Arrays from A to Z

Phil Spector

Statistical Computing Facility
Department of Statistics
University of California, Berkeley

http://www.stat.berkeley.edu/~spector

What is a SAS Array?

An array in SAS provides a means for repetitively processing variables using a do-loop. Arrays are merely a convenient way of grouping variables, and do not persist beyond the data step in which they are used.

SAS arrays can be used for simple repetitive tasks, reshaping data sets, and "remembering" values from observation-to-observation.

Arrays can be used to allow some traditional matrix-style programming techniques to be used in the data step.

Array Statement: Syntax

ARRAY *name*<{*nelem*}> <$> <<*elements* <(*initial-values*)>>;

Examples:

```
array x x1-x3;
array check{5} _temporary_;
array miss{4} _temporary_ (9 9 99 9);
array dept $ dept1-dept4 ('Sales','Research','Training');
array value{3};    * generates value1, value2 and value3;
```

- All variables in an array must have the same type (numeric or character).
- An array name can't have the same name as a variable.
- You must explicitly state the number of elements when using `_temporary_`; in other cases SAS figures it out from context, generating new variables if necessary.

Advanced Features of Arrays

- You can specify the range of subscripts in an array with the notation ***`start:finish`***. For example, the declaration:
  ```
  array income{1997:2000}  in1 - in4;
  ```
  would allow you to refer to `income{1997}`, `income{1998}`, etc. The functions `lbound` and `hbound` will return the lowest and highest indices defined for an array.
- Array names can be used in `RETAIN` statements, and, when used with the subscript `{*}` in `PUT` or `INPUT` statements;
- If an array name coincides with the name of a SAS function, the array will override the function for the duration of the data step.
- When an array is declared using `_temporary_`, values of the elements of the array are not set to missing at the beginning of each observation.

Past and Future

When arrays were first introduced, they were designed to be used implicitly. The array name itself (unsubscripted) represented the variable defined in the array statement and the `do over;` group allowed repetitive processing. When it was necessary to access elements by position, a counter variable could be specified, or the system variable `_i_` was used.

While this form of array is supported for legacy applications, it should be avoided for new work, and explicit subscripting should always be used.

Version 7 of the SAS system will allow using an array name in a variable list to represent all the elements of the array, as well as more flexible syntax for initializing the array.

A Simple Example

Suppose we have a data set with 10 variables, named x1,x2,. . .,x10. Whenever any of these variables has a value of 9, we wish to replace it with a missing value (`.`).

```
data new;
   set old;
   array x x1-x10;
   do i=1 to dim(x);
      if x{i} = 9 then x{i} = .;
      end;
run;
```

- Using `dim(x)` instead of a constant (10) eliminates the need to know the size of the array.
- Special variable lists (like `first -- last` or `x:`) can be very useful when setting up an array.

Using Parallel Arrays

In the previous example, suppose missing values had been coded as 9 for variables `x1-x5`, and as 99 for the remaining variables.

```
data new;
    set old;
    array x x1-x10;
    array mval _temporary_ (9 9 9 9 9 99 99 99 99 99);
    do i=1 to dim(x);
       if x{i} = mval{i} then x{i} = .;
       end;
run;
```

- If you don't use `_temporary__`, you usually need to include a `drop` statement.
- There's no limit to the number of parallel arrays you can create.

Another Example of the `array` Statement

The array statement is especially useful when you need to make logical decisions about a set of variables, since these things can't be done with either summary procedures or data set functions.

Consider a data set with variables `class1` through `class5`, containing the number of classes taken in each year of college, and we want to find how many years it takes each student to complete 10 courses:

```
array class class1-class5;
total = 0;
do i = 1 to 5 until(total >= 10);
    total = total + class{i};
    end;
year = i;
if total lt 10 then year = .;
```

Reshaping Data Sets: I. One to Many

Some SAS procedures require all observations for an experimental unit to be included in a single observation in the data set; others require that each individual observation is a separate observation in the data set. Thus, it is often useful to convert between the two cases.

Consider a data set with 4 variables (`x1-x4`) stored as follows:

```
ID    X1    X2    X3    X4
 1    17    19    22    24
 2    18    14    33    16
 3    19    28    31    42
            . . .
```

The goal is to create four observations for each original observation, one for each variable.

Reshaping Data Sets: Example 1

The strategy is to store the four observations in an array, and loop over the array, creating a new observation each time. A `drop` or `keep` statement is essential for programs like this.

```
data new;
set old;
array xx x1-x4;
do time=1 to 4;
   x = xx{time};
   output;
   end;
drop x1-x4;
```

A Review of the `retain` Statement

By default SAS initializes all the variables in a data set to missing each time it processes a new observations. To "remember" data from other observations, variables may need to be declared using the `retain` statement.

If initial values are specified for any variables in an array, then all the variables in the array are automatically retained. (Note that you may still have to reinitialize the variables when you are using `by`-groups).

Since arrays declared as `_temporary_` do not contain variables, they do not get reset to missing at the beginning of each observation.

Reshaping Data Sets II - One to Many

To create a data set where many observations are combined into a single observation, we create an array to hold the individual variables, and then output them after the last observation for each experimental unit. The basic transformation can be represented as follows:

```
Subj   Time    X
  1      1    10
  1      2    12
          ...            Subj   X1   X2  ...  Xn
  1      n     8   ==>     1    10   12  ...   8
  2      1    19           2    19    7  ...  21
  2      2     7
          ...
  2      n    21
```

Strategy for Many to One Transformations

Programs which merge several observations into a new, single observation can usually be broken down into four parts:

1. Declaration of arrays and specification of `retain`ed variables as needed.
2. Initialization at the beginning of each experimental unit. Special care needs to be taken with values stored in arrays.
3. Storage and/or manipulation of each original observation. While this is the core of the program, it is often surprisingly short.
4. Final manipulation of the data, and output of a new observation at the end of each experimental unit.

To implement these ideas, we need to use `by` statements and `first.` and `last.` variables.

Reshaping Data Sets: Example 2

Consider the transformed data set from the previous example. Suppose we wish to put it back to it's original form:

```
data next;
   set new;
   by id;

   array xx x1-x4;
   retain xx;

   if first.id then do i=1 to 4;xx{i} = .;end;

   xx{time} = x;

   if last.id then output;
   drop x i;
run;
```

Multi-Dimensional Arrays

By providing multiple subscripts in the array statement, you can create multidimensional arrays. The number of variables in the array is the product of the ranges of their subscripts, and the variables in the array are stored by rows.

If you declare a multi-dimensional array without specifying a list of variables (or the `_temporary_` keyword), SAS will create variables with the array name followed by a single number ranging from 1 to the total number of elements in the array.

To use the `dim`, `hbound`, or `lbound` functions with multi-dimensional arrays, use a second argument specifying the dimension you are interested in. (like `dim(x,2)`). (You can also use a function of the form `dim`*n*().)

Multi-Dimensional Arrays - Example

Suppose we are testing different lakes for the presence or absence of a variety of organisms, recorded as a zero/one variable. For each lake, we test for 10 organisms, at each of 5 times. We need a data set which has, for each lake, one observation for each organism which was ever present in the lake, as well as the first time it was observed, and the last time it was observed. Thus, a set of observations for a single lake might look like this:

```
ID TIME  O1  O2  O3  O4  O5  O6  O7  O8  O9  O10
 1   1    0   0   0   0   1   1   0   0   0    0
 1   2    1   0   0   0   1   0   1   1   0    0
 1   3    0   1   1   1   0   1   0   0   0    0
 1   4    1   0   0   0   1   1   1   0   0    0
 1   5    0   1   1   1   0   0   0   1   1    0
```

Multi-Dimensional Arrays - Example(cont'd)

We begin by declaring the necessary arrays, and initializing them each time a new id is encountered:

```
data two;
  set lakes;
  by id;
  array all{5,10} _temporary_;
  array firsts{10} _temporary_;
  array lasts{10}  _temporary_;
  array orgs{10} o1-o10;

  if first.id then do;
     do i=1 to 10;
        firsts{i} = 0;
        lasts{i} = 0;
        do j=1 to 5;
           all{j,i} = 0;
           end;
        end;
     end;
```

Multi-Dimensional Arrays - Example(cont'd)

The body of the program is especially simple - we just copy the data from the multiple observations into the doubly-dimensioned array:

```
  do i=1 to 10;
     all{time,i} = orgs{i};
     end;
```

Now we check to see which organisms were found for this observation, by storing the first and last times that the organism was seen.

```
  if last.id then do;
     do i=1 to 10;
        do j=1 to 5;
           if all{j,i} = 1 then do;
              if firsts{i} = 0 then firsts{i} = j;
              lasts{i} = j;
              end;
           end;
        end;
```

Multi-Dimensional Arrays - Example(cont'd)

Finally we check the `firsts` array – each time an element is not zero, it means that the organism was seen in this lake, so we output an observation.

```
     do i=1 to 10;
        if firsts{i} ^= 0 then do;
           org = i;
           first = firsts{i};
           last = lasts{i};
           output;
           end;
        end;
     end;    * - this ends the last.id loop;

  keep id org first last;
run;
```