Paper 52-26

Array tutorial(2) $ beginning intermediate;

Marge Scerbo, CHPDM/UMBC

Abstract

This tutorial will present the basics of array statements and show easy to intermediate examples of usage. Arrays are SAS data step statements that allow clever programmers to do a great deal of work with little code. These statements provide a method to process the identified variables as a group rather than individually, therefore saving both machine and programmer time. This paper should help lower the attendee's anxiety level when using arrays.

Introduction

Some programmers actually like writing lines and lines of simple code. As long as the project is completed and the results are within the expected time frame, this is usually acceptable. Other programmers hate to type and find any method possible to write as little code as possible, but similarly want to get the job done. In either case, code may be easy or hard to review. A very long monotonous program may be difficult to debug simply because of volume; fix one mistake and another one is discovered. A very terse program with complicated algorithms may also be difficult to interpret by either the original or subsequent programmer.

SAS code allows programmers to write either way, long or short, easy or complicated. It is easy to get to the same result from different directions. Data step programming is the core to SAS code. Within a data step it is possible to accomplish many different tasks. This paper will add one new technique, the array statement, to a programmer's tool belt. In addition to an overview of the array statement, several basic and intermediate examples of array usage will be presented.

This paper will cover only:

- One-dimensional arrays
- Explicit arrays - SAS Language guide recommends use of explicit arrays rather than implicit arrays

Long Way Around – Example #1

The best way to show how arrays are useful is by demonstrating two different methodologies. The first one uses data step code without arrays and the second method includes array processing.

Make the assumption that a data set has been received from another group. Someone in that department decided that all missing values should be set to 0 so that simple assignment statements does not cause a warning if a missing value was encountered. Unfortunately, these 0's can cause inaccurate results in other studies, so the process had to be reversed.

Here is an example of a program that uses repetitive simple statements to reset these fields to missing.

```
data nonarray1;
        set basefile;
        If tot1  = 0  then tot1  = . ;
        If tot2  = 0  then tot2  = . ;
        If tot3  = 0  then tot3  = . ;
        If tot4  = 0  then tot4  = . ;
        If tot5  = 0  then tot5  = . ;
        If tot6  = 0  then tot6  = . ;
        If tot7  = 0  then tot7  = . ;
        If tot8  = 0  then tot8  = . ;
        If tot9  = 0  then tot9  = . ;
        If tot10 = 0 then tot10 = . ;
        If tot11 = 0 then tot11 = . ;
        If tot12 = 0 then tot12 = . ;
        If tot13 = 0 then tot13 = . ;
        if tot14 = 0 then tot14 = . ;
run;
```

This code is acceptable and readable, but imagine what this code would look like if there were 400 different variables!

Basic Array Syntax

An array statement must 'exist' in a SAS data step. It does not function within a Procedure. The basic format of an array is:

Array **array-name(number-of-elements) array-elements;**

The pieces of the ***array*** puzzle include:

- An **array-name** identifies the group of variables in the array.
- The **number-of-elements** shows the number of elements (variables) in a one-dimensional array.
- The **array-elements** are the elements or fields that make up the array.

The **array-name** cannot be the name of a SAS variable used in that data step. In other words, it cannot be the name of any variable in the data set(s) read in or output. Although there are no errors encountered if the array-name is the name of a SAS function, it is dangerous to use a function-name as the array-name. The results may not be valid.

The **number-of-elements** can be either a number or numbers, a calculation, a numeric variable, or an asterisk '*'. Again, the **number-of-elements** designates how many elements exist in an array. If the number of elements is unknown, using the asterisk, '*', allows SAS to count the number of elements. This subscript is enclosed in parentheses or brackets.

Arrays can be either numeric or character but not a combination of types. If the array is character, the subscript is followed by a dollar sign '$'. The subscript and, if needed, the dollar sign designating a character array can be followed by a number that assigns the length of the elements.

Array-elements are a list of the variable names. Again, these must be all character or all numeric fields. These are variables that are referred to as **array-name**(subscript number) during processing. If no **array-elements** are named, SAS creates new variables that are named array-name with the subscript number concatenated to the end.

Arrays themselves are not data in a SAS data set.

The **array-name** and all the **array-elements** must be valid SAS names. In Version 6, this means that the name can be between 1 and 8 characters long beginning with a letter (A-Z) or an underscore (_). These names cannot contain a blank or any special character except the underscore. Finally, names cannot be SAS reserved words. Version 8 allows names to be between 1 and 32 characters in length with all the other rules still enforced.

Explicit versus Implicit arrays

Briefly, there are two types of arrays, explicit and implicit. The major difference between these two array types is that explicit arrays specify the number of elements in the array.

- **Implicit arrays** were used in earlier versions of SAS; the elements of this array type are referenced by evaluating the current value of the index variable associated with the array.
- **Explicit arrays** are considered more powerful and are much more straightforward. As stated in the introduction of the paper, only explicit arrays are discussed here.

Shorter way around - Example #1

The first example is a perfect beginning to show the implementation of an array statement. Remember, this process is checking the values of variables *tot1* through *tot14* and changing each value of 0 to a missing value:

```
data array1;
   set basefile;

   array tots(14) tot1-tot14;

   do i = 1 to 14;
        if tots(i) = 0 then tots(i) = . ;
   end;
   drop i;
run;
```

To further identify the array parts:

- the **array-name** is *tots*
- the **number-of-elements** is 14
- the **array-elements** are *tot1* through *tot14*.

Note that the number of fields (*tot1* through *tot14*) equals the subscript (14). The **do loop** is executed 14 times and the **index-variable** i is incremented from 1 to 14. This causes the value of tot1, tot2, etc. to be checked to see if each is equal to 0. Every time this condition is true, the **array-element** value is reset to missing. Before exiting the data step, the **index-variable**, i, is dropped, as it does not need to be stored in the data set.

To step through the first increment of the **do loop**:

- the **index-variable** i is set to 1.
- **tots(1)** is resolved to the value of **tot1**
- the condition **if tot1 = 0** is tested
- if this condition is true, then **tot1 = .**
- the **end** of the **do loop** is encountered and the process begins again with the **index-variable** incremented to 2

If the situation were such that there were 400 elements involved, creation of an array statement containing that many fields would be no more difficult:

```
data arraybig1;
   set bigbasefile;

   array tots(400) tot1-tot400;

   do i = 1 to 400;
      if tots(i) = 0 then tots(i) = . ;
   end;
   drop i;
run;
```

Changing the array code to add or subtract **array-elements** is easy and can make the SAS code more flexible than the earlier, non-array, versions. Imagine adding 376 more if statements!

Long Way Around – Example #2

The last example involved variables with the same base name, tot. It is very probable that some projects require manipulation of a list of fields with a variety of names.

In this next case, there is a need to recalculate a list of charges to include a cost-of-living increment if the amount is greater than 0. There are 7 fields, and the non-array code would be:

```
data nonarray2
      set basefile2;

      if basepay gt 0
        then basepay = basepay * 1.1345;
      if copay gt 0 then
        copay = copay *1.1345;
      if fedpay gt 0 then
        fedpay = fedpay *1.1345;
      if insurepay gt 0 then
        insurepay = insurepay * 1.1345;
      if deductible gt 0 then
        deductible = deductible * 1.1345;
      if savepay gt 0 then
        savepay = savepay * 1.1345;
      if pretaxpay gt 0 then
        pretaxpay = pretaxpay * 1.1345;

run;
```

Of course, this does work. If new fields are to be included, then more lines need to be added, and so on.

Shorter way around - Example #2

The last example can be written in array code. In the example below, charge(1) equals *basepay*, charge(2) equals *copay*, and so on.

```
data array2;
     set basefile3;

     array charge(7) basepay copay fedpay insurepay
                        deductible savepay pretaxpay;

       do i = 1 to 7;
             if charge(i) gt 0 then
                  charge(i) =  charge(i) * 1.1345;
       end;
       drop i;
run;
```

If new fields were added to this process, they would be added to the list of **array-elements** and the **subscript** and **do loop** counter must be changed.

A Short Character Example

So far, the examples have shown arrays used for numeric fields, but arrays are equally useful in character manipulation. Here is another example of non-array vs. array code. The specification of this analysis includes:

- This hospital file contains 6 surgical procedures, any number of which may be set to missing, a value of blank.
- The outcome of the program is a count of how many procedures are complete in each record.
- In each record, a new field is created called *surgcnt* that contains the number of valid values.

The non-array code could be written as:

```
data nonarray3;
      set newproc;
      surgcnt = 0;

      if procs1 ^= '  '
            then surgcnt = surgcnt + 1;
      if procs2 ^= '  '
            then surgcnt = surgcnt + 1;
      if procs3 ^= '  '
            then surgcnt = surgcnt + 1;
      if procs4 ^= '  '
            then surgcnt = surgcnt + 1;
      if procs5 ^= '  '
            then surgcnt = surgcnt + 1;
      if procs6 ^= '  '
            then surgcnt = surgcnt + 1;

run;
```

The array code can be demonstrated as:

```
data array3;
      set newproc;
      surgcnt = 0;
      array procs (6) $5 procs1-procs6;

      do i = 1 to 6;
             if procs(i) ^= ' ' then surgcnt = surgcnt + 1;
      end;
     drop i;
run;
```

Again, the code is similar to the numeric example. Since there are only 6 procedures involved, there are no spectacular differences between the non-array and array examples. If more codes were added, the non-array code would get longer and longer, and errors could easily appear. It is quite easy to copy the line of code over and over, but remember that the number attached to the variable name must be changed, and it is simple to miss one number or repeat a number.

An Even Better Character Example

Here is another example of non-array versus array code. The specifications for this process include:

- This hospital file includes several repeating fields. Two of these fields are billing codes (*code1-code50*) and units of service (*unit1-unit50*).
- The output of this run is to count the number of regular room and board days for each observation in the file. This new field is called *rbunits*.
- Room and board billing codes are 150, 151, 152, 153, and 160.
- The variables *code1-code50* are character and *unit1-unit50* are numeric.

The non-array code is quite long. Note that this example only contains 5 iterations of the process, not the full 50:

```
data nonarray3;
     set hospital;

     rbunits = 0;
     if code1 in('150','151','152','153','160') then do;
               rbunits = sum(rbunits, unit1);
     if code2 in('150','151','152','153','160') then do;
               rbunits = sum(rbunits, unit2);
     if code3 in('150','151','152','153','160') then do;
               rbunits = sum(rbunits, unit3);
     if code4 in('150','151','152','153','160') then do;
               rbunits = sum(rbunits, unit4);
     if code5 in('150','151','152','153','160') then do;
               rbunits = sum(rbunits, unit5);

     *there would be 45 more of these if statements!
run;
```

The array code is much shorter (and is complete):

```
data array4;
     set hospital;

     rbunits = 0;
     array codes(50) $ code1-code50;
     array units(50) unit1-unit50;

     do i = 1 to 50;
        if codes(i) in('150','151','152','153','160') then
              rbunits = sum(rbunits, units(i));
     end;
     drop i;
run;
```

As these examples get more complicated, the efficiency of array programming becomes more evident!

Simple Input Example

Arrays can be useful during the creation of a data set. There may be instances when the file layout contains fields that would be read in the same order and with the same specifications, except further along the line of data.

These examples use input pointer control. A **pointer-control** input statement includes an at sign '@' followed by the column specification, the variable name and the informat of the field. This column specification can be a number, a calculation, or a numeric variable.

In this example, the hospital file data layout defines that one record may have up to 6 surgical procedures. (This file is used in an earlier example.) Again, it would be possible to write code to define each column, each variable, and each format. A non-array code example of reading these surgical procedures, which are character values 5 digits long, would be:

```
data surgproc;
        infile 'hospital.dat' lrecl = 568 missover;
        input
          @1    recipid    $11.
          @12   servdate   mmddyy10.
➔         @400 surgpr1     $5.
          @405 surgpr2     $5.
          @415 surgpr3     $5.
          @415 surgpr4     $5.
          @420 surgpr5     $5.
          @425 surgpr6     $5.  ;
run;
```

Note that this group of 6 procedures begins in ➔column 400, and each procedure is 5-digits long.

The same code can be written with an array statement. As this example begins, it is important to review an important option in an input statement. If a line is being read and processes need to occur in the middle of the process, a **trailing at sign** '@' holds the line until released, either programmatically or by reaching the end of the data step.

Before presenting the array code, note how this trailing at sign '@' is used in the example:

```
data medrecs.hospital;
        infile 'hospital.dat' lrecl = 5421 missover;
              *trailing at sign holds this line;
➔             input    @19     clmstatus        $ 1.  @;

              *keep only records that were paid;
              if clmstatus eq 'P' then do;
                input  @1      invnum           $ 17.
                       @18     acctcode         $ 1.
                       @20     clmtype          $ 1.
                       @131    provnum          $ 9.
                       @140    category         $ 2.
                       … ;
              end;
run;
```

On the line identified with the arrow ➔, note that only one field is read. This field designates the record as a paid claim or not. The final database contains only paid claims, and the steps to accomplish this are:

- The claim status variable, *clmstat*, is read.
- The pointer remains on that record or line. This is indicated by the at sign '@' as the last character before the semicolon.
- The status field is tested to see if the value equals 'P'.
- If this condition is true, the rest of the record is read.
- If the condition is false, the processing continues until the bottom of the data step, at which point the processing begins again and a new record is read.

This background is important to understand the next piece of code. The new code using an array to complete the task above follows:

```
data surgproc;
          infile 'hospital.dat' lrecl = 568 missover;
          input
             @1        recipid              $11.
             @12       servdate              mmddyy10. @ ;

           array surgpr(6) $5;
           input @400 surgpr(*) $5. ;
run;
```

So, to parse the various pieces of code in this example:

- A portion of the record is read, including the fields, *recipid* and *servdate*. The trailing at sign causes the pointer to remain on this record.
- A **character** ($) **array** *procs* is built with 6 elements, corresponding to the 6 surgical procedures.
- The **array-elements** are automatically defined as variables *surgpr1* through *surgpr6*. As shown, if no specific array elements are listed, the elements are named *arrayname1* through *arrayname6*.
- **Input** begins again in column 400 (**@400**), and each of the surgical codes is read in sequence (**procs(*)**) and each as a character field five characters long (**$5.**)
- If additional fields are to be read following the procedures, the variable definitions can be added to the **input** statement.

This example clearly demonstrates how very little code can be used to perform this operation.

Simple Output Example

In preparing data for certain types of analysis, it is sometimes best to reconfigure the fields to be searched, assuming there are a group of variables in one record which may contain similar data. In other words, when searching for a set of values, at times it is easier to search down the data set rather than across the records. So the process makes a short fat data set a long thin one!

Take the example of the data set just created. In order to search for a group of surgical procedures, it is possible to recreate the data set as one long and narrow data set, containing one procedure per record and whatever other fields may be needed. Previous studies have shown that there may be missing values in some of the procedure fields; these values may be blank, '000', '0000', or '00000'. It is also known that once a missing value has been encountered, all succeeding procedures in that record is missing.

In this example, a program is written which reviews each of the 6 procedure fields. If a valid value is encountered, a record containing that procedure, an identifier, and a date is output to a new data set. If a missing value is encountered, the process should continue to the next record, and so on. Note that checking is required only if there is a valid code found in the previous procedure field. Therefore, if the second code is found to be missing, the selection process will stop.

Without using an array statement, the code might be written as:

```
data surgproc (keep = idnum surgdate surgproc);
       set hospital;
       length surgproc $6;

   if surgpr1 not in(' ','000','0000','00000') then do;
              surgproc =surgpr1;
              output;
      if surgpr2 not in(' ','000','0000','00000') then do;
              surgproc =surgpr2;
              output;
        if surgpr3 not in(' ','000','0000','00000') then do;
              surgproc =surgpr3;
              output;
          if surgpr4 not in(' ','000','0000','00000') then do;
                surgproc =surgpr4;
                output;
             if surgpr5 not in(' ','000','0000','00000') then do;
                 surgproc =surgpr5;
                 output;
              if surgpr6 not in(' ','000','0000','00000') then do;
                     surgproc =surgpr6;
                     output;
              end;
            end;
          end;
        end;
      end;
    end;
run;
```

Using an array the data step could be:

```
data surgproc (keep = idnum surgdate surgproc);
       set basefile4;
       length surgproc $6;
       array surg(6) $ surgpr1-surgpr6;

       do i = 1 to 6;
              if surg(i) not in(' ','000','0000','00000') then do;
                           surgproc =surg(i);
                           output;
              end;
              else leave;
       end;
       drop i;
run;
```

The **in** operator allows a list of values to be tested. In the above case, the value of the surgical procedure should not be **in** that list of missing values. If indeed a missing value is encountered, the **leave** command causes the **do loop** to end. *Thanks to Ron Cody for his introduction to the leave statement.

Reporting with Arrays

In almost any industry, the data are only as valuable as the reports that can be generated from it. Generating complicated reports is a time-consuming job. Arrays can be used in a variety of ways to help create both the summarized and/or subset data sets and the final reports.

In the example below, an array is used to create a string of procedure codes. Note that the data set accessed (*hospital*) is the same data set manipulated in the last example. This large file contains information on inpatient stays, including up to six surgical procedures in one record. The report to be created is to contain the recipient id number (*recipid*), the date of service (*servdate*), the number of procedures performed (*proccnt*), and a list of procedures (*surglist*) separated by commas in the format:

procedure1, procedure2, …, procedure6

The following code creates an output data set. In production mode, this step could be combined with the actual report creation.

```
data report (keep = recipid servdate proccnt surglist);
     set hospital;

     array surg(6) $5 surgpr1-surgpr6;
     *must set the length of the final procedure list!;
     length surglist $42;

     proccnt = 0;
     do i = 1 to 6;
          if surg(i) not in(' ','000','0000','00000') then do;
             proccnt + 1;
             if proccnt = 1 then surglist = surg(i);
          else surglist =  left(trim(surglist)) || ', ' || surg(i);
        end;
    end;
run;
```

This code uses the character functions **left** and **trim** and the concatenation operator **||** to create the list in the correct format. Within the **do loop**, these operations occur:

- The code counter (*proccnt*) is incremented each time a valid procedure is encountered.
- When the first valid procedure is encountered, the list of codes (*surglist*) is set to this procedure.
- Any additional valid codes are added to the list string, which is first left-justified (**left**), trimmed of trailing blanks (**trim**), and concatenated (**||**) with a comma (**,**) and a blank space.

The output data set (*report*) can now be used to create a complete report of procedures.

Arrays to Separate Observations into Multiple Data Sets

UB-92 inpatient data contain many fields, and many of those are repeating fields. It is a correct assumption that one inpatient stay may create up to 50 repeats of a set of values. The example below satisfies the need to produce reports identifying records as good or bad. If any of the 50 detail charge dates(*detdate1-detdate50*) are identified as greater than the date of discharge(*datedischarge*), these records are in error (*badrecords*) and should be separated from the correct observations (*goodrecords*). Therefore, a large file is split to create these reports.

```
data goodrecords badrecords;
    set inpatient01;
    array detdates(50) detdate1-detdate50;
    bad = 0;
    do i = 1 to 50;
            if detdate(i) gt datedischarge then do;
                        bad = 1;
                        output badrecords;
                        leave;
            end;
    end;
    if bad = 0  then output goodrecords;
    drop i;
run;
```

This process is an easy method to separate observations into desired data sets. If an array were not used, each of the 50 dates would have to have been checked individually.

Arrays to Eliminate Unwanted Observations

There are times when it may be necessary to eliminate observations from a specific study. These deletions may be due to very specific requirements of the analysis. One reason might be that there are too many missing values in the observation. Of course, code can be written to test each variable to determine if the value is missing or not and ascertain whether or not to include the observation. The example below contains a generic approach to this situation.

```
data complete;
    set maybeokay;
    misscount = 0;

    array  ch(*)   _character_ ;
    array  nm(*)  _numeric_ ;

    do i = 1 to dim(ch);
            if ch(i) = ' '  then misscount + 1;
    end;
    do j = 1 to dim(nm);
            if nm(j) = .   then misscount + 1;
    end;

    drop i j ;
     if misscount ge 10 then delete;
run;
```

Three new SAS code bits are included in this code:

- **_Character_** is a special SAS variable that identifies and includes only character variables in the process described.
- **_Numeric_** similarly identifies only numeric fields.
- **Dim()** allows SAS to count the number of elements within the array. In using this code, it allows the program to function across almost any data set without requiring the program to include an exact count of the array elements, be they character or numeric.

Dates and Arrays

The following example uses a SAS date function and an array to satisfy a request. Assume a data set contains 12 flags (*jan99-dec99*) that indicate whether or not a person has been enrolled during that month. The fields are set to either 1 (enrolled) or 0 (not enrolled).

The specifications of this study require that for each month where the person has both been enrolled and received a service, a record is output to the analytic data set. An intermediate data set (*newfile*) has already been created that merged the enrollment file to the service file. Below is code to accomplish output of a new data set:

```
data serviced;
     set newfile;

     array enroll(12) jan99-dec99;
     mon = month(servdate);
     if enroll(mon) = 1 then output;
run;
```

This code creates an **array** containing each of the 12 monthly flags. Note that this code utilizing an array does not include a **do loop**. Instead, a new variable, *mon*, is created by using the **month** date function. This function reads a SAS date and returns the number of the month. This new variable is then used as the subscript of the array. For example, in one observation:

- The service date (*servdate*) is 9/12/1999.
- The **month** function returns the value 9.
- The program resolves the **if** statement as:
  if sep99 = 1
- If the person was enrolled in September 1999 (the variable sep99 is set to 1), a new observation is output to the data set *serviced*.

What could have been a very complicated task was made easy through the use of an array!

Conclusion

With a little practice and common sense, arrays can become a standard tool in a programmer's tool belt. Follow these tips:

- First, always have a SAS Language Guide, either paper or online, available!
- In the process of learning how to use arrays, make sure to test the program with non-array code. Print out the Log and Output.
- Then rework the program to include array code and compare these results with the non-array code.

After a while, it will become second nature to use arrays. Once the learning curve is over, the usefulness will increase and soon there will be multiple arrays and do loops within do loops!

Thanks for Ron Cody, Mike Zdeb, Richard Allen, and Steve Hinnenkamp for sharing their code. These samples provided new ideas for array usage.

References

Leighton, Ralph, (1992), "Working with Arrays: Doing More with Less Code", in the Proceedings of the NorthEast SAS Users Group Conference, 129-139

Contact Information

For more information contact:

Marge Scerbo
CHPDM/UMBC
1000 Hilltop Circle
Social Science Room 309
Baltimore, MD 21250
Email: scerbo@chpdm.umbc.edu