Paper 1516 - 2017

Five Ways to Create Macro Variables: A Short Introduction to the Macro Language

Arthur L. Carpenter
California Occidental Consultants, Anchorage, AK

ABSTRACT

The macro language is both powerful and flexible. With this power, however comes complexity, and this complexity often makes the language more difficult to learn and use. Fortunately one of the key elements of the macro language is its use of macro variables, and these are easy to learn and easy to use.

Macro variables can be created using a number of different techniques and statements. However the five most commonly methods are not only the most useful, but also among the easiest to master. Since macro variables are used in so many ways within the macro language, learning how they are created can also serve as an excellent introduction to the language itself. These methods include:

- %LET statement
- macro parameters (named and positional)
- iterative %DO statement
- using the INTO in PROC SQL
- using the CALL SYMPUTX routine

KEYWORDS

macro variable, %LET, INTO, CALL SYMPUT, macro parameter, %DO

INTRODUCTION

The macro variable, also known as a symbolic variable, is key to the use of the macro language. With the capability of storing up to 64k bytes of information, you could store a complete program or even the text of a novel within a single macro variable. While neither of these uses of macro variables is particularly valuable, understanding what can be stored in a macro variable and how to get the information into the macro variable is crucial to the use of the macro language.

The text and examples that follow show a few ways to load values into a macro variable. Although valuable in and of itself, the resulting discussion also serves as a brief introduction to the wider topics of the macro language. The discussion starts with what tend to be the five most commonly used methods for creating and loading macro variables. This is followed by a brief discussion of a few other 'bonus' methods.

%LET

The %LET statement is very often the first macro language statement that is learned. It is roughly the macro language equivalent of the of the DATA step's assignment statement.

One of the easiest ways to define a macro variable is through the %LET statement. (Macro language statements always start with a %). This statement works much like an assignment statement in the DATA step.

The %LET statement is followed by the macro variable name, an equal sign (=), and then the text value to be assigned to the macro variable. Notice that quotation marks are not used. Unlike data set variables, macro variables are neither character nor numeric; they always just store text. While learning the macro language, SAS programmers familiar with DATA set variables, may find it easier to think of them as character. Because SAS knows that whatever is to the right of the equal sign is to be assigned to the macro variable, quotes are not needed. Indeed, when they are used they become part of the value that is stored.

The syntax of the %LET statement is

```
%LET macro-variable-name = text-or-text-value;
```

The following statement assigns the text string clinics to the macro variable DSN:

```
%LET dsn = clinics;
```

If the %LET statement is outside of any macro, its value will be available throughout the entire program, and it is said to be a global macro variable. On the other hand, if the macro variable is defined inside of a macro it may be local, and its value will only be available within that macro.

The macro language does not support the concept of a missing value. Unlike data set variables, macro variables can actually contain nothing. In the macro language this is often referred to as a null value. The %LET statement does not store non-embedded blanks, so each of the following pairs of %LET statements will store the same value (in this case the value stored in &NADA is actually nothing – null).

```
%let nada =;
%let nada =     ;

%let dsn =clinics;
%let dsn =        clinics   ;
```

If you do wish to store a blank, as opposed to a null value, you will need to use a quoting function.

USING MACRO VARIABLES

You could use the following SAS program to determine the contents and general form of the data set WORK.CLINICS. It uses PROC CONTENTS and PROC PRINT (limiting the print to the first ten observations).

```
PROC CONTENTS DATA=CLINICS;
   TITLE 'DATA SET CLINICS';
   RUN;
PROC PRINT DATA=CLINICS (OBS=10);
   RUN;
```

Macro variables are especially useful when you generalize programs. The previous program works for only one data set. If you want to apply it to a different data set, you will need to edit it in three different places. This is trivial in this situation, but edits of existing production programs can be a serious problem in actual applications.

Fortunately the program can be rewritten and generalized. ❶ The %LET statement defines the macro variable. ❷ A macro variable (&DSN) replaces the data set name. The program becomes:

```
%LET DSN = CLINICS; ❶
PROC CONTENTS DATA=&dsn;❷
   TITLE "DATA SET &dsn"; ❸
   RUN;
PROC PRINT DATA=&dsn ❷(OBS=10);
   RUN;
```

To change the data set name, you still need to edit the %LET statement. At least it is now a simpler task.

Notice that in the rewritten code, quotes in the TITLE statement ❸ were changed from single to double quotes. Macro variables that appear inside of a quoted string will not be resolved unless you use double quotes (").

You can change the value of a macro variable simply by issuing a new %LET statement. The most recent definition will be used at any given time.

The period or dot can be used to terminate the name of the unresolved macro variable. Although the macro variable name &DSN is interchangeable with &DSN., most macro programmers only add the period when it is needed to minimize confusion.

DISPLAYING MACRO VARIABLES

The %PUT statement, which is analogous to the DATA step PUT statement, writes text and the current values of macro variables to the SAS System LOG. As a macro statement the %PUT statement (unlike the PUT statement) does not need to be inside of a DATA step. The following two SAS statements comprise a complete (albeit silly) program:

```
%LET dsn = clinics;
%PUT ***** selected data set is &dsn;
```

Notice that unlike the PUT statement the text string is not enclosed in quotes. The quotes are not needed because, unlike in the DATA step, the macro facility does not need to distinguish between variable names and text strings. Everything is a text string, a macro language reference, or other macro language trigger. The macro language can easily recognize macro variables, for instance, since they are preceded by an ampersand.

There are several options that can be used on the %PUT statement. If you want to see the current values of the macro variables that you have created you can use the following:

```
%put _user_;
```

Messages that mimic those written to the LOG can also be generated by using the %PUT statement. When the %PUT is followed by ERROR:, WARNING:, or NOTE: the text associated with the %PUT will be written to the LOG in the color appropriate to that message. Under the default settings in an interactive environment, the following %PUT generates a red error message in the log.

```
%PUT ERROR: Files were not copied as expected.;
```

The keywords must be capitalized, must immediately follow the %PUT, and must be immediately followed by a colon.

MACRO PARAMETERS

Positional Parameters

Positional parameters are defined by listing the macro variable names that are to receive the parameter values in the %MACRO statement. When parameters are present, the macro name is followed by a comma-separated list of macro variables that are enclosed in a pair of parentheses.

The following version of %LOOK uses the %LET to establish two global macro variables (&DSN and &OBS).

```
%LET DSN = CLINICS;
%LET OBS = 10;
%MACRO LOOK;
  PROC CONTENTS DATA=&dsn;
    TITLE "DATA SET &dsn";
    RUN;

  PROC PRINT DATA=&dsn (OBS=&obs);
    TITLE2 "FIRST &obs OBSERVATIONS";
    RUN;
%MEND LOOK;
```

We can easily convert this macro so that it uses positional parameters rather than relying on the %LET. The following version of %LOOK has two positional parameters, and it is more flexible:

```
%MACRO LOOK(dsn,obs);
   PROC CONTENTS DATA=&dsn;
     TITLE "DATA SET &dsn";
     RUN;

   PROC PRINT DATA=&dsn (OBS=&obs);
     TITLE2 "FIRST &obs OBSERVATIONS";
     RUN;
%MEND LOOK;
```

The only difference in these two versions of %LOOK is in the %MACRO statement. The parameters allow us to create &DSN and &OBS as local macro variables and we are not required to modify the macro itself. Because the parameters are positional, the first value in the macro call is assigned to the macro variable that is listed first in the macro statement's parameter list. When you have multiple parameters, you need to use commas to separate their values.

The macro call for %LOOK could be

```
%LOOK(CLINICS,10)
```

You do not have to give all parameters a value. Alternative invocations of the %LOOK macro might include:

```
%LOOK()
%LOOK(CLINICS)
%LOOK(,10)
```

Macro variables that are not assigned a value will resolve to a null string. Thus, the macro call `%LOOK(,10)` resolves to

```
PROC CONTENTS DATA=;
  TITLE "DATA SET ";
  RUN;

PROC PRINT DATA= (OBS=10);
  TITLE2 "FIRST 10 OBSERVATIONS";
  RUN;
```

The resolved code contains syntax errors, and it will not run. Be careful to construct code that will resolve to what you expect, and when possible anticipate and code around problems like this one.

Keyword or Named Parameters

Keyword parameters are designated by following the parameter name with an equal sign (=). Default values, when present, follow the equal sign. You can use keyword parameters to redefine the previous version of the %LOOK macro.

```
%MACRO LOOK(dsn=CLINICS,obs=);
    PROC CONTENTS DATA=&dsn;
         TITLE "DATA SET &dsn";
         RUN;

    PROC PRINT DATA=&dsn (OBS=&obs);
         TITLE2 "FIRST &obs OBSERVATIONS";
        RUN;
%MEND LOOK;
```

In this version of %LOOK, the macro variable &DSN will have a default value of CLINICS, while &OBS does not have a default value. If a value is not passed to &OBS, &OBS will take on a null value, in much the same way as a positional parameter will when it is not provided a value.

When you use the version of the macro %LOOK that is defined with keyword parameters, the macro call `%LOOK(OBS=10)` resolves to

```
PROC CONTENTS DATA=CLINICS;
     TITLE "DATA SET CLINICS";
     RUN;

PROC PRINT DATA=CLINICS (OBS=10);
     TITLE2 "FIRST 10 OBSERVATIONS";
     RUN;
```

Because the macro call %LOOK(obs=10) did not include a definition for &DSN, the default value of CLINICS was used. However since &OBS does not receive a default value, syntax errors will still result if a value is not provided for &OBS. As a general rule it is a good idea to provide appropriate default values for all your parameters so that the macro will work correctly regardless of which parameters are specified by the user.

ITERATIVE %DO

The macro language allows the user to specify %DO loops that are similar to the DO loop in the DATA step. Although the form of the iterative %DO is similar to the DO statement, it differs in that

- the %WHILE and %UNTIL specifications cannot be added to the increments
- increments are integer only
- only one specification is allowed.

Syntax

```
%DO macro-variable = start %TO stop <%BY increment>;
. . . text . . .
```

```
        %END;
```

The iterative %DO defines and increments a macro variable. In the following example, the macro variable &YEAR is incremented by one starting with &START and ending with &STOP ❶. Although the DATA step also has a variable YEAR, the two will not be confused. The incoming data sets are named YR95, YR96, and so on. These are read one at a time, and they are appended to the all-inclusive data set ALLYEAR ❷.

```
%MACRO ALLYR(START,STOP);
     %DO YEAR = &START %TO &STOP; ❶
          DATA TEMP;
               SET YR&YEAR;
               YEAR = 1900 + &YEAR;
          RUN;
          PROC APPEND BASE=ALLYEAR DATA=TEMP; ❷
          RUN;
     %END;
%MEND ALLYR;
```

The macro call `%ALLYR(95,97)` generates the following code:

```
DATA TEMP;
     SET YR95;
     YEAR = 1900 + 95;
RUN;
PROC APPEND BASE=ALLYEAR DATA=TEMP;
RUN;

DATA TEMP;
     SET YR96;
     YEAR = 1900 + 96;
RUN;
PROC APPEND BASE=ALLYEAR DATA=TEMP;
RUN;

DATA TEMP;
     SET YR97;
     YEAR = 1900 + 97;
RUN;
PROC APPEND BASE=ALLYEAR DATA=TEMP;
RUN;
```

You can greatly simplify this code by taking better advantage of the %DO loop. Rather than having the macro create separate DATA and APPEND steps for each year, you can build the code dynamically.

```
%MACRO ALLYR(START,STOP);
   DATA ALLYEAR;
      SET
      %DO YEAR = &START %TO &STOP;
            YR&YEAR(IN=IN&YEAR)
      %END;;

      YEAR = 1900
      %DO YEAR = &START %TO &STOP;
           + (IN&YEAR*&YEAR)
      %END;;

      RUN;
%MEND ALLYR;
```

This time the call to `%ALLYR(95,97)` produces the following:

```
DATA ALLYEAR;
   SET
        YR95(IN=IN95)
        YR96(IN=IN96)
        YR97(IN=IN97)
   ;

   YEAR = 1900
        + (IN95*95)
```

```
                + (IN96*96)
                + (IN97*97)
           ;

    RUN;
```

In this code, the value of YEAR is assigned by taking advantage of the IN= data set option. The values of IN95, IN96, and IN97 will be either true or false (1 or 0), and only one of the three will be true at any given time.

INTO IN PROC SQL

PROC SQL can be used to create macro variables by writing directly to the symbol tables.

Placing a single value in a macro variable

The following example uses SQL to count the number of observations that contain a specified string in the table column CLINNAME. The string is placed in a macro variable (&CLN) and the SQL COUNT function is used to count the observations that match the WHERE clause.

```
%let cln = Beth;
proc sql noprint;
  select count(*)
    into :nobs ❶
      from clinics(where=(clinname=:"&cln")); ❸
  quit;
%put number of clinics for &cln is &nobs; ❷
```

❶ The INTO clause is used to create the new macro variable NOBS. The colon informs the SELECT statement that the result of the COUNT function is to be written into a macro variable.

❷ Once created, the new macro variable is used in the same way as any other macro variable. Both macro variables are preceded by an ampersand in the %PUT statement.

❸ Notice that the colon in the FROM clause is used as a character comparison operator just as it is in the DATA step. SAS will not be confused by these two very different uses of the colon.
Within SQL, macro variables are also known as environmental variables. When being assigned values the name of the macro variable will follow an INTO and will be preceded by a colon. As is shown in this example, when macro variables are to be used within a SQL step the macro variable is preceded by the ampersand.

Creating lists of values

It is possible to create more than one macro variable in the SELECT statement. In the example below two macro variables are created (&LASTNAMES and &DOBIRTHS) each will contain a comma separated list of values.

```
proc sql noprint;
select lname, dob ❶
   into :lastnames separated by ',', ❷
   :dobirths separated by ',' ❸
      from sasclass.clinics(where=(lname=:'S')); ❹
%let numobs=&sqlobs; ❺
quit;
%put lastnames are &lastnames;
%put dobirths are &dobirths;
%put number of obs &numobs;
```

❶ The values of the variables LNAME and DOB will be written INTO macro variables.

❷ The list of values of LNAME will be written into &LASTNAMES with the values separated by a comma. Without the SEPARATED BY clause the individual values will replace previous values rather than be appended onto the list. The last comma is needed to separate the two macro variables (:LASTNAMES and :DOBIRTHS).

❸ The list of the dates of birth will be written to &DOBIRTHS.

❹ The WHERE clause is applied to the incoming data.

❺ The SQL step automatically loads the macro variable &SQLOBS. In this case it contains the number of rows in the incoming data set that meet the WHERE criteria.

The data set contains five observations with a last name, LNAME, which starts with an 'S'. The variable DOB has been assigned the format DATE7., and the format is automatically used when the values are written to the macro variable &DOBIRTHS. The LOG shows:

```
29   %put lastnames are &lastnames;
lastnames are Smith,Simpson,Simpson,Stubs,Saunders
30   %put dobirths are &dobirths;
dobirths are 18MAR52,18APR33,18APR33,11JUN47,01MAR49
31   %put number of obs &numobs;
number of obs 5
```

Placing a List of Values into a Series of Macro Variables

Rather than placing a list of values into a specific macro variable as was done in the previous example, you can create a series of macro variables each with its own distinct value. In the following example PROC CONTENTS will create a table that will contain one row for each variable in the original data set (&DSN). PROC SQL is then used to write the variable names into a series of macro variables of the form &VARNAME1, &VARNAME2, ...

```
%macro varlist(dsn);
* Determine the list of variables in this
* base data set;
proc contents data= &dsn ❶
              out= cont noprint;
   run;

* Collect the variable names;
proc sql noprint;
   select distinct name ❷
      into :varname1-:varname999 ❸
         from cont;
   quit;

%do i = 1 %to &sqlobs; ❹
   %put &i &&varname&i;
%end;
%mend varlist;

%varlist(sasclass.clinics)
```

❶ The OUT= data set (WORK.CONT) has one row per variable in &DSN. The names of these variables are stored in the variable NAME.

❷ Select each unique variable name from NAME (they should already be unique).

❸ The variable names are stored in a series of macro variables &VARNAME1, &VARNAME2, etc. As it is currently coded, no more than 999 values can be stored. This is way more than is needed in this example, but SAS will only create the number that are actually required (not the full 999).

CAVEAT: When specifying the upper bound for the number of macro variables (999 in this case), you should be **VERY** sure that the number is large enough. If it is too small, values that do not fit will be lost and no error or warning will be issued.

❹ The number of distinct values of NAME is stored in &SQLOBS. This value can be used to cycle through the list of macro variables. The %DO loop is included here only as an illustration, and would not normally be used in a macro such as %VARLIST.

The data set SASCLASS.CLINICS has 20 variables, and the LOG shows:

```
1 ADMIT
2 CLINNAME
3 CLINNUM
4 DEATH
5 DIAG
6 DISCH
7 DOB
8 DT_DIAG
9 EDU
10 EXAM
11 FNAME
12 HT
13 LNAME
14 PROCED
15 RACE
16 REGION
17 SEX
18 SSN
19 SYMP
20 WT
```

Starting in SAS 9.0 you can specify the list of macro variables with leading 0s. The SELECT statement becomes:

```
         select distinct name
            into :varname001-:varname999
```

Unlike the previous example which created &VARNAME1, &VARNAME2, etc., this revised statement would produce &VARNAME001, &VARNAME002, etc. While specifying your list this way would make it easier to sort the macro variable names, it would however, make it more difficult to step through the variables using an incremented %DO loop.

USING THE CALL SYMPUT ROUTINE

A common difficulty for programmers new to the macro language is to use DATA set values and logic to assign values to macro variables. Because of the timing of when macro language statements such as %LET are executed, they cannot be used to assign the values contained in DATA step variables to macro variables.

In the following pseudo code, the programmer wants to assign the value of the variable FNAME to the macro variable &FIRSTN.

```
      data new;
        set old;
        %let firstn = fname;
        run;
```

Instead, because the %LET is executed long before the DATA step is even completely compiled, the macro variable &FIRSTN will contain the lowercase letters f-n-a-m-e. To get around this problem we need to be able to use the DATA step to place information onto the macro variable symbol tables. We can do this with SYMPUT which is a DATA step call routine, not a macro language statement.

Like the %LET statement, the SYMPUT call routine can be used to assign a DATA step value to a macro variable. However, SYMPUT is NOT a macro level statement; instead it is a DATA step routine. As such, it is used as part of a DATA step and enables you to directly assign values of data set variables to macro variables.

The routine has two arguments, either of which may be a variable or a constant (such as a character string) or a combination of the two. The first argument identifies the name of the macro variable, and the second argument identifies the value to be assigned to it.

Syntax

```
      CALL SYMPUT(macro_varname,value);
```

The TITLE1 statement in the following example contains a macro variable (&SEX) that is defined in the previous DATA step. &SEX will contain the value of the data set variable SEX.

```
data regn1;
set clinics;
where reg='1';
call symput('sex',sex);
run;

title1 "Region 1 data for &sex";
```

Because the SYMPUT routine is executed for each observation that is processed (each pass of the DATA step), this code will reassign the value of &SEX for each observation that meets the WHERE criteria. In this example the macro variable &SEX will contain the value of the DATA variable SEX from the last observation read from the data set CLINICS.

In the SQL examples we were able to create a list of macro variables each containing a separate value. We can do the same in a DATA step using SYMPUT. The following DATA step creates a series of macro variables containing the names of the students in the SASHELP.CLASS data set.

```
data _null_;
   set sashelp.class;
   cnt = left(put(_n_,6.));
   call symput('name'||cnt,name);
   call symput('namecnt', cnt);
   run;

%put _user_;
```

The LOG shows.

```
17   %put _user_;
GLOBAL NAME8 Janet
GLOBAL NAMECNT 19
GLOBAL NAME9 Jeffrey
GLOBAL NAME6 James
GLOBAL NAME7 Jane
GLOBAL NAME4 Carol
GLOBAL NAME5 Henry
GLOBAL NAME2 Alice
GLOBAL NAME3 Barbara
GLOBAL NAME11 Joyce
GLOBAL NAME10 John
GLOBAL NAME1 Alfred
GLOBAL NAME13 Louise
GLOBAL NAME12 Judy
GLOBAL NAME15 Philip
GLOBAL NAME14 Mary
GLOBAL NAME17 Ronald
GLOBAL NAME16 Robert
GLOBAL NAME19 William
GLOBAL NAME18 Thomas
```

It is also not uncommon for users to try to access a macro variable in the same DATA step in which it is created. However, you cannot define a macro variable using SYMPUT and then attempt to access that macro variable using an & in the same DATA step. This makes sense when you remember that values are assigned to the macro variable through SYMPUT during DATA step execution. However, any macro variable references involving the use of an & are resolved before the compilation phase of the DATA step. And since the DATA step execution phase comes after the compilation of the entire DATA step is complete, the macro facility would be attempting to resolve a macro variable that would not be defined until the DATA step's execution phase. It is generally considered unfair to ask any language to resolve or use variables that have not yet been defined. This means that macro variable values cannot be obtained by referencing them with an & in the same step that they are created.

MORE THAN THE FIVE - A BONUS OF FOUR MORE

%WINDOW

Through the use of the %WINDOW statement, the macro language provides the programmer with a tool that can be used to establish a basic user interface. Similar to the WINDOW statement in the DATA step, %WINDOW can be used to create and display message boxes and to collect information from the user that can then be placed into macro variables.

The %WINDOW statement can be used to:

- display a window
- control window attributes including size and color
- make use of existing key and menu definitions
- display existing macro variable values
- define and assign values to macro variables

Once a window has been defined with the %WINDOW statement it can then be displayed by using the %DISPLAY statement. Both of these statements can be used in open code.

Syntax

```
%WINDOW window-name<attributes and display characteristics>;
%DISPLAY window-name <display control options>;
```

The following macro defines and then displays a macro window, which prompts the user for the name of a data set.

```
%macro dsnprompt(lib=sasuser);
* prompt user for the data set name;
%window verdsn color=white ❶
  #2 @5 "Specify the data set of interest" ❷
  #3 @5 "for the library &lib" ❸
  #4 @5 'Enter Name: '
         dsn 20 ❹ ATTR=UNDERLINE REQUIRED=YES ❺;

%display verdsn; ❻

proc print data=&lib..&dsn;
run;
%mend dsnprompt;

%dsnprompt(lib=sasclass)
```

When the %DSNPROMPT macro is executed, the VERDSN macro window will be defined and displayed.

❶ The VERDSN window will have a white background with no specifications for size.

❷ The text (in single or double quotes) is to be displayed at row 2 and column 5. The same notation for row (#) and column (@) is used as in the PUT and INPUT statements.

❸ Macro variables can be included in the text (see caveat below).

❹ The user is prompted for the name of a data set which is placed into &DSN.

❺ Attributes can be assigned to the display of the macro variable.

❻ Although defined, the VERDSN window is not displayed until the %DISPLAY statement is executed.

Automatic macro variables

There are a number of automatic macro variables of which the user can take advantage. Most of these are, as the name implies, created automatically by the system. It is also possible for the user to create a few automatic macro variables. &SYSPARM is one of these macro variables.

Usually used in the batch environment, this macro variable accesses the same value as is stored in the SYSPARM= system option and can also be retrieved using the SYSPARM( ) DATA step function. This option is most useful when its value is loaded when SAS is initially executed, and you may use it to pass a value into a program through the JCL or batch calling routines.

The SYSPARM initialization option specifies a character string that can be passed into SAS programs. In the following example, your programs will automatically direct your data to either a test or production library. To make this switch, assign &SYSPARM the value TST or PROD when you start the SAS session.

Assume that a VAX SAS session is initiated with

```
$ sas/sysparm=tst
```

A typical LIBNAME statement on VAX might be

```
libname projdat "usernode:[study03.gx&sysparm]";
```

The resolved LIBNAME statement becomes

```
libname projdat "usernode:[study03.gxtst]";
```

The syntax that you use to load a value into &SYSPARM depends on the operating environment that you are using. See the SAS Companion for your operating environment for more information.

On Windows, `-sysparm tst` appears on the TARGET LINE in the Properties Window. In JCL, the option is used on the SYSIN card.

The Output Delivery System, ODS

The MATCHALL= option for the OUTPUT destination on the ODS statement can be used to name a macro variable that will contain a list of SAS data tables. In the following example a PROC UNIVARIATE is executed with a BY statement. For each value of the BY variable (PRODUCT) a data set will be created which will contain the summary known as MOMENTS. The MATCHALL option requests separate data sets and the list of the names of these data sets is to be written to the macro variable &NAMELIST.

```
ods output Moments(match_all=namelist)=work.Moments;

proc univariate data=magdata;
   by product;
   var ampida;
   title1 'ODS Output - with BY Statement';
run;
ods output close;
```

Later these data sets can be combined by using the macro variable &NAMELIST.

```
data allmoments;
  set &namelist;
  run;
```

The LOG shows.

```
53
54   data allmoments;
55      set &namelist;
56      run;

NOTE: There were 6 observations read from the data set WORK.MOMENTS.
NOTE: There were 6 observations read from the data set WORK.MOMENTS1.
NOTE: There were 6 observations read from the data set WORK.MOMENTS2.
NOTE: There were 6 observations read from the data set WORK.MOMENTS3.
NOTE: There were 6 observations read from the data set WORK.MOMENTS4.
NOTE: The data set WORK.ALLMOMENTS has 30 observations and 11 variables.
```

%GLOBAL / %LOCAL

The %GLOBAL and %LOCAL statements are used to force macro variables into specific referencing environments or scopes. When they are used, and the macro variable(s) that they point to don't already exist, they will create the macro variables with null values.

If the macro variable &DSN does not already exist in the global symbol table, the following statement will create it.

```
%global dsn;
```

SUMMARY

Knowing how to create and use macro variables is the first step in learning the SAS Macro Language. One of the primary techniques is through the %LET statement. Other ways of creating macro variables includes the use of the iterative %DO loop, macro parameters, the SQL INTO: clause, and the DATA step SYMPUT routine.

While these are the primary methods for creating macro variables, there are several other ways within the macro language of assigning values to macro variables or of just creating a macro variable with a null value.

Fortunately there are lots of techniques available for the creation of macro variables. Multiple techniques provides us with flexibility, and flexibility gives us power and control.

REFERENCES

Burlew, Michele, *SAS® Macro Programming Made Easy*, Cary, NC: SAS Institute Inc.,1998, 280pp.

Carpenter, Art, *Carpenter's Complete Guide to the SAS® Macro Language 2nd Edition,* Cary, NC: SAS Institute Inc.,2004. Most of the examples in this paper were taken from this text.

ABOUT THE AUTHOR

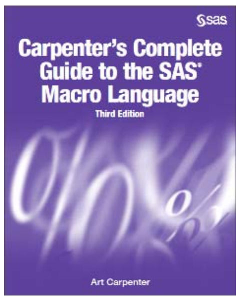
Art Carpenter is a SAS Certified Advanced Professional Programmer and his publications list includes; five books and numerous papers and posters presented at SAS Global Forum, SUGI, PharmaSUG, WUSS, and other regional conferences. Art has been using SAS since 1977 and has served in various leadership positions in local, regional, and international user groups.

Recent publications are listed on my sasCommunity.org Presentation Index page.
http://sascommunity.org/wiki/Presentations:ArtCarpenter_Papers_and_Presentations

AUTHOR CONTACT

Arthur L. Carpenter
California Occidental Consultants
10606 Ketch Circle
Anchorage, AK 99515

(907) 865-9167
art@caloxy.com
www.caloxy.com



TRADEMARK INFORMATION