Paper 077-2022

FROM %LET TO %LOCAL; METHODS, USE, AND SCOPE OF MACRO VARIABLES IN SAS® PROGRAMMING

Jay Iyengar, Data Systems Consultants LLC, Oak Brook, IL

ABSTRACT

Macro variables are one of the powerful capabilities of the SAS® system. Utilizing them makes your SAS code more dynamic. There are multiple ways to define and reference macro variables in your SAS code; from %LET and CALL SYMPUT to PROC SQL INTO. There are also several kinds of macro variables, in terms of scope and other ways. Not every SAS programmer is knowledgeable about the nuances of macro variables. In this paper, I explore the methods for defining and using macro variables. I also discuss the nuances of macro variable scope, and the kinds of macro variables from user-defined to automatic.

INTRODUCTION

Macro variables enable the SAS user to harness the power of the SAS Macro language. Macros and macro variables are part of the BASE SAS package. As SAS programmers are familiar with the DATA STEP, and its abilities to manipulate SAS variables on a data set, the SAS macro language manipulates macro variables, and performs actions on them. The features and processes of the SAS macro language to operate on macro variables are similar to the features and processes of the DATA STEP to operate on SAS data set variables. By definition, a macro variable is just a character or text string. However, the text string can consist of alpha numeric as well as numeric values. The traditional or conventional way to define a macro variable is to use %LET. However, there are some advantages to using CALL SYMPUT, and PROC SQL INTO: to define macro variables.

THE MACRO PROCESSOR

When a SAS program is submitted, the code has to be compiled before it's executed. During the compilation phase, the code is sent to an area called the input stack. From the input stack, the code is sent to the word scanner. In the word scanner the code is broken down into tokens. Tokens are smaller units which fall into one of four categories; literals, numbers, names, or special characters. Eventually, code is sent to the compiler. Here, SAS evaluates and checks the code for correct syntax, and will issue warning or error messages to the log, if the code violates syntax rules. This process is illustrated in Figure 1 below.

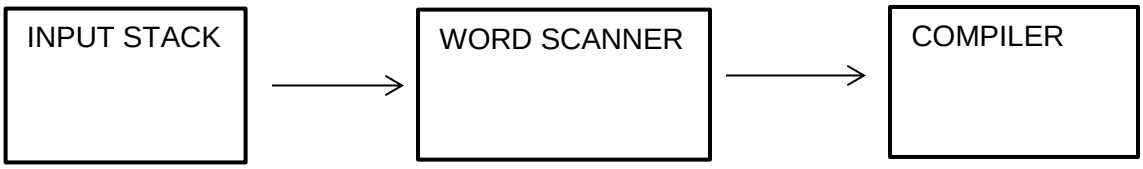

Figure 1. Flow of SAS code during the compilation phase.

SAS macro language syntax contains characters which are tokens and macro triggers. The primary characters which are macro triggers are % and &. When SAS macro language statements are submitted and these triggers are encountered by the word scanner, the text is alternately sent to the macro processor.

The macro processor evaluates the code, requests additional tokens if necessary to complete the statements, and then performs some action. For example, if you create a macro variable using the %LET statement, the code is broken into tokens by the word scanner, and then passed to the macro processor. This process is illustrated in Figure 2 below.

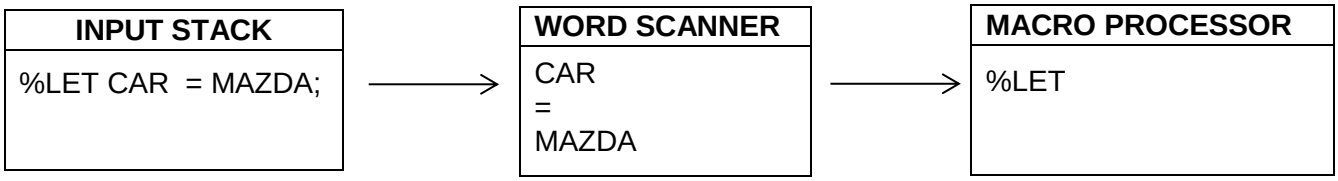

Figure 2. Macro tokens and the macro processor.

The macro processor evaluates and then executes the code. When the macro processor executes the %LET statement, the macro variable is created and stored in a symbol table, along with other user-defined or automatic macro variables. This process is illustrated in Figure 3 below.

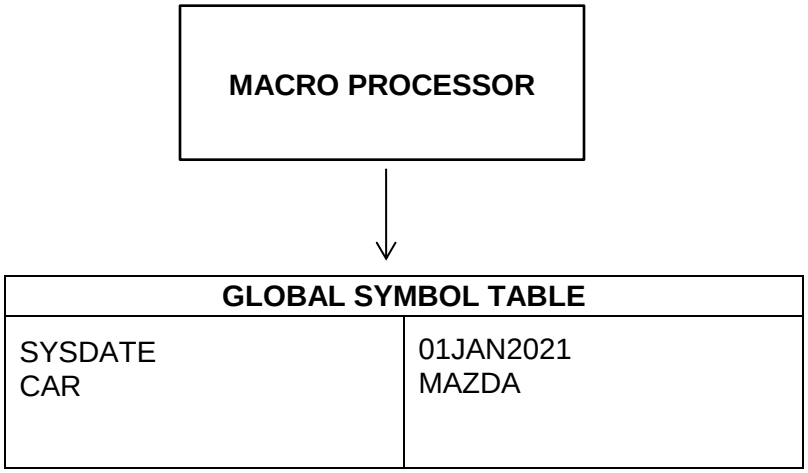

Figure 3. Macro processor and symbol table.

TYPES OF MACRO VARIABLES

The SAS macro facility provides two types of macro variables; automatic and user-defined. Automatic macro variables are built in to the SAS system which can be utilized in your SAS code, and generally have fixed values. User-defined macro variables are created and defined in SAS code by the programmer or SAS user, using one of several constructs in the SAS macro facility.

AUTOMATIC VS. USER DEFINED MACRO VARIABLES

When a SAS session is invoked and initialized, automatic macro variables are created and assigned values. Automatic macro variables provide information about your computing environment, such as the date and time your SAS Session began, the operating system platform SAS is running on, or the version of SAS that's installed. Automatic macro variables are global in scope, which means they're always available during a session, and can be referenced anywhere in your SAS code, either in open code, or inside a macro definition.

Although generally automatic macro variables contain values which cannot be modified during a SAS session, some automatic macro variables contain values which change during the course of a SAS session and can be reassigned. Examples and definitions of automatic macro variables are provided below in Table 1.

Automatic macro variable	Definition	Values
SYSDATE	The Date of the SAS Invocation (Date7. format)	FIXED
SYSTIME	The Time of the SAS Invocation.	FIXED
SYSSCP	The Operating System being used, WIN, HPUX, etc.	FIXED
SYSVER	The release of SAS that is being used.	FIXED
SYSLAST	The name of the most recently created SAS data set.	VARIABLE
SYSPARM	Contains text specified when SAS is invoked	VARIABLE
SYSERR	Return which indicates execution status of SAS code	N/A

Table 1. Automatic macro variables and definitions.

Most macro variables which programmers write, define and then reference in code are user-defined macro variables. User-defined macro variables contain values that are characters or text strings, and are defined in your SAS code, using any one of several SAS constructs. In contrast to automatic macro variables, user-defined macro variables can be either global or local in scope. Values for this type of macro variable can contain up to 65,534 characters.

For the purposes of this paper, user-defined macro variables will be known as macro variables. These macro variables can be processed in the compilation phase, or during the execution phase of SAS processing. Depending on the method chose, macro variable values can be defined with leading and trailing blanks preserved, or have leading\trailing blanks removed.

%PUT

Any SAS user can verify macro variables and their current values which have been defined in a SAS session. Whereas the PUT statement in a DATA STEP prints variables and values to the log, %PUT prints macro variables and their values in the SAS log.

Using %PUT, you have the option of displaying a single macro variable, multiple macro variables or a list of macro variables within a specific category. To validate a macro variable, call the macro variable (with an ampersand before the macro variable name) in a %PUT statement. The example in Figure 4 below demonstrates this.

```
1          OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
72
73         %LET TEAM=TAMPA BAY BUCCANEERS;
74         %PUT The Super Bowl Champions are the &TEAM;
The Super Bowl Champions are the TAMPA BAY BUCCANEERS
75
```

Figure 4. %PUT – displaying a single macro variable in the SAS log.

You can print a listing of macro variables within a specific category by specifying a keyword in the %PUT statement. To see a listing of automatic macro variables, specify the _AUTOMATIC_ keyword in %PUT. As the example in Figure 5 demonstrates, SAS prints a list of automatic macro variables with their current pre-set values.

```
73         %Put _Automatic_;
AUTOMATIC SYSDATE 03FEB21
AUTOMATIC SYSDATE9 03FEB2021
AUTOMATIC SYSDAY Wednesday
AUTOMATIC SYSJOBID 18882
AUTOMATIC SYSLAST WORK.VERSION_1612389170978
AUTOMATIC SYSPARM
AUTOMATIC SYSRC 0
AUTOMATIC SYSSCP LIN X64
AUTOMATIC SYSTIME 21:52
AUTOMATIC SYSVER 9.4
```

Figure 5. SAS log with example using %PUT _AUTOMATIC_.

You can specify the _USER_ keyword in the %PUT statement, to check and confirm user-defined macro variables. Running %PUT _USER_, SAS will generate a list of user-defined macro variables with their current values in the SAS log.

There are other %PUT options. By specifying %PUT _ALL_, SAS will print a listing of all macro variables (automatic and user-defined) in the SAS log. In Figure 6 below is an example containing an excerpt from the SAS log which displays a list of user-defined macro variables.

```
79         %PUT _USER_;
 GLOBAL CLIENTMACHINE 10.0.2.2
 GLOBAL COLOR SILVER
 GLOBAL COUNTRY_ORIGIN KOREA
 GLOBAL GRAPHINIT
 GLOBAL GRAPHTERM
 GLOBAL MAKE HYUNDAI
 GLOBAL MODEL ELANTRA
 GLOBAL STYLE SEDAN
 GLOBAL SYSCASINIT 0
 GLOBAL SYSSTREAMINGLOG true
 GLOBAL SYSUSERNAME sasdemo
```

Figure 6. %PUT listing in SAS log of user-defined macro variables.

Table 2 below displays the %PUT keywords which print different groups of macro variables to the log.

%PUT KEYWORD	DESCRIPTION
AUTOMATIC	AUTOMATIC MACRO VARIABLES
USER	USER DEFINED MACRO VARIABLES
ALL	ALL MACRO VARIABLES
LOCAL	LOCAL MACRO VARIABLES
GLOBAL	GLOBAL MACRO VARIABLES

Table 2. %PUT keywords.

SCOPE OF MACRO VARIABLES AND SYMBOL TABLES

Scope of macro variables adds another dimension to the understanding and use of macro variables. Scope determines where macro variables can be utilized within SAS coding structures and the programming environment. Macro variables can have either global or local scope. Global macro variables can be called and referenced at any location in SAS code; inside macro definitions, in open code, in the program where they're defined, and in other existing programs which have been created. Local macro variables can only be called and resolved within a macro program or macro definition. Most macro variables are global macro variables.

When macro variables are defined and processed, their saved and stored in a symbol table respective to their scope. Global macro variables are saved in the Global Symbol Table. Similarly, local macro variables are stored in a local symbol table. At the start of a SAS Session, a Global Symbol Table is created containing automatic macro variables with their pre-set values. It exists for the duration of the SAS session. The Global Symbol Table is deleted at the end of the session. Local symbol tables pertain to a specific SAS macro. They're created at the start of macro execution, exist while the macro executes, and are deleted at the end of macro execution. By default, macro parameters created in a %MACRO statement are local macro variables.

METHODS FOR DEFINING MACRO VARIABLES

The SAS macro facility within the BASE SAS package provides multiple ways to create macro variables. Methods for creating macro variables include %LET, CALL SYMPUT or CALL SYMPUTX, PROC SQL INTO clause, the %MACRO statement, or %GLOBAL and %LOCAL. By default, %LET, CALL SYMPUT, and PROC SQL INTO clause define global macro variables.

%LET

The common method of creating a macro variable is the %LET statement. In open code, %LET defines a global macro variable which can be called and resolved in any SAS program in a given session. By default, %LET creates a macro variable which has leading and trailing blanks removed from its value. Using %LET, a macro variable is created during the compilation phase of SAS processing, before SAS code is executed.

Figure 7 below provides an example of creating and referencing a macro variable using %LET.

```
73          %Let Make = Audi;
74
75          Proc Print Data=SASHELP.CARS;
76            Var Make Model Type DriveTrain MSRP;
77            Where Make="&Make";
78            Title "Type, Drivetrain, and MSRP &Make Models";
79          Run;

NOTE: There were 19 observations read from the data set SASHELP.CARS.
      WHERE Make='Audi';
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.27 seconds
      cpu time            0.26 seconds

81          Proc Means Data=SASHELP.CARS Mean;
82            Var MSRP;
83            Class DriveTrain;
84            Where Make="&Make";
85            Title1"Average Price for &Make Models";
86            Title2"By Drivetrain";
87          Run;

NOTE: There were 19 observations read from the data set SASHELP.CARS.
      WHERE Make='Audi';
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.13 seconds
      cpu time            0.11 seconds
```

Figure 7. SAS log excerpt featuring %LET.

%STR VS. %NRSTR

Depending on the value of your macro variable, it may be necessary to use a macro quoting function in a %LET statement to quote the value. Macro quoting functions mask specific characters in macro variables, which enable the macro processor to interpret them as ordinary text.

For example, you may have apostrophes in the value, which you don't want interpreted as quotation marks. In another example, you may have an ampersand (&), or a percent sign (%) which SAS normally interprets as macro triggers, but you want interpreted as text.

The %STR macro quoting function quotes apostrophes, parentheses and other special characters so they're interpreted as constant text. However, if you additionally want to mask macro triggers, such as % and &, then you need to use the %NRSTR function.

I've taken the example for %LET in Figure 7 and expanded it to include the use of macro quoting functions in Figure 8 below. In Figure 8, I create two new macro variables, T1 and T2. T1 contains an apostrophe. I used %STR to mask its value. Using %STR, you also need to place a percent sign (%) before the character you're quoting. Macro variable T2 contains a % sign and respectively I use %NRSTR to mask its value.

```
%Let Make = Audi;
%Let Model = 5000S;

%Let T1 = %STR(Vonstuben%'s Audi Dealership);
%Let T2 = %NRSTR(Invoice AS A % OF Retail Price);

%Put MAKE=&MAKE T1=&T1 T2=&T2;

Proc Print Data=SASHELP.CARS;
  Var Make Model Type DriveTrain InvoiceP MSRP PCT_MSRP;
  Where Make="&Make" and Model="&Model";
  Title1 "&T1";
  Title2 "&T2";
  Footnote "&Make &Model";
Run;
```

Figure 8. %LET example with %STR and %NRSTR macro quoting functions.

CALL SYMPUT VS. CALL SYMPUTX

An alternate method of creating a macro variable is CALL SYMPUT. CALL SYMPUT must be programmed in a DATA STEP. CALL SYMPUT is executed during DATA STEP processing. This contrasts with %LET which is processed during SAS program compilation. By default, CALL SYMPUT defines a global macro variable which can be referenced and called in any program in a SAS session.

Commonly, CALL SYMPUT is defined in a DATA _NULL_ step, where no SAS data set is created. By default, CALL SYMPUT preserves leading and trailing blanks in its macro variable value. One limitation on macro variables with CALL SYMPUT is that macro variables cannot be used in the same step they were defined in.

CALL SYMPUT allows macro variables to be based on a data set variable. Data set variables can be specified for both the macro variable name and value, permitting the creation of multiple macro variables. Figure 9 below provides an example of a DATA _NULL_ step with CALL SYMPUT.

```
DATA _NULL_;
     SET SDATASET1 NOBS=TOBS;
     CALL SYMPUT ('TOT_OBS', TOBS);
RUN;
```

Figure 9. DATA _NULL_ step with CALL SYMPUT.

CALL SYMPUTX has the same capabilities of CALL SYMPUT, for creation of macro variables during data step execution. In addition, CALL SYMPUTX has the impact of removing leading and trailing blanks from macro variable values. In Figure 10 below is an example SAS log excerpt showing the use of CALL SYMPUTX.

```
73         Data _Null_;
74            Set CARS (Obs=1);
75            Call Symputx ('Automake', Make);
76         Run;

NOTE: There were 1 observations read from the data set WORK.CARS.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds

78         %Put &Automake;
Acura
85
86         Proc Means Data=SASHELP.CARS Mean;
87           Var MSRP;
88           Class DriveTrain;
89           Where Make="&Automake";
90           Title1"Average Price for &Automake Models";
91           Title2"By Drivetrain";
92         Run;

NOTE: There were 7 observations read from the data set SASHELP.CARS.
      WHERE Make='Acura        ';
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.12 seconds
      cpu time            0.10 seconds
```

Figure 10. SAS log example using CALL SYMPUTX.

PROC SQL INTO CLAUSE

The PROC SQL INTO clause provides yet another method for creating macro variables. The INTO clause defines a global macro variable with leading and trailing blanks preserved in its value. The macro variable is defined and processed during the SAS compilation phase. Since the INTO clause is coded on a SELECT statement, the macro variable can be based on a SAS variable. Figure 11 provides an example of defining a macro variable using the PROC SQL INTO clause.

```
73         Proc Sql Inobs=1 Noprint;
74           Select Distinct Make Into: Make
75           From SASHELP.CARS;
76         Quit;

NOTE: PROCEDURE SQL used (Total process time):
       real time            0.00 seconds
       cpu time            0.01 seconds

84         Proc Means Data=SASHELP.CARS Mean;
85           Var MSRP;
86           Class DriveTrain;
87           Where Make="&Make";
88           Title1"Average Price for &Make Models";
89           Title2"By Drivetrain";
90         Run;

NOTE: There were 7 observations read from the data set SASHELP.CARS.
      WHERE Make='Acura       ';
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.13 seconds
      cpu time            0.11 seconds
```

Figure 11. SAS log example of PROC SQL INTO clause.

In addition, using the SEPARATED BY clause, you can store a concatenated list of values from a SAS variable separated by a delimiter in a macro variable. Figure 12 below provides an example of the INTO clause with SEPARATED BY.

```
Proc Sql;
     Select MSRP into : PRICELIST Separated By ‘ ‘
     From SASHELP.CARS;
Quit;
```

Figure 12. PROC SQL INTO clause with SEPARATED BY.

Another capability of the PROC SQL into clause is the ability to create multiple macro variables in a single SELECT statement.

OTHER METHODS

The %MACRO statement provides another way of defining and processing macro variables. The %MACRO statement signals the beginning of a macro definition or SAS macro. On the %MACRO statement, you can add macro parameters in parentheses, which are really macro variables. Macro parameters are local macro variables, and are only available during execution of the macro.

Macro parameters can be either keyword or positional parameters. For keyword parameters you must specify the macro variable name and its value in the %MACRO statement or macro call. Figure 13 below demonstrates the %MACRO statement with positional and keyword parameters in parentheses to define the macro TEST.

```
%MACRO TEST (VARLIST, DSN=SASHELP.CARS, LIB1=WORK);
   TITLE "DATASET: &DSN WITH VARIABLES &VARLIST, IN LIBRARY &LIB1;
%MEND TEST;
```

Figure 13. %MACRO statement with macro parameters.

The %GLOBAL and %LOCAL statements are another method which can be used to create macro variables of a specific scope. The limitation is they cannot assign values as the other methods do. They assign null values to macro variables. These statements can also be used to reset the scope of macro variables.

Table 3 below provides a comparison of the available methods to create macro variables. For each method, the table documents the scope of the macro variable, what phase of SAS processing its created, and whether the values preserve or remove leading and trailing blanks. There appear to be advantages and disadvantages to each method.

Method	Scope of Variable	Processing	Leading\Trailing Blanks
%LET	GLOBAL	COMPILATION	REMOVED
CALL SYMPUT\SYMPUTX	GLOBAL	EXECUTION	PRESERVED
PROC SQL INTO CLAUSE	GLOBAL	EXECUTION	PRESERVED
%MACRO STATEMENT	LOCAL	COMPILATION	N/A
%GLOBAL\%LOCAL	GLOBAL\LOCAL	COMPILATION	N/A

Table 3. Comparison of methods for defining macro variables.

%GLOBAL AND %LOCAL

In open code, %LET defines a global macro variable, which can be utilized in any SAS program or code during the session. However, in a macro definition, %LET creates a local macro variable, which can only be used within the macro.

In the example below in Figure 14, %LET statements are wrapped inside a macro definition for the macro AUTOS. The %LET statements define two macro variables, MAKE and MODEL. %PUT is used to validate the scope of the macro variables. %PUT _LOCAL_ is used to print a list of local macro variables to the log. The log below the code confirms that MAKE and MODEL are local macro variables pertaining to the macro AUTOS.

CODE

```
%Macro Autos;

   %Let Make = Audi;
   %Let Model = 5000S;

   %Put Make=&Make Model=&Model;

   Proc Print Data=SASHELP.CARS;
      Var Make Model Type DriveTrain MSRP;
      Where Make="&Make" and Model="&Model";
      Title "Type, Drivetrain, and MSRP &Make &Models Models";
   Run;

   %Put _LOCAL_;

%Mend Autos;

%Autos
```

LOG

```
AUTOS MAKE Audi
AUTOS MODEL 5000S
```

Figure 14. SAS code and log with %LET inside macro definition.

It's possible to change the scope of the macro variable created inside a macro definition. Using the %GLOBAL statement, you can redefine the scope of a macro variable from local to global. In this example, it changes the scope of the macro variable back to what it is by default.

In Figure 15, we revisit the same example we used for Figure 14, with the AUTOS macro. The macro AUTOS is essentially the same. We have added the %GLOBAL statement for the macro variables MAKE and MODEL. %GLOBAL creates MAKE and MODEL as global macro variables.

At the end we use %PUT _GLOBAL_ to print a list of global macro variables to the log. The log below validates that the macro variables MAKE and MODEL are now global macro variables, and displays their respective values.

SAS logs from the programs and examples in Figures 14-18 are provided at the end of the paper in the appendices.

CODE

```
%Macro Autos;

   %Global Make Model;

   %Let Make = Audi;
   %Let Model = 5000S;

   %Put Make=&Make Model=&Model;

   Proc Print Data=SASHELP.CARS;
        Var Make Model Type DriveTrain MSRP;
        Where Make="&Make" and Model="Model";
        Title "Type, Drivetrain, and MSRP &Make &Model Models";
   Run;

   %Put _GLOBAL_;

%Mend Autos;

%Autos
```

LOG

```
GLOBAL MAKE Audi
GLOBAL MODEL 5000S
```

Figure 15. SAS code and log for macro AUTOS using %GLOBAL.

In open code, CALL SYMPUT creates a global macro variable. Inside a macro, however, CALL SYMPUT will define a global macro variable as long as the macro doesn't have parameters. Creating a macro with parameters populates a local symbol table. Thus, if the macro has parameters, CALL SYMPUT will define local macro variables.

In the example in Figure 16, I define the macro VEHICLE. Inside the macro, using CALL SYMPUT in a DATA _NULL_ step, I've defined two macro variables, AUTOMAKE, and AUTOMODEL. Notice that the macro doesn't contain parameters.

At the end of the macro definition, there is a %PUT _GLOBAL_ statement which prints a list of global macro variables to the SAS log. The excerpt from the log below contains the list of global macro variables. The log confirms that AUTOMAKE and AUTOMODEL are global macro variables.

CODE

```
%Macro Vehicle;

  Data _Null_;
       Set SASHELP.Cars (Obs=1);
       Call Symput('Automake', Make);
       Call Symput('Automodel', Model);
  Run;

  %Put Automake=&Automake;
  %Put Automodel=&Automodel;

  Proc Means Data=SASHELP.CARS Mean;
       Var MSRP;
       Class DriveTrain;
       Where Make="&Automake" and Model="&Automodel";
       Title1"Average Price for &Automake &Automodel Models";
       Title2"By Drivetrain";
  Run;

  %Put _Global_;

%Mend Vehicle;

%Vehicle
```

LOG

```
GLOBAL AUTOMAKE Acura
GLOBAL AUTOMODEL MDX
```

Figure 16. SAS code and log for macro VEHICLE with CALL SYMPUT.

Similar to %GLOBAL, you can use the %LOCAL statement to change the scope of a macro variable from global to local. I can think of two examples where it would make sense to do this. Suppose you have another macro containing macro variables with the same name, and calls to this macro are nested inside your macro. With macro variable calls, you want to ensure global macro variables don't conflict with the local macro variables from the other macro. Take another scenario where global macro variables with the same names already exist. Resetting the scope to local prevents you from unintentionally overwriting the values of those variables.

In Figure 17, we revisit the VEHICLE macro example from Figure 16. A %LOCAL statement has been added for macro variables AUTOMAKE and AUTOMODEL at the beginning of the macro. A %PUT _LOCAL_ statement has been added at the end of the macro to print the values of any local macro variables in the SAS log.

Below the code, the log confirms that AUTOMAKE and AUTOMODEL have been recreated as local macro variables, along with their values. The macro variables are specific to the VEHICLE macro. In this example, adding parameters to the macro would also create AUTOMAKE and AUTOMODEL as local.

CODE

```
%Macro Vehicle;

  %Local Automake Automodel;

  Data _Null_;
       Set SASHELP.Cars (Obs=1);
       Call Symput('Automake', Make);
       Call Symput('Automodel', Model);
  Run;

  %Put Automake=&Automake;
  %Put Automodel=&Automodel;

  Proc Means Data=SASHELP.CARS Mean;
      Var MSRP;
      Class DriveTrain;
      Where Make="&Automake";
      Title1"Average Price for &Automake Models";
      Title2"By Drivetrain";
  Run;

  %Put _LOCAL_;

%Mend Vehicle;

%Vehicle
```

LOG

```
VEHICLE AUTOMAKE Acura
VEHICLE AUTOMODEL MDX
```

Figure 17. SAS code and log for macro VEHICLE using CALL SYMPUT with %LOCAL.

Similar to %LET and CALL SYMPUT, in open code PROC SQL INTO defines a global macro variable. Inside a macro without parameters, PROC SQL INTO still defines a global macro variable, the same scope as CALL SYMPUT. Using %LOCAL, you can redefine the scope of macro variables created with PROC SQL INTO.

In Figure 18 is an example utilizing the PROC SQL INTO clause. This time the SAS code is wrapped inside the AUTOMAKER macro. PROC SQL INTO creates the macro variables MAKE and MODEL. %LOCAL defines MAKE and MODEL as local macro variables, and we use %PUT _LOCAL_ to print their values in the log. When we run the AUTOMAKER macro, MAKE and MODEL are validated as local macro variables to the AUTOMAKER macro in the SAS log.

CODE

```
%Macro Automaker;

  %Local Make Model;

  Proc Sql Inobs=1 Noprint;
       Select Distinct Make, Model Into :Make, :Model
       From SASHELP.CARS;
  Quit;

  %PUT MAKE=&MAKE;
  %PUT MODEL=&MODEL;

  Proc Means Data=SASHELP.CARS Mean;
    Var MSRP;
    Class DriveTrain
    Where Make="&Make" and Model="&Model";
    Title1"Average Price for &Make &Model";
    Title2"By Drivetrain";
  Run;

  %Put _LOCAL_;

%Mend Automaker;

%Automaker
```

LOG

```
AUTOMAKER MAKE Acura
AUTOMAKER MODEL MDX
```

Figure 18. Using %LOCAL with PROC SQL INTO in AUTOMAKER macro.

COMPARISON

Table 4 below provides a summary comparison of the methods I've demonstrated and reviewed based on scope, and on the place where it's used, in open code or in a macro. Although all three techniques create a global macro variable by default (in open code). However, the scope of the macro variable inside a macro definition depends on the technique which is utilized.

Notice when used in a macro with parameters, all three SAS constructs create local macro variables. This occurs because SAS populates a local symbol table. The table also documents that the macro variables scope for each of methods can be reset using %LOCAL or %GLOBAL. Although not included in the table, the %MACRO statement always defines local macro variables for macro parameters. Macro parameters can be reset to global using %GLOBAL.

Method	Scope in Open-Code	Scope inside Macro	Scope Inside Macro with Parameters	Scope Inside Macro with %LOCAL or %GLOBAL
%LET	**GLOBAL**	**LOCAL**	**LOCAL**	**GLOBAL**
CALL SYMPUT\X	**GLOBAL**	**GLOBAL**	**LOCAL**	**LOCAL**
PROC SQL INTO	**GLOBAL**	**GLOBAL**	**LOCAL**	**LOCAL**

Table 4. Comparison of methods for defining macro variables according to scope.

CONCLUSION

Macro variables are the core and essential part of the SAS macro facility. Utilizing them makes your code more portable, dynamic and efficient. Becoming familiar with the different methods to create macro variables and their capabilities, makes your code more versatile, and enhances your SAS skill set. Macro variable scope is determined by the method used, but also depends on where the construct is placed in your code. Learning the consequences of the different methods according to scope helps you build better programs which are more robust.

REFERENCES

Dominic, Littish. ”CALL SYMPUT-Global or Local” Pharmaceutical Users Software Exchange (PhUSE) 2009 Conference. https://www.lexjansen.com/phuse/2009/po/PO15.pdf

Horstman, Joshua M. “Using Macro Variable Lists to Create Dynamic Data-Driven Programs.” Midwest SAS® Users Group (MWSUG) 2019 Conference. https://www.lexjansen.com/mwsug/2019/SP/MWSUG-2019-SP-053.pdf

Burlew, Michelle M. SAS® Macro Programming Made Easy, Third Edition. 2014. Cary, NC. SAS Institute Inc.

Repole, Warren. SAS® Macro Language Course Notes. 1998 Cary, NC. SAS Institute Inc.

ACKNOWLEDGMENTS

The author would like to thank Lisa Mendez, WUSS 2022 Academic Chair, Tasha Chapman, WUSS 2022 Academic Chair, Isaiah Lankham and Matthew Slaughter, E-Poster Section Co-chairs, and the WUSS Executive Committee and Conference Team for accepting my abstract and paper and for organizing this conference.

CONTACT INFORMATION

Your comments and questions are valued and encouraged.

Contact the author at:

Jay Iyengar
Data Systems Consultants LLC
datasyscon@gmail.com
https://www.linkedin.com/in/datasysconsult/

Jay Iyengar is director of Data Systems Consultants LLC. He is a SAS consultant, trainer, and SAS Certified Advanced Programmer. He’s been co-leader and organizer of the Chicago SAS Users Group (WCSUG) for the last 5 years. He’s presented papers and training seminars at SAS Global Forum (SGF), Midwest SAS Users Group (MWSUG), Wisconsin Illinois SAS Users Group (WIILSU), Northeast SAS Users Group (NESUG), and Southeast SAS Users Group (SESUG) conferences. He has been using SAS since 1997. His industry experience includes International Trade, Health care, Database Marketing and Educational Testing.

TRADEMARK CITATION

APPENDIX I

```
1          OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
 SYMBOLGEN:  Macro variable _SASWSTEMP_ resolves to
/folders/myfolders/.sasstudio/.images/b8196986-c174-4b45-9b7c-a68bcb2676f1
 SYMBOLGEN:  Some characters in the above value which were subject to macro
quoting have been unquoted for printing.
 SYMBOLGEN:  Macro variable GRAPHINIT resolves to
 72
 73         Options Symbolgen;
 74
 75         /***************************/
 76         /* %LET */
 77         /* OPEN CODE - GLOBAL    */
 78         /* MACRO DEF - LOCAL     */
 79         /* MACRO W/%GLOBAL - GLOBAL */
 80         /**/
 81         /***************************/
 82
 83         %Macro Autos;
 84          %Global Make Model;
 85          %Let Make = Audi;
 86          %Let Model = 5000S;
 87          %Put Make=&Make Model=&Model;
 88
 89          Proc Print Data=SASHELP.CARS;
 90           Var Make Model Type DriveTrain MSRP;
 91           Where Make="&Make";
 92           Title "Type, Drivetrain, and MSRP&Make Models";
 93          Run;
 94
 95          %Put _GLOBAL_;
 96         %Mend Autos;
 97
 98         %Autos
 SYMBOLGEN:  Macro variable MAKE resolves to Audi
 SYMBOLGEN:  Macro variable MODEL resolves to 5000S
 Make=Audi Model=5000S
 SYMBOLGEN:  Macro variable MAKE resolves to Audi
 SYMBOLGEN:  Macro variable MAKE resolves to Audi

NOTE: There were 19 observations read from the data set SASHELP.CARS.
       WHERE Make='Audi';

NOTE: PROCEDURE PRINT used (Total process time):
       real time           0.23 seconds
       cpu time            0.23 seconds
```

```
 GLOBAL AUTOMAKE Acura
 GLOBAL AUTOMODEL  MDX
 GLOBAL CLIENTMACHINE 10.0.2.2
 GLOBAL GRAPHINIT
 GLOBAL GRAPHTERM
 GLOBAL MAKE Audi
 GLOBAL MODEL 5000S
 GLOBAL OLDPREFS foldersmyfolders/.wepreferences
 GLOBAL OLDSNIPPETS foldersmyfolders/.mysnippets
 GLOBAL OLDTASKS foldersmyfolders/.mytasks
 GLOBAL SASWORKLOCATION "/tmp/SAS_workE71F00004EC0_localhost
.localdomain/SAS_workA74200004EC0_localhost. localdomain/"
 GLOBAL STUDIODIR foldersmyfolders/.sasstudio
 GLOBAL STUDIODIRNAME .sasstudio
 GLOBAL STUDIOPARENTDIR foldersmyfolders
 GLOBAL SYSCASINIT 0
 GLOBAL SYSSTREAMINGLOG true
 GLOBAL SYSUSERNAME sasdemo
 GLOBAL USERDIR foldersmyfolders
 GLOBAL _BASEURL http:localhost:10080SASStudio
 GLOBAL _CLIENTAPP 'SAS Studio'
 GLOBAL _CLIENTAPPABREV Studio
 GLOBAL _CLIENTAPPVERSION 3.8
 GLOBAL _CLIENTMACHINE 10.0.2.2
 GLOBAL _CLIENTMODE basic
 GLOBAL _CLIENTUSERID sasdemo
 GLOBAL _CLIENTUSERNAME sasdemo
 GLOBAL _CLIENTVERSION 3.8
 GLOBAL _EXECENV SASStudio
 GLOBAL _MACRO_FOUND 0
 GLOBAL _SASHOSTNAME localhost
 GLOBAL _SASPROGRAMFILE foldersmyfoldersMacrov_PaperExample1_Let.sas
 GLOBAL _SASPROGRAMFILEHOST localhost
 GLOBAL _SASSERVERNAME localhost
 GLOBAL _SASWORKINGDIR /opt/sasinside/SASConfig/Lev1/SASApp

GLOBAL _SASWSTEMP_
foldersmyfolders.sasstudio.imagesb8196986c1744b459b7ca68bcb2676f1

GLOBAL _SASWS_ foldersmyfolders
 99
 100
 101
 102
 103        OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
 SYMBOLGEN:  Macro variable GRAPHTERM resolves to
 115
```

APPENDIX II

```
1          OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
 SYMBOLGEN:  Macro variable _SASWSTEMP_ resolves to
/folders/myfolders/.sasstudio/.images/29141e25-a5c9-4a70-9869-91218b513002

 SYMBOLGEN:  Some characters in the above value which were subject to macro
quoting have been unquoted for printing.

 SYMBOLGEN:  Macro variable GRAPHINIT resolves to
 72
 73         Options Symbolgen;
 74
 75         /*************************/
 76         /* Call Symput      */
 77         /*  */
 78         /* OPEN CODE - Global  */
 79         /* MACRO DEF - Global  */
 80         /* MACRO W/%LOCAL - Local */
 81         /*  */
 82         /*************************/
 83
 84         %Macro Vehicle;
 85         %Local Automake Automodel;
 86
 87         Data _Null_;
 88            Set SASHELP.Cars (Obs=1);
 89            Call Symput('Automake', Make);
 90            Call Symput('Automodel', Model);
 91         Run;
 92
 93         %Put Automake=&Automake;
 94         %Put Automodel=&Automodel;
 95
 96         Proc Means Data=SASHELP.CARS Mean;
 97           Var MSRP;
 98           Class DriveTrain;
 99           Where Make="&Automake";
 100          Title1"Average Price for &Automake Models";
 101          Title2"By Drivetrain";
 102        Run;
 103
 104        %Put _LOCAL_;
 105
 106        %Mend Vehicle;
 107
 108        %Vehicle
```

```
NOTE: There were 1 observations read from the data set SASHELP.CARS.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

SYMBOLGEN:  Macro variable AUTOMAKE resolves to Acura
Automake=Acura
SYMBOLGEN:  Macro variable AUTOMODEL resolves to  MDX
Automodel= MDX
SYMBOLGEN:  Macro variable AUTOMAKE resolves to Acura
SYMBOLGEN:  Macro variable AUTOMAKE resolves to Acura
NOTE: There were 7 observations read from the data set SASHELP.CARS.
      WHERE Make='Acura        ';
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.18 seconds
      cpu time            0.17 seconds

VEHICLE AUTOMAKE Acura
VEHICLE AUTOMODEL  MDX
109
110
111
112        OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
SYMBOLGEN:  Macro variable GRAPHTERM resolves to
124
```

APPENDIX III

```
1          OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
72
73         Options Symbolgen;
74
75         /*************************/
76         /* Proc SQL Into Clause  */
77         /*  */
78         /* OPEN CODE - Global  */
79         /* MACRO DEF - Global  */
80         /* MACRO W/%LOCAL - Local */
81         /*  */
82         /*************************/
83
84         %Macro Automaker;
85
86         %Local Make Model;
87
88         Proc Sql Inobs=1 Noprint;
89           Select Distinct Make, Model Into :Make, :Model
90           From SASHELP.CARS;
91         Quit;
92
93         %PUT MAKE=&MAKE;
94         %PUT MODEL=&MODEL;
95
96         Proc Means Data=SASHELP.CARS Mean;
97           Var MSRP;
98           Class DriveTrain;
99           Where Make="&Make";
100          Title1"Average Price for &Make Models";
101          Title2"By Drivetrain";
102        Run;
103
104        %Put _LOCAL_;
105
106        %Mend Automaker;
107
108        %Automaker
WARNING: Only 1 records were read from SASHELP.CARS due to INOBS= option.
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

SYMBOLGEN:  Macro variable MAKE resolves to Acura
MAKE=Acura
SYMBOLGEN:  Macro variable MODEL resolves to  MDX
MODEL= MDX
```

```
SYMBOLGEN:  Macro variable MAKE resolves to Acura
SYMBOLGEN:  Macro variable MAKE resolves to Acura
NOTE: There were 7 observations read from the data set SASHELP.CARS.
      WHERE Make='Acura        ';
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.18 seconds
      cpu time            0.18 seconds

AUTOMAKER MAKE Acura
AUTOMAKER MODEL  MDX
AUTOMAKER SQLEXITCODE 0
AUTOMAKER SQLOBS 1
AUTOMAKER SQLOOPS 17
AUTOMAKER SQLRC 4
AUTOMAKER SQLXOBS 0
AUTOMAKER SQLXOPENERRS 0
109
110
111
112        OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
SYMBOLGEN:  Macro variable GRAPHTERM resolves to
124
```