ABSTRACT

Arrays are powerful tools, but may prove intimidating to new users. This hands-on workshop presents the basics of array statements and provides insight into their usage. Arrays are SAS® data step statements that allow clever programmers to do a great deal of work with little code. Arrays process identified variables as a group rather than individually, therefore saving both machine and programmer time. This workshop will walk the attendees through basic array programming.

INTRODUCTION

Most programmers can remember learning a few tricks that have made their lives simpler and enabled them to program faster and more effectively. For many, arrays can be one of those tricks. Array programming can allow SAS to “step through” a large number of variables, while performing the same operations on each one. This allows for a huge savings in the number of lines of code to be used, because logical operations only need to be coded once. The benefits of this decrease in code are obvious: faster programming, easier debugging, and straightforward interpretation (for experienced programmers). This paper will attempt to explain this powerful programming tool with syntax and programming examples.

BASIC SYNTAX

To begin, basic array syntax will be presented. This must be contained within a SAS data step in order to be valid. The format for an array statement is:

```
array array_name (number_of_elements) $ array_elements;
```

where:

- **array_name** is the name of the array, which will be referenced throughout the rest of the data step.
- **number_of_elements** is an integer representing the number of variables (or elements) in the array. This number can be contained within ( ), [ ], or { }.
- $ is an optional element which, if included, designates a character array. Array elements can be either all character or all numeric, but not a mix of both. If a length for the elements in this array is needed, it can be included after the dollar sign.
- 2 -

- **array_elements** are the names of the variables that will be contained within the array. The array_name cannot be the same as any SAS variables contained in the current dataset. It is helpful to assign a logical name to the array, just like other variables. This will allow for easier reference throughout the rest of the data step. Some programmers begin the name of arrays with an underscore; however this is not required and is up to the preference of the coder.

The number_of_elements is usually a positive integer value (or values), but may contain a numeric variable, equation, or an asterisk (*). The asterisk acts as a “wild card”, and SAS will create the array with as many dimensions as needed in order to accommodate the array_elements. When an integer value is used, there must be exactly that number of elements in the array. These elements will then be referenced sequentially up to that number, starting with one. In this case, the first element in the array will be referenced as array_name(1). If a different span is desired, the programmer can specify both a start and end value. For example, if an array is created with 0:9 as the number_of_elements, then the first element of the array will be referenced as array_name(0).

The array_elements are the names of the variables to be included in the array. If no array_elements are specified, then SAS will create default variables to fill the array. It is important to note that arrays only point to variables, they do not replace them.

**EXAMPLE 1: Simple Numeric Array**

This will show a group of numeric variables being processed both with and without the use of arrays. Imagine, for example, that the programmer is working with a financial dataset that was received from another division. In this dataset, some values of variables have been set to missing, while now it has been decided that they should contain the value zero. First, a data step without using an array:

```sas
data noarray1;
set basefile1;
   if value1 = . then value1 = 0;
   if value2 = . then value2 = 0;
   if value3 = . then value3 = 0;
   if value4 = . then value4 = 0;
   if value5 = . then value5 = 0;
   if value6 = . then value6 = 0;
   if value7 = . then value7 = 0;
   if value8 = . then value8 = 0;
   if value9 = . then value9 = 0;
run;
```

where:
• `noarray1` is the name of the new dataset.
• `basefile1` is the name of the original dataset.
• `value1 – value9` are the variables which need to be changed.

This data step accomplishes the objective with relative ease. It is straightforward and easy to interpret. Imagine, however, that instead of 9 variables, there were hundreds! Coding would become time consuming and repetitive, and any changes would be difficult to promote. Now, the same logic will be preformed using an array:

```
data array1;
  set basefile1;
  array vals(9) value1 - value9;
  do i = 1 to 9;
    if vals(i) = . then vals(i) = 0;
  end;
  drop i;
run;
```

where:
• `array1` is the name of the new dataset.
• `basefile1` is the name of the original dataset.
• `value1 – value9` are the variables which need to be changed, and also the `array_elements`.
• `vals` is the `array_name`.
• `9` is the `number_of_elements`.

In this data step, an array is used to “hold” the variables that need to be changed if necessary. For clarification, let us look at each line of code after the set statement individually:

• `array vals(9) value1 - value9;`
  o This line creates the array `vals` with 9 elements, which are `value1` through `value9`.
• `do i = 1 to 9;`
  o A `do` loop will allow SAS to step through each element in the array. In this example, the loop will repeat 9 times, so there will be a value of `i` for each of the elements in the array.
• `if vals(i) = . then vals(i) = 0;`
  o The `if` statement is built to perform the same logic for each iteration of the `do` loop, using `i` to reference each element of the array. For example, when `i = 1`, the statement becomes `if vals(1) = . then vals(1) = 0`, which is equivalent to `if value1 = . then value1 = 0`. This then repeats for each element of the array.
• `end;`
  o This `end`s the `do` loop.
• `drop i;`
At the end, the index for the do loop can be dropped. This step is at the discretion of the programmer.

It is easy to see why programming with arrays is often preferred over sequential statements, especially if there are many variables!

EXAMPLE 2: More Complex Numeric Array

In this example, numeric array will be taken a step further. Again, a group of numeric variables will be processed both with and without the use of arrays. In this case, however, the variables have different names. Also, multiple arrays will be used in order to match up different, but related, variables. For this example, the programmer has three distinct groups of variables: one containing pretax prices, one containing tax rates, and a final group to create containing post tax prices. Each price varies by state, as well as the tax rate. First, a data step without using an array:

```
data noarray2;
  set basefile2;
  al_posttax = al_pretax * al_taxrate;
  ak_posttax = ak_pretax * ak_taxrate;
  as_posttax = as_pretax * as_taxrate;
  ar_posttax = ar_pretax * ar_taxrate;
  ca_posttax = ca_pretax * ca_taxrate;
  co_posttax = co_pretax * co_taxrate;
  ct_posttax = ct_pretax * ct_taxrate;
  de_posttax = de_pretax * de_taxrate;
  dc_posttax = dc_pretax * dc_taxrate;
  fl_posttax = fl_pretax * fl_taxrate;
  ga_posttax = ga_pretax * ga_taxrate;
run;
```

where:

- `noarray2` is the name of the new dataset.
- `basefile2` is the name of the original dataset.
- `[state]_posttax` are the variables which need to be created containing the post tax price.
- `[state]_pretax` are the variables containing the pretax price.
- `[state]_taxrate` are the variables containing the tax rate.

While this data step creates the variables as needed, it would become quite unwieldy if data for all 50 states is needed! Next, arrays will be used to simplify the coding. Remember, in this example the values are dependent on the state, so care must be taken in building the arrays. The variables must “line up” in the arrays, so identical states have
the same reference number throughout (this is also called “position dependence”). Here is a data step using arrays:

```plaintext
data array2;
set basefile2;
array posttax(*) ak_posttax -- ga_posttax;
array pretax(*) ak_pretax -- ga_pretax;
array taxrate(*) ak_taxrate -- ga_taxrate;
do i = 1 to dim(posttax);
  posttax(i) = pretax(i) * taxrate(i);
end;
drop i;
run;
```

where:
- `array2` is the name of the new dataset.
- `basefile2` is the name of the original dataset.
- `posttax(*)`, `pretax(*)`, and `taxrate(*)` are the arrays containing the post tax price, pretax price, and tax rate, respectively.
- `dim(posttax)` returns the dimensions of the array.

Again, this data step utilizes arrays in order to facilitate processing. There are a few new concepts used in this example. First, the `number_of_elements` in the array is specified by an asterisk. This will allow the array to take as many `array_elements` as needed. These array elements are specified with a double dash, meaning that the variables will be included as they are ordered in the data vector. It is important to check these arrays after they are built in order to make sure that the variables match up among the three arrays. Finally, the `dim( )` operator is used in order to determine the number of elements in the array. This is often used in conjunction with an asterisk, so that the exact number of elements does not need to be known. `lbound( )` and `hbound( )` could also be used in this situation, where `lbound` returns the first dimension of the array (one in this case), and `hbound` returns the last.

**EXAMPLE 3: A Character Array**

Up to this point, only arrays that contain numeric variables have been discussed. But there are also many opportunities for character variables to be included in arrays as well. Imagine a group of variables that contain five diagnostic codes for a specific doctor visit. The programmer would like to see if a patient was diagnosed with any level of mental retardation. This code could fall in any of the diagnostic fields, so all must be checked. A data step without an array would look like the following:

```plaintext
data noarray3;
set basefile3;
```

length mr_LVL $20;
if diagcd_1 in ('317','318.0','318.1','318.2') then
    mr_LVL = 'Handicapped';
if diagcd_2 in ('317','318.0','318.1','318.2') then
    mr_LVL = 'Handicapped';
if diagcd_3 in ('317','318.0','318.1','318.2') then
    mr_LVL = 'Handicapped';
if diagcd_4 in ('317','318.0','318.1','318.2') then
    mr_LVL = 'Handicapped';
if diagcd_5 in ('317','318.0','318.1','318.2') then
    mr_LVL = 'Handicapped';
run;

where:
- noarray3 is the name of the new dataset.
- basefile3 is the name of the original dataset.
- diagcd_# are the variables containing the diagnostic code for each patient visit.
- mr_LVL contains the diagnostic code and level of mental retardation if available.

With only five codes that need to be checked, this may be an acceptable way of performing the desired logic. The main difficulty in this example would be debugging if a problem occurs, or updating the logic if a new code is added. These problems could be solved with the use of an array:

```sas
data array3;
  set basefile3;
  length mr_LVL $20;
  array diags(5) diagcd_1 - diagcd_5;
  mr_LVL = 'None';
  do i = 1 to 5;
    if diags(i) in ('317','318.0','318.1','318.2') then
      mr_LVL = 'Handicapped';
  end;
run;
```

where:
- array3 is the name of the new dataset.
- basefile3 is the name of the original dataset.
- diags(5) is the array containing the diagnostic code variables for each patient.
- mr_LVL contains the diagnostic code and level of mental retardation if available.
This data step uses character arrays in the same way that numeric arrays have been used in the previous examples. The diagnostic codes are stepped through, checking for a value that is either '317', '318.0', '318.1', or '318.2'. If any of these are found, then mr_lvl is set to ‘Handicapped’. Otherwise, it contains ‘None’. Any changes that need to be made to the logic only need to be made once, because the array will allow the same statement to be performed on each array element.

EXAMPLE 4: Inputting with an Array

Arrays are not limited to performing logic operations, but can be used in other situations as well. In this case, a comma delimited file has been given to the programmer, and a SAS dataset needs to be created from it:

```sas
data noarray4;
  infile 'C:\basefile4.csv' dsd missover;
  input trans_num $17.
    diagcd1 $10.
    diagcd2 $10.
    diagcd3 $10.
    diagcd4 $10.
    diagcd5 $10.;
run;
```

where:
- **noarray4** is the name of the new dataset.
- **basefile4.csv** is the name of the original comma delimited data file.
- **diagcd_#** are the variables containing the diagnostic code for each patient visit.

The input can also be accomplished using an array:

```sas
data array4;
  infile 'C:\basefile4.csv' dsd missover;
  array diagcd(5);
  input trans_num :$17.
    diagcd(*) :$10.;
run;
```

where:
- **array4** is the name of the new dataset.
- **basefile4.csv** is the name of the original comma delimited data file.
- **diagcd(5)** is the array used to create and contain the diagnostic code variables for each patient.
As mentioned before, because this array does not have any array_elements, variables will be created in the form of the array_name followed by a sequential number. So the dataset array4 will have exactly the same variables as noarray4.

EXAMPLE 5: Summarizing with Arrays

As a final example, an array will be used in conjunction with other procedures in order to alter to structure of a dataset. When combined with the sort and summary procedures, a programmer can easily and quickly convert a “long” dataset (multiple entries per unique identifier) to a “wide” dataset (one entry per unique identifier with chronological variables across). First, without an array:

```latex
data noarray5;
set basefile5;
  if year(pay_date) = 2000 then sales2000=pay_amt;
  else if year(pay_date) = 2001 then sales2001=pay_amt;
  else if year(pay_date) = 2002 then sales2002=pay_amt;
  else if year(pay_date) = 2003 then sales2003=pay_amt;
  else if year(pay_date) = 2004 then sales2004=pay_amt;
  else if year(pay_date) = 2005 then sales2005=pay_amt;
run;

proc sort data=array5;
by store;
runch;

proc summary data=array5;
by store;
var sales2000 – sales2005;
output out=sum5 (drop=_TYPE_ _FREQ_) sum=;
run;
```

where:
- **noarray5** is the name of the new dataset.
- **basefile5** is the name of the original dataset.
- **sales#** are the programmer created yearly sales variables.
- **pay_date** is the date of payment.
- **year(pay_date)** is the year of the date of payment.
- **pay_amt** is the payment amount.
- **store** is the unique store identifier.

In this case, basefile5 has payment, payment date, and store identifier listed chronologically across multiple years. But instead of each record being one payment, the
The programmer would like a record for each unique store, with yearly sales totals as the variables. This can be accomplished by inserting each payment into the corresponding sales variable based on the year that the payment was received. Next, the same objective will be accomplished using an array:

```
data array5;
  set basefile5;
    array sales(2000:2005);
    sales(year(pay_date)) = pay_amt;
  run;

  proc sort data=array5;
    by store;
  run;

  proc summary data=array5;
    by store;
    var sales2000 - sales2005;
    output out=sum5 (drop=_TYPE_ _FREQ_) sum=;
  run;
```

where:

- **array5** is the name of the new dataset.
- **basefile5** is the name of the original dataset.
- **sales(2000:2005)** is the array used to create and contain the yearly sales variables.
- **pay_date** is the date of payment.
- **year(pay_date)** is the year of the date of payment.
- **pay_amt** is the payment amount.
- **store** is the unique store identifier.

The data step above is deceivingly simple. Instead of using a do loop in order to pick an array_element, the year of the payment is calculated as the dimension of the array, and the corresponding element is assigned. With this logic, the number of years contained in a data file can be updated with relative ease.

**CONCLUSION**

Arrays are a wonderful programming tool when implemented correctly. While learning to use arrays, it may be helpful to work through a program without them first. Then, the programmer can go back and insert arrays where appropriate, and check to make sure the results did not change. Once the programmer is comfortable, though, using arrays will become second nature. Remember to always have a SAS reference guide handy, and good luck!
CONTACT INFORMATION

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

Ian Stockwell
Center for Health Program Development and Management,
University of Maryland, Baltimore County.
istockwell@chpdm.umbc.edu

Marge Scerbo
Center for Health Program Development and Management,
University of Maryland, Baltimore County.
scerbo@cpdm.umbc.edu

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.