSE), the IBM iSeries (i5/OS and AS/400), OS/2 (including eCS and osFree), AmigaOS (including AROS and MorphOS), and IBM PC-DOS (versions 2000 and 7). Java-compatible Rexx runs with any Java Virtual Machine (JVM).

A Solution

Here’a solution to the problem, coded in Rexx using associative arrays. I'll walk through the code, line by line, in the discussion that follows.

```
/* Create an associative array reflecting */
/* the values in the first list of names  */

flag. = 0                          /* Create array, initialize elements to 0  */
do a = 1 to list_a.0               /* Process all the names in LIST_A array   */
      aa = strip(list_a.a)         /* Strip out any preceding/trailing blanks */
      flag.aa = 1                  /* Mark the name with a 1                  */
end

/* Try to match names in the second list   */
/*  against those in the associative array */

m = 0                              /* M counts the number of missing names   */
do b = 1 to list_b.0               /* Look for matching name from LIST_B     */
      bb = strip(list_b.b)         /* Put LIST_B name into variable BB       */
      if  \ flag.bb then do        /* If the name isn't in FLAG array        */
          m = m+1                  /*    add 1 to the count of missing names */
          missing.m = bb           /*    add missing name to MISSING array   */
      end
end
missing.0 = m                      /* Save the count of unmatched names      */
```

In Rexx, arrays are expressed in the form of *compound variables*, two or more variable names strung together by periods. The entire array is referenced by the name of the array followed by a period.

The first line of code in the above program refers to an array named **flag**, denoted by the array name **flag** followed immediately by a period. So this first line creates the associative array named **flag** and initializes all possible elements to 0:

```
flag. = 0  /* Create an array, initialize elements to 0 */
```

All Rexx arrays are dynamic, so we do not need to specify a size for the flag array.

In Rexx, you commonly store the number of array elements in the first array position (denoted by the 0 subscript). So for an array named **list_a**, array element **list_a.0** holds the number of items in the array. This do loop thus processes all the names in the array named **list_a**:

```
do a = 1 to list_a.0  /* Process all the names in LIST_A array */
```

The first line inside the **do** loop removes any leading or trailing blanks from the name through the **strip** function. It places the result in the variable:

```
aa = strip(list_a.a) /* Strip out preceding/trailing blanks   */
```

Next we mark the presence of the name in the flag array by flagging it. Here you see the use of the associative array. We denote that a value exists simply by using that value as the subscript into the **flag** array:

```
flag.aa = 1  /*  Mark the name with a 1 */
```

At the conclusion of the **do** loop, the **flag** array consists of a group of name indexes that are flagged as present.

The second **do** loop looks at each name in the second array, called **list_b**, and sees if it exists as a flagged member in the **flag** array. If so, we have matched names between the two lists, **list_a** and **list_b**.

The first line in the second **do** loop processes all names in the second array, called **list_b**:

```
do b = 1 to list_b.0 /* Look for matching name from LIST_B    */
```

The next line in the second **do** loop removes leading and trailing blanks from a name in **list_b**, and places that name into the variable **bb**:

```
bb = strip(list_b.b) /*  Put LIST_B name into variable BB  */
```

Now we can subscript the **flag** array with this name from the second list. If it does not exist in the **flag** array (denoted by the backslash symbol "\" meaning "NOT"), then we know we have a name from the second list that does not exist in the first list:

```
if  \ flag.bb then do  /* If the name isn't in FLAG array */
```

If the name does not exist, we add **1** to the count of unmatched names. We also add the missing name to the list of missing names in the array we've named missing:

```
m = m+1          /*    add 1 to the count of missing names */
missing.m = bb   /*    add missing name to MISSING array   */
```

There is no need to "declare" or pre-define an array in Rexx. Define it simply by using it, as we do above in our first reference to the missing array.

The last line in the routine sets the total count of missing names in the missing array. In Rexx, by convention we store this value as element 0 in that array:

```
missing.0 = m  /* Save the count of unmatched names     */
```

After the code executes, the missing array contains all names from the second list that are not in the first list. The first element of the missing array, **missing.0**, contains the number of unmatched names.

More Real-World Examples

The coding solution above, implemented in Rexx, is pretty flexible. Arrays can be defined in advance or through first use. Array sizes do not have to be specified; they are dynamic up to the size of available memory.

Rexx allows any value for indexing an associative array, including numeric values and character, bit, or hex strings. Rexx indexing even works with strings that contain illegal character values! This was key to the solution Les devised because the two input files were in different formats and had different data type definitions.

While associative arrays are straightforward, they have many useful applications. As Les observes, "with a little imagination, ... this technique can be applied to a lot of situations where you want to associate one or more sets of data with some arbitrary index for lookup purposes."

Frank Clarke faced just such a situation. He used associative array processing to drive an 11-minute process down to 9 seconds. Frank dryly notes, "The reduced code ran so fast everyone assumed it had failed."

Bob Hamilton's rewrite of an ADABAS/Natural script with associative arrays produced similar benefits. The 17-hour program tied up the entire system while reading through 900,000 records looking for student ID matches. Since associative arrays enabled a single scan of the data, his result was a five-minute run.

And now we present a possible world record-holder: Steve Coalbran who faced a legacy PL/I program that took 18 hours to compare about 10,000 records in a standard table to a database. He scrapped the program and rewrote the application using list processing with associative arrays and far fewer file OPENs

and CLOSEs. The program's run time dropped to 4.7 seconds. Steve ended up having to prove his solution really worked to two disbelieving operations analysts!

For more information on Rexx – including free downloads – go to www.RexxInfo.org.