MWSUG 2018 - Paper SP-062

Efficiency Programming with Macro Variable Arrays

Veronica Renauldo, QST Consultations, LTD, Allendale, MI

ABSTRACT

Macros in themselves boost productivity and cut down on user errors. However, most macros are not robust and serve only a few specific repetitive purposes. Just like arrays increase the efficiency of a DATA step, macro variable arrays increase the efficiency of a macro. Macro variable arrays allow the macro to function more autonomously than what is typical for macro processing and work in all SAS® platforms that support macro processing. Automating the process of determining the number of times a macro needs to be utilized for a task is just one of the several applications of macro variable arrays. There are numerous ways to create macro variable arrays such as %LET statements, PROC SQL, and CALL SYMPUT statements; each with their own user-friendly approach. Macro variable arrays employ the use of loops and logic to construct comprehensive macros allowing for a multitude of output types functioning within one macro call. Constructing dynamic macros will increase the capacity of a macro while dramatically decreasing the lines of code in each program. In conjunction with macro functions such as %SYSFUNC, %SCAN, and %STR, macro variable arrays allow the creator and user of a macro to be more flexible with their coding; ultimately leading to more productivity with less code alterations. Impress your boss, your friends, and yourself with macro code that almost writes itself.

INTRODUCTION

DATA step arrays increase the efficiency of programming by allowing programmers to apply several steps of logic to several variables of the same type with less code. Macro variables allow the user to symbolically substitute a value in a multitude of areas via a coding short cut. Arrays and macros make programming faster, more dynamic, and reduce human error. The combination of these two approaches, called macro variable arrays, fuses the symbolic substitution from macro variables with the ability to run several chunks of code on various data sets, variables, or the contents within a given variable. Macro variable arrays combine the ease of array processing but at a macro level. Background information on macro variables, macro functions, and macro variable resolutions will provide the basis for understanding macro variables arrays. Through examples of applications using macro variable arrays the three ways to create macro variables will be explored along with various uses and applicable macro functions.

MACRO VARIABLES

Macro variables are character stings that are used for symbolic substitutions of text within SAS code. The maximum length allowed in a macro variable is 65,534 characters. Macro variable calls are always prefixed with at least one & and compile, or end, in at least one period for the macro variable to resolve to the defined value. Macro variables have a constant value that are set in two different ways: automatically by SAS or user defined (SAS Institute Inc., 2009).

Automatic SAS macro variables are macro variables that are set when a SAS product is deployed. For example, the macro variable SYSDATE is the date at which a given SAS product was invoked in the DATE7 format. There are several automatic macro variables that are created when a SAS product is deployed. By typing %PUT _all_; into the SAS enhanced editor all macro variables, automatic and user-defined, will output to the log with the type of macro variable, macro variable name, and the value of the macro variable (SAS Institute Inc., 2009).

```
    %put _all_;
AUTOMATIC SYSDATE 03AUG18
AUTOMATIC SYSDATE9 03AUG2018
AUTOMATIC SYSDAY Friday
AUTOMATIC SYSDEVIC
AUTOMATIC SYSDMG 0
AUTOMATIC SYSDSN          _NULL_
AUTOMATIC SYSENCODING wlatin1
AUTOMATIC SYSENDIAN LITTLE
AUTOMATIC SYSENV FORE
AUTOMATIC SYSERR 0
AUTOMATIC SYSERRORTEXT
AUTOMATIC SYSFILRC 0
AUTOMATIC SYSINDEX 0
AUTOMATIC SYSINFO 0
AUTOMATIC SYSJOBID 10184
AUTOMATIC SYSLAST _NULL_
```

User-define macro variables are exactly as they are described: created by the user. User-define macro variables are text, or numeric values stored as text, that a programmer deems worthy of being substituted where needed instead of manually typed in their program each time the value is required (SAS Institute Inc., 2009). For example, if a particular footnote will be used for multiple figures that a program is creating that footnote might be placed in a macro variable *footnot1* such that the programmer doesn't have to type the footnote each time it is required but rather just call the macro variable *footnot1* in the footnote section of each of the figures. Macro variables can be created in one of three ways: %LET statement, CALL SYMPUT routine, or the SQL procedure. These three approaches to creating macro variables, and thus macro variable arrays, will be illustrated throughout the Macro Variable Arrays section.

MACRO FUNCTIONS

Before tackling how an arrays and macro variables come together to form macro variable arrays, knowledge of useful macro functions is vital to comprehending all of the aspects of more dynamic macro code. Below is a table of macro functions, their inputs, and a description of the function (SAS Institute Inc., 2009).

Table 1: SAS Macro Functions

Macro function and Inputs	Description
%STR()	%STR allows the user to "mask" any operator or special character in a macro variable so that the macro variable of interest is able to compile. The masking occurs at the time the macro variable is compiled. Below are examples of special characters and operators %STR is able to mask. Special Characters: & % ' " ( ) + - * / < > = ¬ ^ ~ ; , blank Operators: AND OR NOT EQ NE LE LT GE GT For example, if a title contains an apostrophe SAS would be unable to compile the title because of unmatched quotes so %STR can be used to mask the apostrophe so that the title can compile.
%SCAN(argument, n, <delimiters>)	Searches the argument and returns the *nth* word. A word is one or more characters separated by one or more delimiters. If no delimiter is specified then the function uses all delimiters possible to find identify a "words". This function does NOT mask special characters or operators in its result even when input argument has been masked by a quoting function. %SCAN works the same as the data set function SCAN. The following are default delimiters: blank . < (+ & ! $ *) ; ^ - / , % |. If you want to only use a single blank or a single comma as a delimiter then the character must be enclosed in %STR function ex. %STR() or %STR(,).
%BQUOTE(string)	Masks special characters and mnemonic operators in a resolved value at macro execution.

%SYSFUNC(function)	%SYSFUNC executes SAS DATA step functions or user written functions on macro variables. Almost all functions that can be used in a DATA step can be used on macro variables when using this macro function. There are a few select functions that cannot be used with %SYSFUNC: all variable information functions, DIF, IORCMSG, LBOUND, LEXPERK, PUT, ALLCOMB, DIM, INPUT, LEXCOMB, LEXPERM, RESOLVE, ALLPERM, HBOUND, LAG, LEXCOMBI, MISSING, INPUT, SYMGET
COUNTW(string, chars, modifiers) used within %SYSFUNC	Counts the number of words in a macro variable

MACRO VARIABLE RESOLUTION AND ARRAY OF MACRO VARIABLES

In order to compile a macro variable an ampersand (&), followed by the name of the macro variable, followed by a period is required to compile the macro variable and make the substitution (SAS Institute Inc., 2009). Several macro variables can share the same prefix with a counter or other suffix portion. Macro variables that are the same prefixes are known as an array of macro variables. An array of macro variables is an ordinal list of macro variables where each contains one element to be processed (Long & Heaton, 2008). The three macro variables below, *race1, race2*, and *race3* comprise an array of macro variables such that each macro variable starts with *race* and then has an iterative integer counter on the end. When the macro PRINT is executed it will first compile *i* macro variable to the loop iteration number, just like in a DATA step loop. Next, race in conjunction with the compiled *i* macro variable will compile to either white, asian, or other if the macro variable *i* is 1, 2, or 3, respectively.

```
%let race1 = white;
%let race2 = asian;
%let race3 = other;
```

To compile just one of the macro variables the macro variable itself would be called in the form of *&race1.* if the first race macro variable was desired.

```
15   %put &race1.;
white
```

Since these macro variables all have the same prefix, a macro that uses consecutive ampersands can be created in order to print the three macro variable values. The macro *print* can loop through the printing process of the three macro variables of interest using consecutive ampersands. Using the macro variable *i* as the iterative counter, the loop within the macro should go from the lowest to the highest counter in the array of macro variables. For this example there are three macro variables where the lowest and highest counter values are 1 and 3. By specifying two ampersands at the beginning of the call for the array of macro variables, the double ampersand instructs the macro processer to concatenate the prefix of the macro variables, *race*, with the resolution of the counter macro variable *i.* By placing the resolution periods at the end of the macro variable call the macro processer will first resolve the macro variable *i* to the loop iteration number, just like in a DATA step loop, and then resolve the macro variable *race* in conjunction with the resolved value of the macro variable *i* (race<*value of i*>) to successfully compile each of the macro variables (Werner, 2014). The first loop through the macro will result with a value of white, then asian, followed by other.

```
%macro print();
   %do i = 1 %to 3;
      %put &&race&i..;
   %end;
%mend;


%print();
```

The global options MRPINT, MLOGIC, and SYMBOLGEN will print the macro, logic execution, and macro variable resolution information to the log.

```
19   %macro print();
20      %do i = 1 %to 3;
21         %put &&race&i..;
22      %end;
23   %mend;
24
25   %print();
MLOGIC(PRINT):  Beginning execution.
MLOGIC(PRINT):  %DO loop beginning; index variable I; start value is 1; stop value is 3;
      by value is 1.
MLOGIC(PRINT):  %PUT &&race&i..
SYMBOLGEN:  && resolves to &.
SYMBOLGEN:  Macro variable I resolves to 1
SYMBOLGEN:  Macro variable RACE1 resolves to white
white
MLOGIC(PRINT):  %DO loop index variable I is now 2; loop will iterate again.
MLOGIC(PRINT):  %PUT &&race&i..
SYMBOLGEN:  && resolves to &.
SYMBOLGEN:  Macro variable I resolves to 2
SYMBOLGEN:  Macro variable RACE2 resolves to asian
asian
MLOGIC(PRINT):  %DO loop index variable I is now 3; loop will iterate again.
MLOGIC(PRINT):  %PUT &&race&i..
SYMBOLGEN:  && resolves to &.
SYMBOLGEN:  Macro variable I resolves to 3
SYMBOLGEN:  Macro variable RACE3 resolves to other
other
MLOGIC(PRINT):  %DO loop index variable I is now 4; loop will not iterate again.
MLOGIC(PRINT):  Ending execution.
```

The log output above illustrates how the symbolic resolution of macro variables works when more than one ampersand is utilized. If only one ampersand was utilized at the beginning of the macro variable call: &race&i.. the macro processor throws out a warning stating that the symbolic reference to the macro variable *race* could not be resolved because, up to this point, a macro variable *race* has not been defined. A double ampersand is required for the processer to call each macro variable in our array of macro variables correctly (Werner, 2014).

```
26   %macro print();
27      %do i = 1 %to 3;
28         %put &race&i..;
29      %end;
30   %mend;
31
32   %print();
MLOGIC(PRINT):  Beginning execution.
MLOGIC(PRINT):  %DO loop beginning; index variable I; start value is 1; stop value is 3;
      by value is 1.
MLOGIC(PRINT):  %PUT &race&i..
WARNING: Apparent symbolic reference RACE not resolved.
SYMBOLGEN:  Macro variable I resolves to 1
&race1.
MLOGIC(PRINT):  %DO loop index variable I is now 2; loop will iterate again.
MLOGIC(PRINT):  %PUT &race&i..
WARNING: Apparent symbolic reference RACE not resolved.
SYMBOLGEN:  Macro variable I resolves to 2
&race2.
MLOGIC(PRINT):  %DO loop index variable I is now 3; loop will iterate again.
MLOGIC(PRINT):  %PUT &race&i..
WARNING: Apparent symbolic reference RACE not resolved.
SYMBOLGEN:  Macro variable I resolves to 3
&race3.
MLOGIC(PRINT):  %DO loop index variable I is now 4; loop will not iterate again.
MLOGIC(PRINT):  Ending execution.
```

MACRO VARIABLE ARRAYS

A macro variable array is one singular macro variable that contains an entire array of elements to be processed (Long & Heaton, 2008). The macro variable can contain an array of data set names, data set variable names, or variable values that will be parsed out of the macro variable to use in whatever fashion the macro is created to accomplish. To follow the previous example, instead of having three macro variables, each containing one race value of interest, one macro variable *race* can be defined which contains the three race values of interest.

```
%let race = white asian other;
```

The %LET statement is one way of creating a macro variable array in which the user defies the elements in the array explicitly in their code. This macro variable array will not employ the use of consecutive ampersands to resolve to the desired values of interest but instead the %SCAN function can be utilized. The %SCAN function will extract the *i*th element of interest from the macro variable array *race*. Just like in the previous PRINT macro, the do loop will start at one and end when the counter is greater than 3. The first time the loop is executed SAS scans the macro variable array *race* for the first word. It should be noted that no delimiter is specified in the scan function so all possible default delimiters will be utilized. The first word, or element, in the macro variable array *race* is *white* which will be outputted to the log due to the %PUT function. The second and third times through the loop will result in *asian* and *other* being outputted to the log, respectively.

```
%macro print();
   %do i = 1 %to 3;
      %put %scan(&race., &i.);
   %end;
%mend;


%print();
```

The log below utilized the MRPINT, MLOGIC, and SYMBOLGEN options to show the macro, logic, and symbolic substitution of the macro variable array.

```
35   %macro print();
36      %do i = 1 %to 3;
37         %put %scan(&race., &i.);
38      %end;
39   %mend;
40
41   %print();
MLOGIC(PRINT):  Beginning execution.
MLOGIC(PRINT):  %DO loop beginning; index variable I; start value is 1; stop value is 3;
     by value is 1.
MLOGIC(PRINT):  %PUT %scan(&race., &i.)
SYMBOLGEN:  Macro variable RACE resolves to white asian other
SYMBOLGEN:  Macro variable I resolves to 1
white
MLOGIC(PRINT):  %DO loop index variable I is now 2; loop will iterate again.
MLOGIC(PRINT):  %PUT %scan(&race., &i.)
SYMBOLGEN:  Macro variable RACE resolves to white asian other
SYMBOLGEN:  Macro variable I resolves to 2
asian
MLOGIC(PRINT):  %DO loop index variable I is now 3; loop will iterate again.
MLOGIC(PRINT):  %PUT %scan(&race., &i.)
SYMBOLGEN:  Macro variable RACE resolves to white asian other
SYMBOLGEN:  Macro variable I resolves to 3
other
MLOGIC(PRINT):  %DO loop index variable I is now 4; loop will not iterate again.
MLOGIC(PRINT):  Ending execution.
```

The advantage of this method is in its efficiency. The first example of the PRINT macro required 9 lines of code while this method only required 7. Two lines of code may not seem like a lot but depending on the

construction and robustness of the user-defined macro this can end up saving a lot of space and programming time.

Throughout the following examples the three different ways of creating macro variable arrays will be illustrated along with increasing difficult macro code.

EXPLICITLY DEFINED MACRO VARIABLE ARRAY EXAMPLE

One way to create a macro variable array is to explicitly define it by means of a %LET statement or as a macro variable within a macro call. Refer to the Macro Variable Array section for an example of defining a macro variable array by means of a %LET statement. Depending on the specific task at hand, having a macro that reads in several permanent SAS data sets and outputs them to the work library for temporary use can be vital. The RIN macro below (read-in macro) will read in several data sets from the same library sorting each data set by the same variable(s) and outputting the sorted data set into the work folder with the data set prefix *r*. This macro contains three macro variables: LIB, VAR, and DAT. The macro variable LIB specifies the libname of interest, VAR specifies the sorting variable(s) of interest, and DAT specifies the data set(s) of interest. The macro variable DAT is technically a macro variable array since one or more data sets will be specified to run through the macro that are all in the same library and will be sorted by the same variable(s) specified in the LIB and VAR macro variables, respectively.

```
%macro rin(lib, var, dat);
   %do i = 1 %to %sysfunc(countw(&dat.));
      proc sort data=&lib..%scan(&dat., &i.)
                out=r%scan(&dat., &i.);
         by &var.;
      run;
   %end;

%mend;
```

This macro utilizes the %SYSFUNC and COUNTW functions to determine the end counter for the loop. The macro instructs SAS to count the number of words, or elements, in the macro variable array DAT to determine how many times the macro needs to be executed. This macro also utilizes the %SCAN function to parse out data set names from the macro variable array. As outlined above in Table 1: SAS Macro functions, the %SCAN function requires 3 parts: the item to be scanned through (&DAT.), the count of the word of interest in the character string (&i.), and the specification of any delimiters. Since no delimiter is specified in the %SCANs above, all default delimiters will be utilized. If the name of any data set of interest contained a default delimiter then a delimiter should be specified to ensure the %SCAN functions correctly.

For this example the SASHELP library will be utilized to sort and copy the data sets CLASS, CLASSFIT, FISH, GRIDDED, and HEART into the work directory sorting by the variable height (SAS Institute Inc., 2017).

```
%rin(sashelp, height, class classfit fish gridded heart);
```

The first macro variables LIB and VAR resolve to *sashelp* and *height*, respectively. Since the macro variable array DAT contains 5 elements the loop end counter will resolved to 5. Each of the data sets of interest will be executed through the macro which will result in 5 data sets in the work directory that all begin with the prefix *r*.

```
53   %rin(sashelp, height, class classfit fish gridded heart);
MLOGIC(RIN):  Beginning execution.
MLOGIC(RIN):  Parameter LIB has value sashelp
MLOGIC(RIN):  Parameter VAR has value height
MLOGIC(RIN):  Parameter DAT has value class classfit fish gridded heart
MLOGIC(RIN):  %DO loop beginning; index variable I; start value is 1; stop value is 5; by
      value is 1.
MPRINT(RIN):   proc sort data=sashelp.class out= rclass;
MPRINT(RIN):   by height;
MPRINT(RIN):   run;

NOTE: There were 19 observations read from the data set SASHELP.CLASS.
NOTE: The data set WORK.RCLASS has 19 observations and 5 variables.
NOTE: Compressing data set WORK.RCLASS increased size by 100.00 percent.
      Compressed is 2 pages; un-compressed would require 1 pages.
NOTE: PROCEDURE SORT used (Total process time):
      real time           0.02 seconds
      cpu time            0.01 seconds
```

…

```
MLOGIC(RIN):  %DO loop index variable I is now 5; loop will iterate again.
MPRINT(RIN):   proc sort data=sashelp.heart out= rheart;
MPRINT(RIN):   by height;
MPRINT(RIN):   run;

NOTE: There were 5209 observations read from the data set SASHELP.HEART.
NOTE: The data set WORK.RHEART has 5209 observations and 17 variables.
NOTE: Compressing data set WORK.RHEART decreased size by 31.19 percent.
      Compressed is 75 pages; un-compressed would require 109 pages.
NOTE: PROCEDURE SORT used (Total process time):
      real time           0.02 seconds
      cpu time            0.03 seconds


MLOGIC(RIN):  %DO loop index variable I is now 6; loop will not iterate again.
MLOGIC(RIN):  Ending execution.
```

Depending on how many data sets need to be pulled into the work library from permanent libraries this approach cuts down on programming code and subsequently programmer error. Macro variable arrays are employed in areas where efficiency and optimization in code are desired and user-error reduction can be promoted.

CREATING A MACRO VARIABLE ARRAY USING CALL SYMPUT

Macro variables, and thus macro variable arrays, can be created within a data set by employing the CALL SYMPUT routine. The CALL SYMPUT routine is a method of assigning a value to a macro variable within a DATA step. The CALL SYMPUT method can be employed in a regular DATA step or in a DATA NULL step (SAS Institute Inc., 2009). This example will use a macro variable array to obtain frequency data of origin and type for each manufacturer in the SASHELP data set CARS.

Using a DATA NULL step and by-group processing, the unique values of the variable *make,* which contains the car manufacturer name, will be concatenated into the variable *brand* and output in the macro variable array *brands*. The CARS SAS data set is already sorted by *make* and *type* so by-group processing can be utilized to capture the unique values of *make.* Below is the DATA NULL step along with a description of each component that is required to make the macro variable array *brands*.

```
data null_;
   ❶set sashelp.cars end = last;
   ❷by make;
   ❸length brand $5000;
   ❹retain brand;
   ❺if _n_ = 1 then brand = '';
   ❻if first.make then brand = strip(brand)||'*'||strip(make);
```

```
    ❼if last then do;
       brand = strip(substr(brand, 2));
       call symput('brands', strip(brand));
    end;
run;
```

❶The SET statement specifies which data set is going to be utilized for the DATA NULL step. Additionally, the *end* = *last* option creates a temporary binary variable *last* that has a value of 1 if the record is the last record in the data set otherwise *last* has a value of 0 (SAS Institute Inc., 2011). This will be utilized to output the macro variable array once the last record in the data set has been executed.

❷By-group processing of the variable *make* allows unique values of *make* to be determined for concatenation into the variable *brand* later on in the DATA step.

❸The LENGTH statement sets the length of the variable *brand* is set to 5,000 characters to ensure the variable is large enough to contain all unique values of the variable make. It is always good practice to specify a length for a retained variable unless the default character variable length of 8 is sufficient.

❹The RETAIN statement is utilized for the variable *brand* in order to carry the value of *brand* down the rows in the data set. The RETAIN statement carries the value of the retained variable of interest down the rows in the data set until the retained value is reassigned or changed. The variable *brand* needs to be retained down the data set while having each unique value of *make* concatenated into the variable.

❺The first observation in the data set *cars* initializes the value of the variable *brand* to missing.

❻Each unique value in the variable *make* is concatenated on to the variable *brand*, separated from the previous values contained in the variable *brand* by an asterisk. By-group processing is employed to only concatenate the distinct values of *make* onto *brand* if a new value of *make* is reached.

❼For the last observations in the data set *cars, brand* will be updated and output into the macro variable *brands*. Since each unique value of cars is concatenated onto the variable *brand* after an asterisk, the first unique value of *make* in the data set will be preceded by an asterisk. The asterisk is used as a delimiter to determine the number of words in the macro variable array. A delimiter other than a space had to be specified since one of the values of *make*, Land Rover, contains a space indicating that all default delimiters should not be used when employing the %SCAN function to parse out the macro variable array. Since having an asterisk prior to the first word is not useful the variable *brand* is sub stringed starting at the second position in order to remove the first asterisk. Next, the CALL SYMPUT routine is utilized to create the macro variable array *brands* which will contain the concatenation of all unique *makes* in the *cars* data set that are stored in the variable *brand*.

Below is the log output for the macro variable array *brands* that contains all distinct car manufactures from the data set SASHELP.CARS.

```
69   %put &brands.;
Acura*Audi*BMW*Buick*Cadillac*Chevrolet*Chrysler*Dodge*Ford*GMC*Honda*
Hummer*Hyundai*Infiniti*Isuzu*Jaguar*Jeep*Kia*Land
Rover*Lexus*Lincoln*MINI*Mazda*Mercedes-Benz*Mercury*Mitsubishi*Nissan
*Oldsmobile*Pontiac*Porsche*Saab*Saturn*Scion*Subaru*Suzuki*Toyota*Vol
kswagen*Volvo
```

The desired output for this task is a frequency table of each manufacturer by origin and type. The name of each car manufacturer should also be in the title of each frequency output. To accomplish this task the title statement along with the FREQ procedure can be contained within a macro that parses out unique manufacturers from the macro variable array *brands* with an asterisk as the delimiter (see macro below). The asterisk delimiter should be specified in both the COUNTW function as well as the %SCAN function such that manufacturer names are parsed out of the macro variable array in the correct fashion. Just like with a conventional macro variable, if the contents of a macro variable need to be used in a character setting double quotes are required around the macro variable. Double quotes are used in both the TITLE and WHERE statements below such that the *ith* manufacturer name in the macro variable array *brands*

will be used as a character string. There are 38 manufacturers in the *cars* data set indicating that this macro will execute a total of 38 times.

```
%macro frqz();
   %do i = 1 %to %sysfunc(countw(&brands., *));
      title "%scan(&brands., &i., *)";
      proc freq data=sashelp.cars;
         where make = "%scan(&brands., &i., *)";
         tables origin*type/list;
      run;
      title;
   %end;
%mend;


%frqz();
```

Below is the log execution of this macro.

```
84   %frqz();
MLOGIC(FRQZ):  Beginning execution.
MLOGIC(FRQZ):  %DO loop beginning; index variable I; start value is 1; stop value is 38;
      by value is 1.
MPRINT(FRQZ):   title "Acura";
MPRINT(FRQZ):   proc freq data=sashelp.cars;
MPRINT(FRQZ):   where make = "Acura";
MPRINT(FRQZ):   tables origin*type/list;
MPRINT(FRQZ):   run;

NOTE: Writing HTML Body file: sashtml.htm
NOTE: There were 7 observations read from the data set SASHELP.CARS.
      WHERE make='Acura';
NOTE: PROCEDURE FREQ used (Total process time):
      real time           1.50 seconds
      cpu time            0.73 seconds

...

MLOGIC(FRQZ):  %DO loop index variable I is now 38; loop will iterate again.
MPRINT(FRQZ):   title "Volvo";
MPRINT(FRQZ):   proc freq data=sashelp.cars;
MPRINT(FRQZ):   where make = "Volvo";
MPRINT(FRQZ):   tables origin*type/list;
MPRINT(FRQZ):   run;

NOTE: There were 12 observations read from the data set SASHELP.CARS.
      WHERE make='Volvo';
NOTE: PROCEDURE FREQ used (Total process time):
      real time           0.05 seconds
      cpu time            0.06 seconds


MPRINT(FRQZ):   title;
MLOGIC(FRQZ):  %DO loop index variable I is now 39; loop will not iterate again.
MLOGIC(FRQZ):  Ending execution.
```

Below is the result window for the first manufacturer Acura.

Acura

The FREQ Procedure

Origin	Type	Frequency	Percent	Cumulative Frequency	Cumulative Percent
Asia	SUV	1	14.29	1	14.29
Asia	Sedan	5	71.43	6	85.71
Asia	Sports	1	14.29	7	100.00

A PROC FREQ by *type* can almost get to the desired output but the name of the manufacturer would not be in the title of the output. By employing the use of the macro variable array the program will contain 23 lines of code (including the DATA NULL step) instead of 228 lines (when including the reset of the title after each PROC FREQ). Additionally, this method of programmatically creating the macro variable array can be faster and more accurate than explicitly defining the macro variable array since the programmer is not responsible for writing out and having the correct spelling each of the unique values of the manufacturers.

CREATION OF MACRO VARIABLE ARRAYS USING PROC SQL

The last method to create a macro variable, and subsequently a macro variable array, is by use of PROC SQL. PROC SQL can be used to select distinct values from a variable of interest from a data set and place these values into a macro variable via the INTO function (Long & Heaton, 2008). The delimiter separating unique values can also be specified in PROC SQL on the SELECT line after the macro variable is specified. In this example, the desired output is one report of the demographic information for each student in the SASHELP data set CLASS. Each student's name is contained inside the variable *name*. To accomplish this two macro variables will be created, one is a macro variable array and the other is a single-value macro variable. The PROC SQL step is outlined in detail below.

```
❶proc sql noprint;
   ❷select distinct(name) into :names separated by ' '
   from sashelp.class;
   ❸select count(distinct(name)) into :counts trimmed
   from sashelp.class;

❹quit;
```

❶ Since macro variables are being created using PROC SQL the *noprint* option is utilized to eliminate any results from the SQL statements.

❷Distinct values of the variable *name* are concatenated into the macro variable *names*, separated by a space, from the SASHELP.CLASS data set using the DISTINCT and INTO functions.

❸ The number of distinct values of the variable *name* is placed into the macro variable *count,* with the excess trailing spaces in the macro variable trimmed off, from the SASHELP.CLASS data set.

❹ The QUIT statement, rather than the RUN statement, is required to end the SAS SQL procedure.

Instead of making the macro variable *counts* that contains the number of distinct names in the *class* data set, %SYSFUNC in conjunction with COUNTW functions could be employed like in the previous examples. The REPORTS macro below utilizes the *counts* macro variable as the end counter for the macro do loop. One report will be generated per student in the *class* data set using PROC REPORT. Since there are 19 students there will be 19 reports generated.

```
%macro reports();
   %do i = 1 %to &counts.;
      title "%scan(&names., &i.)";
      proc report data=sashelp.class nowd;
```

```
            where name = "%scan(&names, &i.)";
            columns sex age height weight;
            define sex / 'Sex';
            define age / 'Age';
            define height / 'Height';
            define weight / 'Weight';
         run;
         title;
      %end;
   %mend;

   %reports();
```

By turning off the options MPRINT and MLOGIC the log will contain only notes for each report that is generated. If a report does not function correctly then the log will also contain any errors or warnings.

```
110  %reports();
NOTE: Writing HTML Body file: sashtml1.htm

NOTE: There were 1 observations read from the data set SASHELP.CLASS.
      WHERE name='Alfred';
NOTE: PROCEDURE REPORT used (Total process time):
      real time           1.36 seconds
      cpu time            0.53 seconds
```

…

```
NOTE: There were 1 observations read from the data set SASHELP.CLASS.
      WHERE name='William';
NOTE: PROCEDURE REPORT used (Total process time):
      real time           0.01 seconds
      cpu time            0.03 seconds
```

Below are the first and last generated reports using this macro.

Alfred

Sex	Age	Height	Weight
M	14	69	112.5

William

Sex	Age	Height	Weight
M	15	66.5	112

Having a counter macro variable instead of using %SYSFUNC in combination with COUNTW functions can be useful if the programmer wants to double check that the number of elements in the macro variable array match what they are expecting to be.

ADVANCED MACRO VARIABLE ARRAY EXAMPLE

Multiple Macro Variable Arrays Example

To make macro processing more advanced, multiple macro variable arrays can be utilized. In this example, two macro variable arrays will be used simultaneously to produce 4 outputs. Using the 2004 car data from the SASHELP.CARS data set, four correlations are going to be produced: the correlation between invoice amount and MPG highway, invoice amount and cylinders, MSRP and MPG highway, and MSRP and cylinders (SAS Institute Inc., 2017).

In the macro below there are three macro variables: DAT, VAR1, and VAR2. The macro variable DAT is the data set of interest from the SASHELP library. The macro variables VAR1 and VAR2 are going to be used as macro variable arrays such that multiple variables of interest can be specified in both macro variables. Each variable that is specified in the VAR1 macro variable array will be correlated against each

variable in the VAR2 macro variable array. Since there are two macro variable arrays there are two sets of loops: the first loop (or the outer loop) corresponds to the VAR1 macro variable and uses *i* as the counter while the second loop (or the inner loop) corresponds to the VAR2 macro variable and uses *k* as the counter. Both macro variable arrays will have each variable of interest parsed out of the array using the %SCAN function. Since there are two variables specified in each macro variable array the CORR macro will execute a total of four times.

```
%macro corr(dat, var1, var2);
   %do i = 1 %to %sysfunc(countw(&var1.));
      %do k = 1 %to %sysfunc(countw(&var2.));
         proc corr data=sashelp.&dat. noprint
                   outp=c_%scan(&var1., &i.)_%scan(&var2., &k.);
            var %scan(&var1., &i.) %scan(&var2., &k.);
         run;

         proc print data=c_%scan(&var1., &i.)_%scan(&var2., &k.)
                         noobs label;
            where lowcase(_name_) =  "%scan(&var1., &i.)";
            var %scan(&var2., &k.);
            label %scan(&var2., &k.) =
                 "Corr %scan(&var1., &i.)-%scan(&var2., &k.)";
         run;
      %end;
   %end;
%mend;

%corr(cars, invoice msrp, mpg_highway cylinders);
```

Below is the log for the first and last iterations of the CORR macro. HERE

```
130  %corr(cars, invoice msrp, mpg_highway cylinders);
MLOGIC(CORR):  Beginning execution.
MLOGIC(CORR):  Parameter DAT has value cars
MLOGIC(CORR):  Parameter VAR1 has value invoice msrp
MLOGIC(CORR):  Parameter VAR2 has value mpg_highway cylinders
MLOGIC(CORR):  %DO loop beginning; index variable I; start value is 1; stop value is 2;
      by value is 1.
MLOGIC(CORR):  %DO loop beginning; index variable K; start value is 1; stop value is 2;
      by value is 1.
MPRINT(CORR):   proc corr data=sashelp.cars noprint outp=c_invoice_mpg_highway;
MPRINT(CORR):   var invoice mpg_highway;
MPRINT(CORR):   run;

NOTE: Writing HTML Body file: sashtml1.htm
NOTE: The data set WORK.C_INVOICE_MPG_HIGHWAY has 5 observations and 4 variables.
NOTE: Compressing data set WORK.C_INVOICE_MPG_HIGHWAY increased size by 100.00 percent.
      Compressed is 2 pages; un-compressed would require 1 pages.
NOTE: PROCEDURE CORR used (Total process time):
      real time           1.26 seconds
      cpu time            0.73 seconds
```

…

```
MPRINT(CORR):   proc print data=c_msrp_cylinders noobs label;
MPRINT(CORR):   where lowcase(_name_) = "msrp";
MPRINT(CORR):   var cylinders;
MPRINT(CORR):   label cylinders = "Corr msrp and cylinders";
MPRINT(CORR):   run;

NOTE: There were 1 observations read from the data set WORK.C_MSRP_CYLINDERS.
      WHERE LOWCASE(_name_)='msrp';
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds


MLOGIC(CORR):  %DO loop index variable K is now 3; loop will not iterate again.
MLOGIC(CORR):  %DO loop index variable I is now 3; loop will not iterate again.
MLOGIC(CORR):  Ending execution.
```

The correlation values below show that both MSRP and the invoice price of vehicles in 2004 have a moderately negative correlation with MPG highway while both MSRP and invoice price have a moderately positive correlations with the number of cylinders in the vehicles.

Corr invoice and mpg_highway	Corr invoice and cylinders	Corr msrp and mpg_highway	Corr msrp and cylinders
-0.43459	0.64523	-0.43962	0.64974

Using the %STR and %BQUOTE Functions within a Macro Variable Array

For this example the infant birth weight data set form the SASHELP library will be utilized to obtain mean summaries of Mother's age (*mom_age*) and Mother's pregnancy weight gain (*m_wtgain*) such that the title of each means summaries matches the label of the variable of interest. In the MEANS macro below there are three macro variables: DAT for the data set of interest from the SASHELP library, the macro variable array VAR containing a list of the variables of interest, and the macro variable array LAB containing the title for the variables of interest.

```
%macro means(dat, var, lab);
   %do i = 1 %to %sysfunc(countw(&var.));
      title "%scan(&lab., &i., ~)";
      proc means data=sashelp.&dat.;
         var %scan(&var., &i.);
      run;
      title;
   %end;
%mend;
```

Since the labels of both *mom_age* and *m_wtgain* contain an apostrophe a masking function of some kind will need to be used within the macro variable array due to unbalanced quotes. Additionally, every label contains a space so a delimiter, set to ~, needed to be specified in the %SCAN of the LAB macro variable array. Either the macro functions %BQUOTE or %STR can be used to mask the unbalanced apostrophe in the labels. The %BQUOTE function masks the unbalanced quote when the macro is executed. The %STR function masks the unbalanced quote when the macro is complied. Additionally, unbalanced quotes using %STR function need to have a percent sign prior to the unbalanced quote such that the compiler executes correctly (Guo, 2016).

```
%means(bweight, mom_age m_wtgain, %bquote(Mother's
Age)~%bquote(Mother's Pregnancy Weight Gain));

%means(bweight, mom_age m_wtgain, %str(Mother%'s Age)~%str(Mother%'s
Pregnancy Weight Gain));
```

Both of the macro calls function the same the difference is in where the masking of the unbalanced quoting occurs. Depending on the task at hand knowing how and when to use masking functions can negate headaches that occur due to special characters contained within macro variables. Below is the log of this macro with the MLOGIC option on to show that each masking method works as intended.

```
146  %means(bweight, mom_age m_wtgain, %bquote(Mother's Age)~%bquote(Mother's Pregnancy
146! Weight Gain));
MLOGIC(MEANS):  Beginning execution.
MLOGIC(MEANS):  Parameter DAT has value bweight
MLOGIC(MEANS):  Parameter VAR has value mom_age m_wtgain
MLOGIC(MEANS):  Parameter LAB has value ┘Mother◄s Age□~┘Mother◄s Pregnancy Weight Gain□
MLOGIC(MEANS):  %DO loop beginning; index variable I; start value is 1; stop value is 2;
      by value is 1.

NOTE: There were 50000 observations read from the data set SASHELP.BWEIGHT.
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.04 seconds
      cpu time            0.06 seconds


MLOGIC(MEANS):  %DO loop index variable I is now 2; loop will iterate again.

NOTE: There were 50000 observations read from the data set SASHELP.BWEIGHT.
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.03 seconds
      cpu time            0.03 seconds


MLOGIC(MEANS):  %DO loop index variable I is now 3; loop will not iterate again.
MLOGIC(MEANS):  Ending execution.
150  %means(bweight, mom_age m_wtgain, %str(Mother%'s Age)~%str(Mother%'s Pregnancy Weight
150!  Gain));
MLOGIC(MEANS):  Beginning execution.
MLOGIC(MEANS):  Parameter DAT has value bweight
MLOGIC(MEANS):  Parameter VAR has value mom_age m_wtgain
MLOGIC(MEANS):  Parameter LAB has value   Mother◄s Age┐~  Mother◄s Pregnancy Weight Gain┐
MLOGIC(MEANS):  %DO loop beginning; index variable I; start value is 1; stop value is 2;
      by value is 1.

NOTE: There were 50000 observations read from the data set SASHELP.BWEIGHT.
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.03 seconds
      cpu time            0.03 seconds


MLOGIC(MEANS):  %DO loop index variable I is now 2; loop will iterate again.

NOTE: There were 50000 observations read from the data set SASHELP.BWEIGHT.
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.03 seconds
      cpu time            0.04 seconds


MLOGIC(MEANS):  %DO loop index variable I is now 3; loop will not iterate again.
MLOGIC(MEANS):  Ending execution.
```

CONCLUSION

Macro code in general is a way to optimize executing repetitive tasks on multiple variables, data sets, or values. Macro variable arrays take this a step further by combining the efficiency of array processing with macro language. Macro variable arrays are a technique to make macros more robust in nature while cutting down the lines of code in a given program, ultimately reducing programmer error. Macro variable arrays can be explicitly defined as a macro variable during the creation of a macro or by a %LET statement. They can also be created using the CALL SYMPUT routine within a DATA step or the INTO statement within PROC SQL. By programmatically creating macro variable arrays via the last two methods programmer error is dramatically reduced. Depending on the number of repetitive tasks within a given program macro variable arrays can effectively accomplish an objective in minimal code when compared to brute force code writing.

APPENDIX

SAS CODE FOR ALL EXAMPLES

```
/*************************************************************/
/*********** Printing All Macro Variables to the Log **********/
/*************************************************************/
%put _all_;

/*************************************************************/
/*** Macro Variable Resolution and Array of Macro Variables ****/
/*************************************************************/
** Below is an array of macro variables since each macro variable
** starts with the same prefix race and then has an iterative
** integer counter as the suffix;
%let race1 = white;
%let race2 = asian;
%let race3 = other;

** Resolving just one of the macro variables in the array of
** macro variables;
%put &race1.;

** Consecutive && resolution for an array of macro variables;
options mprint mlogic symbolgen;
%macro print();
   %do i = 1 %to 3;
      %put &&race&i..;
   %end;
%mend;

%print();

** Showing error of improper use of &race&i;
%macro print();
   %do i = 1 %to 3;
      %put &race&i..;
   %end;
%mend;

%print();

/*************************************************************/
/******************** Macro Variable Arrays ******************/
/*************************************************************/
%let race = white asian other;

%macro print();
   %do i = 1 %to 3;
      %put %scan(&race., &i.);
   %end;
%mend;
```

```
%print();

/*************************************************************/
/******* Explicitly Defined Macro Variable Array Example *******/
/*************************************************************/
%macro rin(lib, var, dat);
   %do i = 1 %to %sysfunc(countw(&dat.));
      proc sort data=&lib..%scan(&dat., &i.)
                out= r%scan(&dat., &i.);
         by &var.;
      run;
   %end;
%mend;

%rin(sashelp, height, class classfit fish gridded heart);

/*************************************************************/
/** Creating a Macro Variable Array Using CALL SYMPUT Example **/
/*************************************************************/
data _null_;
   set sashelp.cars end = last;
   by make;
   length brand $5000;
   retain brand;
   if _n_ = 1 then brand = '';
   if first.make then brand = strip(brand)||'*'||strip(make);
   if last then do;
      brand = strip(substr(brand, 2));
      call symput('brands', strip(brand));
   end;
run;

%put &brands.;

%macro frqz();
   %do i = 1 %to %sysfunc(countw(&brands., *));
      title "%scan(&brands., &i., *)";
      proc freq data=sashelp.cars;
         where make = "%scan(&brands., &i., *)";
         tables origin*type/list;
      run;
      title;
   %end;
%mend;

%frqz();

/*************************************************************/
/*** Creating a Macro Variable Array Using PROC SQL Example ****/
/*************************************************************/
proc sql noprint;
```

```
   select distinct(name) into :names separated by ' '
   from sashelp.class;
   select count(distinct(name)) into :counts trimmed
   from sashelp.class;
quit;

%put counts = &counts., names = &names.;

options nomprint nomlogic;
%macro reports();
   %do i = 1 %to &counts.;
      title "%scan(&names., &i.)";
      proc report data=sashelp.class nowd;
         where name = "%scan(&names, &i.)";
         columns sex age height weight;
         define sex / 'Sex';
         define age / 'Age';
         define height / 'Height';
         define weight / 'Weight';
      run;
      title;
   %end;
%mend;

%reports();

/*************************************************************/
/*********** Advanced Macro Variable Array Example: ***********/
/*********** Multiple Macro Variable Arrays Example ***********/
/*************************************************************/
options mprint mlogic;

%macro corr(dat, var1, var2);
   %do i = 1 %to %sysfunc(countw(&var1.));
      %do k = 1 %to %sysfunc(countw(&var2.));
         proc corr data=sashelp.&dat. noprint
                   outp=c_%scan(&var1., &i.)_%scan(&var2., &k.);
            var %scan(&var1., &i.) %scan(&var2., &k.);
         run;

         proc print data=c_%scan(&var1., &i.)_%scan(&var2., &k.) noobs
label;
            where lowcase(_name_) =  "%scan(&var1., &i.)";
            var %scan(&var2., &k.);
            label %scan(&var2., &k.) = "Corr %scan(&var1., &i.) and
%scan(&var2., &k.)";
         run;
      %end;
   %end;
%mend;

%corr(cars, invoice msrp, mpg_highway cylinders);
```

```
/*************************************************************/
/*********** Advanced Macro Variable Array Example: ************/
/****** Using %STR Function within a Macro Variable Array ******/
/*************************************************************/
options nomprint mlogic;

%macro means(dat, var, lab);
   %do i = 1 %to %sysfunc(countw(&var.));
      title "%scan(&lab., &i., ~)";
      proc means data=sashelp.&dat.;
         var %scan(&var., &i.);
      run;
      title;
   %end;
%mend;

** This macro call needs BQUOTE around the text of interest that has
** unbalanced apostrophes otherwise the macro does not execute
** correctly;
%means(bweight, mom_age m_wtgain, %bquote(Mother's
Age)~%bquote(Mother's Pregnancy Weight Gain));

** %STR could also be used however unbalanced quotes need to be
** proceeded by a % sign;
%means(bweight, mom_age m_wtgain, %str(Mother%'s Age)~%str(Mother%'s
Pregnancy Weight Gain));
```

REFERENCES

Guo, P. (2016). Macro Quoting: Which Function Should We Use? *Pharma SAS(R) Users Group.* Shanghai, China: SAS Institute Inc. Retrieved from https://www.lexjansen.com/pharmasug-cn/2016/PS/PharmaSUG-China-2016-PS05.pdf

Long, S., & Heaton, E. (2008). Using the SAS(R) DATA Step and PROC SQL to Create Macro Arrays. *SAS(R) Global Forum.* San Antonio. Retrieved from http://www2.sas.com/proceedings/forum2008/105-2008.pdf

SAS Institute Inc. (2009). Introduction to Macro Variables. In S. I. Inc., *SAS® 9.2 Macro Language Reference.* Cary, NC: SAS Institute Inc. Retrieved JULY 10, 2018, from SAS Customer Support: http://support.sas.com/documentation/cdl/en/mcrolref/61885/PDF/default/mcrolref.pdf

SAS Institute Inc. (2011). *SAS® 9.2 Language Reference Dictionary* (Vol. 4). Cary, NC: SAS Institute Inc. Retrieved from SAS(R): http://support.sas.com/documentation/cdl/en/lrdict/64316/PDF/default/lrdict.pdf

SAS Institute Inc. (2017). *Sashelp Data Sets.* Retrieved AUG 03, 2018, from SAS: https://support.sas.com/documentation/tools/sashelpug.pdf

Werner, N. L. (2014). Understanding Double Ampersand [&&] SAS® Macro Variables . *Midwest SAS(R) Users Group.* Chicago: MWSUG. doi:BI-03-2014

ACKNOWLEDGMENTS

A special thanks to Laura Cole for proof reading and Kirk Paul Lafler for mentorship.

RECOMMENDED READING

Carpenter, A. L. (2000). Using Macro Functions. *SAS(R) Users Group International Conference.* Indianapolis: SAS Institute Inc. Retrieved from http://www2.sas.com/proceedings/sugi25/25/aa/25p004.pdf

Guo, P. (2016). Macro Quoting: Which Function Should We Use? *Pharma SAS(R) Users Group.* Shanghai, China: SAS Institute Inc. Retrieved from https://www.lexjansen.com/pharmasug-cn/2016/PS/PharmaSUG-China-2016-PS05.pdf

Long, S., & Heaton, E. (2008). Using the SAS® DATA Step and PROC SQL to Create Macro Arrays. *SAS(R) Global Forum.* San Antonio. Retrieved from http://www2.sas.com/proceedings/forum2008/105-2008.pdf

Spruell, B. (2009). Short, Sweet and Simple…how to do more with less in SAS®. *Southeast SAS(R) Users Group.* Retrieved from https://analytics.ncsu.edu/sesug/2009/CC006.Spruell.pdf

CONTACT INFORMATION

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

Veronica Renauldo
QST Consultations, LTD.
(616) 892-3723
vrenauldo@qstconsultations.com