Using an Array as an If-Switch

Nazik Elgaddal and Ed Heaton, Westat, Rockville, MD

Abstract

Do you sometimes find yourself using nested **IF** statements or nested **SELECT** blocks? Does the code become more difficult to develop, read, or maintain as the nesting gets deeper? Multi-dimensional arrays can replace that code with an array declaration and a simple assignment statement.

Have you ever used multi-dimensional arrays? They really can make your work more elegant. This paper will explore the nature of multi-dimensional arrays as it develops what Ian Whitlock calls an *array implementation of a master **IF**-switch*.

SAS® allows 28 dimensions to its arrays. We doubt you would ever need that many dimensions; and your computer probably doesn't have the memory to process such an array. We recently suggested to a fellow SAS programmer that there was probably never a reason to use an array of more than three dimensions. Less than an hour later, we developed a job that used a four-dimensional array! It accomplished, in 59 lines of easy code, a task that used 289 lines of **SELECT** blocks. This paper will show how it was done.

About Arrays

What is a SAS array?

A SAS variable array is a convenient way of temporarily arranging a group of variables that are identified by an array-name. The array name is not a variable; it simply identifies the array in the **DATA** step. Arrays exist only during the duration of the **DATA** step.

Variable Arrays

Variable arrays in SAS are different from arrays in most other programming languages. In SAS, a variable array is an indexed list of pointers to variables in the Program Data Vector (PDV). The syntax of a one-dimensional variable array is

```
Array arrayName [numberOfVars] <$><w>
   <variableList>
   <(initialValues)>
;
```

All of the variables in the variable list must be of the same type, either character or numeric. This simplest kind of variable array has subscripts beginning with the number one and ending with the number in the brackets (*numberOfVars*). SAS will not allow you to assign an array name which is the name of a variable that is already in the data set. Furthermore, it is probably a bad idea to use the name of a SAS function as the array name. Why? Because SAS will allow you to use parentheses, as well as brackets and braces, to encase the array indices. So, it can become unclear whether code is an array reference or a function call. E.g., we can define an array as

```
Array sqRt (5) ( 10 20 30 40 50 ) ;
```

If we then code

```
x = sqRt(4) ;
Put "The square root of 4 is " x "." ;
```

"The square root of 4 is 40 ." will be written to the log. This is obviously an error; SAS assumed **SQRT** was the name of the array rather than the name of the function. In fact, the **SQRT** function is no longer available to us in this **DATA** step! SAS could correct this problem by requiring us to use brackets, **[]**, or braces, **{}** for array indices and reserving parentheses for functions; but they do not. For clarity in your code, we suggest that you use either brackets or braces for your array references. We will use brackets in this paper.

You don't have to specify the size of a variable array. You can code something like

```
Array nazik [*] $15 child1-child3 (
   "Mehera" "Maha" "Marwa"
) ;
```

and the asterisk will tell SAS to count the variables to determine the size of the array. The remainder of your code can get the size of the array using the **DIM** function. E.g.:

```
Do i=1 to dim(nazik) ;
```

or

```
NumberChildren = dim(nazik) ;
```

Sometimes, we don't want the array indices to start with one. For example, suppose we have a variable for each year from 1998 through 2003. We might want to specify the array as

```
Array inc [1998:2003] gross1-gross6 ;
```

so that we can use an existing variable that contains the year for the array index. Our code can fetch the first index using the **LBOUND** function and it can get the last index using the **HBOUND** function. So, one can code

```
Do i=lBound(inc) to hBound(inc) ;
```

to loop through the array.

Memory Considerations

Let's investigate the memory usage for arrays. We need a baseline. A **DATA** step that does nothing takes about 89 kilobytes of memory under SAS version 9.0 running under Windows 2000.

```
 1 Data _null_ ;
 2 Run ;

NOTE: DATA statement used
      Memory                          89k
```

Suppose we create a data set with 10k variables using SAS version 9.0.

```
 3 Data test ;
 4    Retain x1-x10240 0 ;
 5 Run ;

NOTE: The data set WORK.TEST has 1
      observations and 10240
      variables.
NOTE: DATA statement used
      Memory                        2393k
```

Each of these numeric variables is eight bytes long, but they take about 230 bytes of memory while in the **DATA** step.

$$\frac{2393-89}{10}=230.4$$

Why so much memory? Because SAS has to provide space for storing the data type, the format and informat name, the variable label and length, etc. for each variable.

There is a slight memory impact when you use an array. To test this, let's first associate these 10k variables with two arrays. We used two arrays so that any memory usage that is not additive with each array will stabilize.

```
 6 Data _null_ ;
 7    Set test ;
 8    Array a [*] x: ;
 9    Array b [*] x: ;
10 Run ;

NOTE: DATA statement used
      Memory                        2308k
```

Now let's add one more array.

```
11 Data _null_ ;
12    Set test ;
13    Array a [*] x: ;
14    Array b [*] x: ;
15    Array c [*] x: ;
16 Run ;

NOTE: DATA statement used
      Memory                        2349k
```

It seems to take about four bytes of memory to associate each variable with the array.

$$\frac{2349\text{k}-2308\text{k}}{10\text{k}}=4.1 \text{ bytes.}$$

TEMPORARY Arrays

While a variable array is temporary, the data in the array are not temporary; they are stored in data set variables. SAS allows arrays where both the array and the data are temporary. These arrays use far less memory because SAS does not need to store all of the information that is pertinent only to variables.

To create an array that contains temporary data, use the key word **_TEMPORARY_** in the **ARRAY** statement where you would list the variables for a variable array. You can create an array for ages with the following code.

```
Array age [3] _temporary_ ;
```

You can initialize some or all of the elements in the array as follows.

```
Array age [3] _temporary_ ( 14 12 ) ;
```

You can initialize all of the elements to a single value (e.g., zero) as follows.

```
Array a [10] _temporary_ ( 10 * 0 ) ;
```

Temporary arrays are more like traditional arrays from other languages; they reserve a contiguous block of memory that is accessed through the indices. Temporary arrays are not written to the output data set; the array elements are not SAS variables. Values of temporary array elements are automatically retained; they are not reset to missing at the beginning of each iteration of the **DATA** step. They do not have names or other variable attributes except for the data type and, for character arrays, the length of each element.

Let's create a **_TEMPORARY_** array with 10k numeric elements.

```
17 Data _null_ ;
18    Array a [10240] _temporary_ ;
19 Run ;

NOTE: DATA statement used
      Memory                         175k
```

Remember that a **DATA** step that does nothing takes 89k of memory. This 10k **_TEMPORARY_** array of 8-byte numbers used only 8.6 bytes per element.

$$\frac{175\text{k}-89\text{k}}{10\text{k}}=8.6 \text{ bytes.}$$

You cannot use an asterisk, **[*]**, for the dimension in a temporary array declaration because SAS will not be able to determine its size since it has no variables to count.

Multi-dimensional Arrays

Single-dimensional arrays represent vectors; multi-dimensional arrays are traditionally used to represent tables, cubes, etc.

Multi-dimensional arrays are created by specifying the number of elements in each dimension. E.g., the following statements define and use a two-dimensional array with two rows and three columns.

```
Array nazik [2,3]
   child1-child3
   school1-school3
   (
      "Mehera"       "Maha"          "Marwa"
      "Spring Brook" "Francis Scott" "Burnt's Mill"
   )
;
Do i=1 to dim(nazik,2) ;
   Put
      nazik(1,i) " goes to "
      nazik(2,i) " school."
   ;
End ;
```

Performance Considerations

While arrays make our code more readable and easier to develop and maintain, it comes with a performance cost. Consider the cost of looping through the array. Each time the **DO** statement executes, SAS has to increment the iteration counter and compare it with the stop value. (These tests were run under Windows 2000 with 512 megabytes of memory.)

```
20 Data _null_ ;
21    Do i=1 to 10**7 ;
22    End ;
23 Run ;

NOTE: DATA statement used
      (Total process time):
      real time           0.21 seconds
      user cpu time       0.20 seconds
      system cpu time     0.00 seconds
      Memory                94k
```

It takes about 0.2×10^{-7} seconds per loop for the increment-and-compare step. It also takes time to test whether the index is in the bounds of the array. To find this time-cost, we need a baseline. It takes some time to build our array of ten million elements, but far less time than it would take to build a similar variable array.

```
24 Data _null_ ;
25    Array x [10000000] _temporary_ ;
26    Do i=1 to 10**7 ;
27    End ;
28 Run ;

NOTE: DATA statement used
      (Total process time):
      real time           1.28 seconds
      user cpu time       1.23 seconds
      system cpu time     0.05 seconds
      Memory                78220k
```

It takes very little more time to embed an assignment statement in our loop.

```
29 Data _null_ ;
30    Array x [10000000] _temporary_ ;
31    Do i=1 to 10**7 ;
32       y = i ;
33    End ;
34 Run ;

NOTE: DATA statement used
      (Total process time):
      real time           1.33 seconds
      user cpu time       1.27 seconds
      system cpu time     0.05 seconds
      Memory                78220k
```

Now we have a baseline. Let's do exactly the same process but with the assignment to an array rather than a constant variable.

```
35 Data _null_ ;
36    Array x [10000000] _temporary_ ;
37    Do i=1 to 10**7 ;
38       x[i] = i ;
39    End ;
40 Run ;

NOTE: DATA statement used
      (Total process time):
      real time           2.00 seconds
      user cpu time       1.94 seconds
      system cpu time     0.05 seconds
      Memory                78220k
```

It seems to take about 0.67×10^{-7} seconds to perform the array lookup.

$$\frac{1.94 \text{ seconds} - 1.27 \text{ seconds}}{10^7 \text{ assignments}} = 0.67\times10^{-7} \text{ seconds/assignment.}$$

The CPU cost of the variable lookup is over three times the cost of the **DO** loop.

Arrays as IF Switches

Arrays can be used to make decisions. Suppose we want to collapse the levels of a variable that holds the Federal Information Processing Standard (FIPS) for states. These are integers from 1 to 56; most of these numbers represent a state but some are not used. Now suppose we want to map these to regions. We can use **IF** statements, a **SELECT** clause, or a value format to map these states to regions. We can also use what Ian Whitlock calls an *array if-switch*.

```
Array stToRgn [56] _temporary_ (
/* state  region   state  region */
  /* 01 */  2     /* 02 */  4
  /* 03 */ .E     /* 04 */  4
  /* 05 */  2     /* 06 */  4
  /* 07 */ .E     /* 08 */  4
  /* 09 */  1     /* 10 */ .E
  /* 11 */  1     /* 12 */  2
  /* 13 */  2     /* 14 */ .E
  /* 15 */  4     /* 16 */  4
  /* 17 */  3     /* 18 */  3
  /* 19 */  3     /* 20 */  3
  /* 21 */  2     /* 22 */  2
  /* 23 */  1     /* 24 */  1
  /* 25 */  1     /* 26 */  3
  /* 27 */  3     /* 28 */  2
  /* 29 */  3     /* 30 */  4
  /* 31 */  3     /* 32 */  4
  /* 33 */  1     /* 34 */  1
  /* 35 */  4     /* 36 */  1
  /* 37 */  2     /* 38 */  3
  /* 39 */  3     /* 40 */  4
  /* 41 */  4     /* 42 */  1
  /* 43 */ .E     /* 44 */  1
  /* 45 */  2     /* 46 */  3
  /* 47 */  2     /* 48 */  4
  /* 49 */  4     /* 50 */  1
  /* 51 */  2     /* 52 */ .E
  /* 53 */  4     /* 54 */  2
  /* 55 */  3     /* 56 */  4
) ;
Region = stToRgn[State] ;
```

You may say "What's the advantage in this method?" Well, there may be no advantage for such a simple collapsing of levels. However, sometimes the map requires nested **IF** statements or **SELECT** statements. Let's look at one of these.

The Array If-Switch with Three Control Variables:

Suppose we have specifications as follows, taken directly from an actual specification document:

---

3. Sort Sampling Frame

Prior to sampling, assign a "permanent" random number between 0 and 1 to each of the 83,513 schools in the frame using the uniform random number generator in SAS. Call this random number RAND. Next, sort the schools in the frame by STR83, and then within each level of STR83, further sort the schools by the variables listed in the last column of Table 7. For example, within stratum 1 (STR83=1), schools should be sorted by URBAN and then by RAND within URBAN. Within stratum 2 (STR83=2), schools should be sorted by URBAN, OEREG within URBAN, and finally by RAND within OEREG, and so on.

- LEVEL takes values of 1 for elementary school and 2 for secondary school.
- SIZCL takes values of
  - 1 for fewer than 300 students
  - 2 for 300 to 499 students
  - 3 for 500 to 999 students
  - 4 for 1000 to 1499 students
  - 5 for 1500 students or more
- POVST takes values of
  - 0 for missing (recoded by programmer)
  - 1 for less than 35%
  - 2 for 35% to 49% (percents are integers)
  - 3 for 50% to 74%
  - 4 for 75% to 100%

Table 7 is reproduced as an Appendix 1.

A costly solution to this problem would partition the data into 50 data sets – one for each stratum. Then sort each of the data sets by its particular requirement. Finally, put the data sets back together in order of **STRATUM**. This approach will not be used.

A more efficient algorithm would create a sorting variable. This we will do.

We notice that the column specifying the sort variables uses the random number (**RAND**) for each row. Furthermore, that random variable is the last of the variables used for each level of the sort. Whenever **URBAN** and **OEREG** are used in the sort, they are always used in that order. Both **URBAN** and **OEREG** have values from 1 to 4.

We now have enough information to create a sorting variable. Simply multiply **URBAN** by 10 and then add **OEREG** and a number on the interval (0,1) generated by the **RANUNI** function. We don't even have to create the **RAND** variable.

Of course we still need to create the **STRATUM** and **MOS** variables.

We could create these variables using nested If statements or Select statements. That solution, in part, is included in Appendix 2. The full **DATA** step contains 292 lines of code. A far simpler approach is to use a **_TEMPORARY_** array to make the decisions and then assign the appropriate value.

We need four dimensions to accomplish this task using a **_TEMPORARY_** array.

- The first dimension specifies the type of school (**LEVEL**) and has two levels.
- The second dimension specifies the size of the school (**SIZECL**) and has five levels.
- The third dimension specifies the poverty status (**POVST**) based on the percentage of the students who are eligible for free or reduced-price lunches. Some of these are missing. Missing values will not work as an array index, so we have recoded the missing values to zero. The levels of this dimension range from 0 to 4.
- The fourth dimension has four levels to hold the needed values from Table 7.
  - The first value is the stratum.
  - The second value is the desired sampling rate (**MOS**).
  - The third value specifies whether the **URBAN** variable is used in the sort. (**1**=yes, **0**=no)
  - The fourth value specifies whether the **OEREG** variable is used in the sort.

We copied the last three columns of Table 7 and pasted them into our **ARRAY** statement.

```
Array t [2,5,0:4,4] _temporary_ (
    1   0.005461   URBAN        RAND
    2   0.004368   URBAN OEREG  RAND
    3   0.005461   URBAN        RAND
    4   0.005461   URBAN        RAND
    5   0.008401   URBAN        RAND
    6   0.008917   URBAN OEREG  RAND
    7   0.007134   URBAN OEREG  RAND
    8   0.008917   URBAN        RAND
etc., through
   49   0.051197   URBAN        RAND
   50   0.073138                RAND
) ;
```

We replaced every occurrence of "**URBAN OEREG RAND**" with "**1 1**". Next we replaced every occurrence of "**URBAN RAND**" with "**1 0**". Finally we replaced every occurrence of "**RAND**" with "**0 0**". The resulting **DATA** step follows.

```
%let seed = 235016 ;
Data Frame ;
   Set demo.Frame ;
   Array t[2,5,0:4,4] _temporary_ (
   /* Stratum   MOS    Urban OEReg */
         1    0.005461   1     0
         2    0.004368   1     1
         3    0.005461   1     0
         4    0.005461   1     0
         5    0.008401   1     0
         6    0.008917   1     1
         7    0.007134   1     1
         8    0.008917   1     0
         9    0.008917   1     0
        10    0.013719   1     1
        11    0.012210   1     1
        12    0.009768   1     1
        13    0.012210   1     1
        14    0.012210   1     1
        15    0.018785   1     1
        16    0.015763   0     0
        17    0.012611   1     0
        18    0.015763   0     0
        19    0.015763   0     0
        20    0.024251   1     0
        21    0.019939   0     0
        22    0.015951   0     0
        23    0.019939   0     0
        24    0.019939   0     0
        25    0.030676   0     0
        26    0.014021   1     0
etc., through
        49    0.051197   1     0
        50    0.073138   0     0
   ) ;
   Stratum = t[Level,SizCl,PovSt,1] ;
   MOS     = t[Level,SizCl,PovSt,2] ;
   SortVar =
       t[Level,SizCl,PovSt,3]*10*Urban
     + t[Level,SizCl,PovSt,4]*OEReg
     + ranUni(&seed)
   ;
Run ;
```

Now a simple **SORT** procedure – sorting on the **SORTVAR** variable – gives us our desired result. Our **DATA** step used a **SET** statement, one **ARRAY** statement, and three assignment statements. The values in the array statement were pasted directly from our supplied table (cut-and-paste) and then modified with three global search-and-replace operations. The same task using **SELECT** blocks involves 292 statements! (See Appendix 2.)

The job using **SELECT** blocks ran in 0.16 seconds and used 176k of memory. The version using the array took 0.28 seconds and but used only 105k of memory. So, it's a judgment call; one way runs faster, the other uses less memory.

Conclusion

By using a multi-dimensional array we save development time, run time, and memory. As programmers, it's natural to automatically think of **IF** statements and **SELECT** blocks for making decisions. We need to add the array to the list of decision making tools. Then, when we face a decision-making task, our job will be to determine which of these is best for that task. Of course, there are still more options for making decisions that we have not covered in this paper.

Disclaimer

The contents of this paper are the work of the authors and do not necessarily represent the opinions, recommendations, or practices of Westat.

References

Acknowledgements

We want to thank Ian Whitlock who provided the name, *array if-switch*, when we first described the process to him. We want to thank Mike Rhoads and Duke Owen of Westat for reviewing this paper and for their suggestions.

We also want to thank Bernadette Mahony of Westat who provided the project work where this technique was developed.

Contact Information

Your comments and questions are valued and encouraged. Contact the authors at:

Nazik Elgaddal
Westat
1650 Research Boulevard
Rockville, MD 20850
Phone: (301) 517-4017
Fax: (301) 294-3992
NazikElgaddal@Westat.com

Edward Heaton
Westat
1650 Research Boulevard
Rockville, MD 20850
Phone: (301) 610-4818
Fax: (301) 610-5128
EdHeaton@Westat.com

Appendix 1

Table 7. Stratum code and desired sampling rate to be assigned for sampling purposes

Level	*SizCl*	*PovSt*	*Stratum*	*MOS*	*Sort Variables*
1. Elementary	1. <300	0. Missing	1	0.005461	URBAN RAND
		1. <35%	2	0.004368	URBAN OEREG RAND
		2. 35 to 49%	3	0.005461	URBAN RAND
		3. 50 to 74%	4	0.005461	URBAN RAND
		4. 75%+	5	0.008401	URBAN RAND
	2. 300-499	0. Missing	6	0.008917	URBAN OEREG RAND
		1. <35%	7	0.007134	URBAN OEREG RAND
		2. 35 to 49%	8	0.008917	URBAN RAND
		3. 50 to 74%	9	0.008917	URBAN RAND
		4. 75%+	10	0.013719	URBAN OEREG RAND
	3. 500-999	0. Missing	11	0.012210	URBAN OEREG RAND
		1. <35%	12	0.009768	URBAN OEREG RAND
		2. 35 to 49%	13	0.012210	URBAN OEREG RAND
		3. 50 to 74%	14	0.012210	URBAN OEREG RAND
		4. 75%+	15	0.018785	URBAN OEREG RAND
	4. 1000-1499	0. Missing	16	0.015763	RAND
		1. <35%	17	0.012611	URBAN RAND
		2. 35 to 49%	18	0.015763	RAND
		3. 50 to 74%	19	0.015763	RAND
		4. 75%+	20	0.024251	URBAN RAND
	5. 1,500+	0. Missing	21	0.019939	RAND
		1. <35%	22	0.015951	RAND
		2. 35 to 49%	23	0.019939	RAND
		3. 50 to 74%	24	0.019939	RAND
		4. 75%+	25	0.030676	RAND
2. Secondary	1. <300	0. Missing	26	0.014021	URBAN RAND
		1. <35%	27	0.013482	URBAN OEREG RAND
		2. 35 to 49%	28	0.014021	URBAN RAND
		3. 50 to 74%	29	0.014021	RAND
		4. 75%+	30	0.020030	RAND
	2. 300-499	0. Missing	31	0.022896	URBAN RAND
		1. <35%	32	0.022015	URBAN OEREG RAND
		2. 35 to 49%	33	0.022896	RAND
		3. 50 to 74%	34	0.022896	RAND
		4. 75%+	35	0.032708	RAND
	3. 500-999	0. Missing	36	0.031352	URBAN OEREG RAND
		1. <35%	37	0.030146	URBAN OEREG RAND
		2. 35 to 49%	38	0.031352	URBAN RAND
		3. 50 to 74%	39	0.031352	URBAN RAND
		4. 75%+	40	0.044788	URBAN RAND
	4. 1000-1499	0. Missing	41	0.040475	URBAN RAND
		1. <35%	42	0.038918	URBAN OEREG RAND
		2. 35 to 49%	43	0.040475	URBAN RAND
		3. 50 to 74%	44	0.040475	RAND
		4. 75%+	45	0.057821	RAND
	5. 1,500+	0. Missing	46	0.051197	URBAN RAND
		1. <35%	47	0.049228	URBAN OEREG RAND
		2. 35 to 49%	48	0.051197	URBAN RAND
		3. 50 to 74%	49	0.051197	URBAN RAND
		4. 75%+	50	0.073138	RAND

Appendix 2

```
%let seed = 235016 ;
Data Frame ;
   Set demo.Frame ;
   Select (Level) ;
      When (1) select (SizCl) ;
         When (1) select (PovSt) ;
            When (0) do ;
               Stratum = 1 ;
               MOS = 0.005461 ;
               SortVar = Urban*10 + ranUni(&seed) ;
            End ;
            When (1) do ;
               Stratum = 2 ;
               MOS = 0.004368 ;
               SortVar = Urban*10 + OeReg + ranUni(&seed) ;
            End ;
            When (2) do ;
               Stratum = 3 ;
               MOS = 0.005461 ;
               SortVar = Urban*10 + ranUni(&seed) ;
            End ;
            When (3) do ;
               Stratum = 4 ;
               MOS = 0.005461 ;
               SortVar = Urban*10 + ranUni(&seed) ;
            End ;
            When (4) do ;
               Stratum = 5 ;
               MOS = 0.008401 ;
               SortVar = Urban*10 + ranUni(&seed) ;
            End ;
            Otherwise put "ERROR: Unexpected value (" PovSt= ")." ;
         End ;
         When (2) select (PovSt) ;
            When (0) do ;
               Stratum = 6 ;
               MOS = 0.008917 ;
               SortVar = Urban*10 + OeReg + ranUni(&seed) ;
            End ;
            When (1) do ;
               Stratum = 7 ;
               MOS = 0.007134 ;
               SortVar = Urban*10 + OeReg + ranUni(&seed) ;
            End ;
            When (2) do ;
               Stratum = 8 ;
               MOS = 0.008917 ;
```

```
            SortVar = Urban*10 + ranUni(&seed) ;
         End ;
         When (3) do ;
            Stratum = 9 ;
            MOS = 0.008917 ;
            SortVar = Urban*10 + ranUni(&seed) ;
         End ;
         When (4) do ;
            Stratum = 10 ;
            MOS = 0.013719 ;
            SortVar = Urban*10 + OeReg + ranUni(&seed) ;
         End ;
         Otherwise put "ERROR: Unexpected value (" PovSt= ")." ;
      End ;
      When (3) select (PovSt) ;
         When (0) do ;
            Stratum = 11 ;
            MOS = 0.012210 ;
            SortVar = Urban*10 + OeReg + ranUni(&seed) ;
         End ;
         When (1) do ;
            Stratum = 12 ;
            MOS = 0.009768 ;
            SortVar = Urban*10 + OeReg + ranUni(&seed) ;
         End ;
         When (2) do ;
            Stratum = 13 ;
            MOS = 0.012210 ;
            SortVar = Urban*10 + OeReg + ranUni(&seed) ;
         End ;
         When (3) do ;
            Stratum = 14 ;
            MOS = 0.012210 ;
            SortVar = Urban*10 + OeReg + ranUni(&seed) ;
         End ;
         When (4) do ;
            Stratum = 15 ;
            MOS = 0.018785 ;
            SortVar = Urban*10 + OeReg + ranUni(&seed) ;
         End ;
         Otherwise put "ERROR: Unexpected value (" PovSt= ")." ;
      End ;
```

etc. through

```
      Otherwise put "ERROR: Unexpected value (" SizCl= ")." ;
   End ;
   Otherwise put "ERROR: Unexpected value (" Level= ")." ;
End ;
Run ;
```