Paper SAS3511-2019

SAS® Macros: Beyond the Basics

Ron Coleman, SAS Institute Inc., Cary, NC

ABSTRACT

Basic macros rely on symbolic substitution to place values in particular locations in SAS® code—they simply create code. Beyond that, the sky is the limit! We cover the following advanced techniques: create and populate macro variables; read a list of file names from a folder and use the list for processing code; utility macros that calculate new values without the need for a DATA step; and advanced macro functions. We also include a discussion about macro quoting. Some pointers for passing values when you are using SAS/CONNECT® and SAS® Grid are also discussed.

INTRODUCTION

Macro programmers generally accept that macros create SAS code. The simplest use is to assign a value to a macro variable and then have the value resolve to create code:

```
%let table = sashelp.gnp;
proc print data=&table;
run;
```

Another form of this basic use is to wrap the PROC PRINT step within a macro definition and either use the initial %LET statement to define the value or use a macro argument:

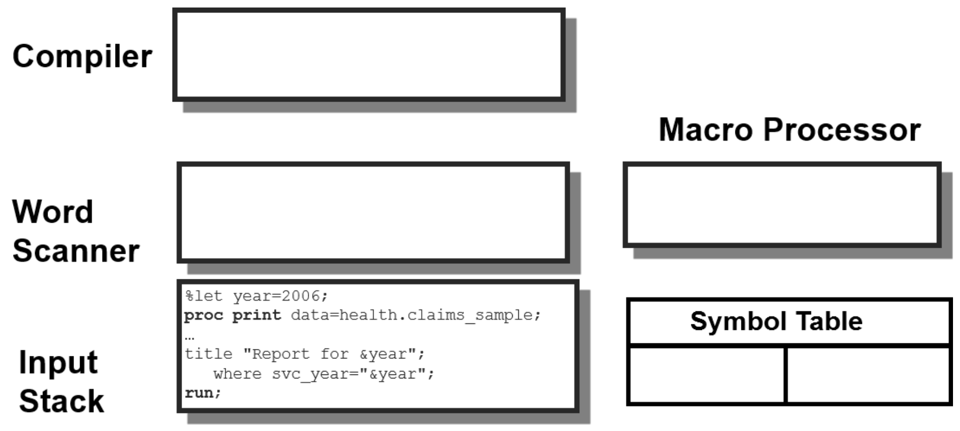
```
%macro myprint(table);
proc print data=&table;
run;
%mend;
%myprint(sashelp.gnp)
```

Beyond these basic, and most commonly used examples there lies a world of macro conditional processing (%IF … %THEN … %ELSE); iterative processing (%DO, %DO WHILE, %DO UNTIL); macro functions; using SAS functions; and a load of other possibilities that are only limited by your imagination.

MACRO VARIABLE SCOPE

Macro variable scope refers to where a macro variables' value can be seen. Especially when multiple macro definitions are used and executed, it can be very frustrating to track down why a variable is not resolving where it should.

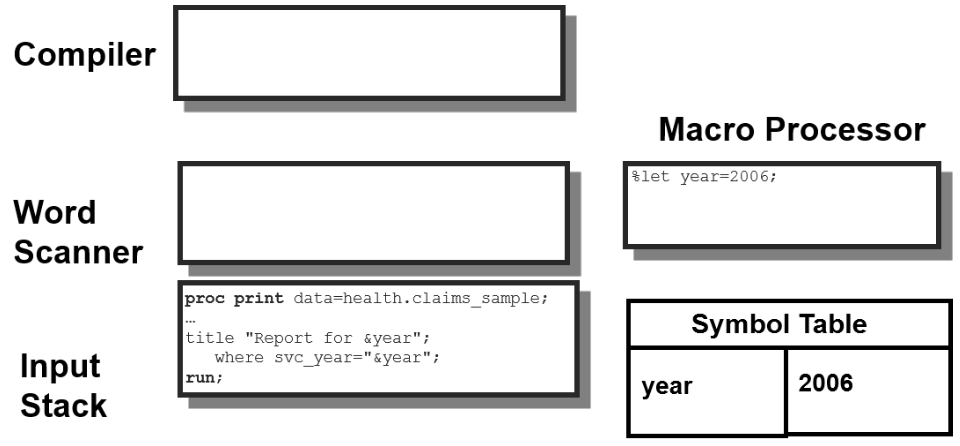
The macro compiler has a system of symbol tables that store the values for each macro variable that is used in a program. Let's look at how SAS uses the macro compiler to create SAS code:



```
%let year=2006;
proc print data=health.claims_sample;
…
title "Report for &year";
   where svc_year="&year";
run;
```

Figure 1. Initial Macro Processing

The SAS code is in the input stack and the first line is a macro statement. The word scanner recognizes it and passes the line to the macro processor.

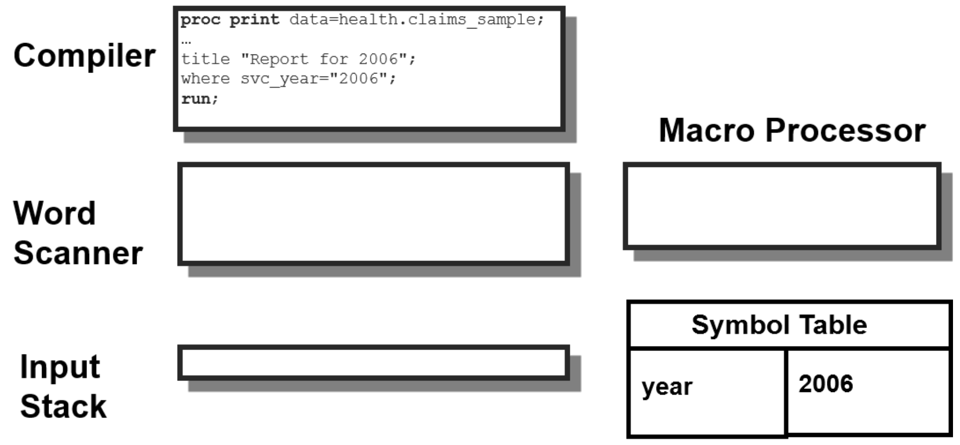


```
%let year=2006;
```

```
proc print data=health.claims_sample;
…
title "Report for &year";
   where svc_year="&year";
run;
```

Figure 2. Creating a Value in the Symbol Table

The macro processor creates an entry in the symbol table for the macro variable Year and assigns a value to it. Now, when there is a macro reference to &year, the macro processor substitutes the correct value in the created code.



```
proc print data=health.claims_sample;
…
title "Report for 2006";
where svc_year="2006";
run;
```

Figure 3. Macro Variables Resolved

The scope of a macro variable is either global or local. Global scope means that the variable definition and value are stored in the global symbol table and can be used or updated by any process in a SAS session.

Local scope is limited to a defined macro that is executing that has its own private symbol table. This prevents macros from overwriting macro variables of the same name that are used in other macros. Think about index variables like ‘I’ that could be used in multiple places. We don’t want another process to overwrite the value in our macro.

The specific scope of a macro variable can be set using a %GLOBAL or %LOCAL macro statement to control where the value can be used.

ASSIGNING A MACRO VARIABLE VALUE AT EXECUTION

Creating and assigning macro variables during the execution of a SAS program is a very powerful way to create dynamic, data-driven programs. Process all *.log files in a folder – simple! Extract values from a DBMS table to determine the number of iterations for an iterative process – easy! Calculate beginning and end dates based on the current date – deceptively simple. Let’s start there!

USING NORMAL SAS FUNCTIONS IN MACRO PROCESSING

Some of the magic in advanced macro processing comes from the ability to use normal SAS functions, sometimes referred to as DATA step functions. In additonal, there is a decided advantage to being able to calculate values outside of a DATA step, because adding another step to the process wastes resources and causes the program to be less efficient.

We can simply use the %SYSFUNC macro function to execute those DATA step functions for us. The syntax is pretty simple:

%SYSFUNC(*function*(*argument(s)*)<, *format*>)

The first argument is the DATA step function to use with any required arguments. The second, optional, argument is a format to apply to the output value that is returned. Here is an example using the TODAY function:

```
36         %put %sysfunc(today());
21595
37         %put %sysfunc(today(),date9.);
15FEB2019
```

Output 1. %SYSFUNC Function Example

A %LET statement could have been used to assign the output values to a macro variable if needed.

ASSIGNING VALUES USING A DATA STEP

Use a DATA step to calculate values and place them into macro variables using the CALL SYMPUTX function. Here is the syntax:

CALL SYMPUTX(*macro-variable*, *value* <, *symbol-table*>);

Macro-variable can be a SAS name enclosed in quotation marks, a SAS variable, or a character expression that produces a SAS name.

Value is the numeric or character value to assign to the macro variable.

The value of *Symbol-table* is either the default value ‘G’ for the global symbol table or ‘L’ for the local symbol table.

Assigning the current data to macro variables using a DATA step and the CALL SYMPUTX function might look like this:

```
           data _null_;
            call symputx('date',today());
            call symputx('date_fmt',put(today(),date9.));
           run;
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds
           %put &date;
21595
           %put &date_fmt;
15FEB2019
```

Output 2. Using CALL SYMPUTX in a DATA Step

Note that we had to use a PUT function to apply the format to the date!

ASSIGNING VALUES WITH THE INTO OPERATOR IN PROC SQL

PROC SQL has an operator, INTO, that puts the resulting values from a query into one or more macro variables. Using the INTO operator creates a macro variable array.

Creating a Vertical Macro Variable Array

A macro variable array is where there are macro variables that are similarly named, such as Var1, Var2, Var3, and so on. These can be processed as shown in a later example.

We create a simple macro array using the distinct values of the variable Origin from the Sashelp.Cars table:

```
           proc sql noprint;
            select distinct origin into :origin1-:origin10
            from sashelp.cars
            ;
           quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
           %put _global_;
GLOBAL ORIGIN1 Asia
GLOBAL ORIGIN2 Europe
GLOBAL ORIGIN3 USA
GLOBAL SQLOBS 3
```

Output 3. Creating a Macro Variable Array

Using the INTO operator places the results of the query into macro variables. Note that the macro variables are preceded by a colon. Although there were 10 variables defined, the INTO operator only creates the number of macro variables needed by the results.

PROC SQL also creates a variable called &SQLOBS, included in the output above, that shows how many rows were returned from the last query so that you know the number of rows to process.

Creating a Horizontal Macro Variable Array

A horizontal array has multiple values stored in a single macro variable. The values are separated by a delimiter value. Entire presentations can be done using this technique alone.

The syntax is similar to a vertical array. Here is an example.

```
36         proc sql noprint;
37          select distinct origin into :origins separated by ' '
38          from sashelp.cars
39          ;
40         quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
41         %put &origins;
Asia Europe USA
42         %put &sqlobs;
3
```

Output 4. Creating a Horizontal Array

Note that there is only one macro variable created that contains all 3 values.

Creating a Comma-Delimited List from the INTO Operator

We can choose the delimiter to separate the values and create a comma-delimited list of values:

```
36         proc sql noprint;
37          select distinct origin into :origins separated by '", "'
38          from sashelp.cars
39          ;
40         quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
41         %put "&origins";
"Asia", "Europe", "USA"
42         %put &sqlobs;
3
```

Output 5. Creating a Comma-Delimited List

Note that the distinct values are separated by the double-quotes and comma, so when I want the list to resolve properly the macro variable had to be enclosed in double-quotes as well: “&origins”.

From this point we can use that macro array in code:

```
45         proc sql;
46          select distinct origin, make
47          from sashelp.cars
48          where origin in ("&origins")
SYMBOLGEN:  Macro variable ORIGINS resolves to Asia", "Europe", "USA
49          ;
50         quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.03 seconds
      cpu time            0.00 seconds
```

Output 6. Using a macro array in code.

USING ADVANCED MACROS AND DATA STEP FUNCTIONS FOR EFFICIENCY

I write a lot of code that parses log files including SAS logs, Metadata Server logs, Object Spawner logs, grid system logs, command output logs, and many others. In most cases, there are multiple log files in a folder (directory) that need to be processed, so I created a simple macro that reads all the file names in a folder and creates a vertical macro array of the names. The name of the macro is ReadDirectory and I include the code in the appendix.

This advanced macro doesn't create any SAS code, it just gathers values into macro variables that are used in later processing!

VALIDATING INPUT VALUES

The first thing needed in most macros is to validate important input values so that a user who calls the macro and receives invalid values does not get a generic error but a real error message that tells them what is wrong. Things like file names, folders, SAS tables can all be checked to be sure they exist.

In the ReadDirectory macro, the only argument is the name of the folder to search, so we validate that the folder actually exists before we try to use it. Here is the code to check and provide a real WARNING message if needed:

```
   %macro readdirectory(folder);
      /* Verify the folder exists */
      %let rc = %sysfunc(fileexist(&folder));
      %if &rc = 0 %then
      %do;
            %put WARNING: Folder &folder does not exist!;
            %return;
      %end;
```

The %RETURN statement tells the macro to end.

READING THE CONTENTS OF A FOLDER

Reading the files in a folder is very easy using some basic SAS functions. This uses the DOPEN function to open the directory, the DNUM function to identify how many items are in the folder, the DREAD function to read each file name, and the DCLOSE function to close the directory. This last step is very important and should never be overlooked!

This example uses a FILENAME statement to assign the folder and then DOPEN to open it:

```
      %let fileref = _readdir;
      filename &fileref "&folder";
      run;
      %let did = %sysfunc(dopen(&fileref));
      %put NOTE: Directory ID is: &did;
      %if &did <= 0 %then
      %do;
            %put WARNING: Could not open folder &folder;
            %return;
      %end;
```

Then we get the file count and loop through each file name creating a global macro variable array of the names (&file1, &file2,…, &file*n*).

```
      %global _dcount;
      %let _dcount = %sysfunc(dnum(&did));
      %put NOTE: &_dcount files found in folder;
      %do i = 1 %to &_dcount;
         %let file = %sysfunc(dread(&did,&i));
         %global file&i;
         %let file&i=&file;
      %end;
      %let did = %sysfunc(dclose(&did));
      filename &fileref clear;
```

```
71         %readdirectory(E:\Projects\Demo Data\BTB Macros\iotest);
NOTE: Directory ID is: 1
NOTE: 3 files found in folder
NOTE: Fileref _READDIR has been deassigned.
72
73         %put _global_;
GLOBAL FILE1 rhel_iotest_sas.results
GLOBAL FILE2 rhel_iotest_sasdata.results
GLOBAL FILE3 rhel_iotest_saswork.results
```

Output 7: Reading the Contents of a Folder

CALCULATING DATE RANGES

Quite often, processing needs to be done for the previous 3, 6, or even 12 months, so a utility macro fits the bill to do these calculations:

```
%macro computeDates(monthsBackBegin, monthsBackEnd, today=);
      %global prevDateBegin prevDateEnd;
      %if &today = %then %let tday = %sysfunc(today());
      %else %let tday = &today;
      %let prevDateBegin = %sysfunc(intnx(month,&tday,&monthsBackBegin));
      %let prevDateEnd   = %sysfunc(intnx(month,&tday,&monthsBackEnd,end));
      %put &tday &prevDateBegin &prevDateEnd;
      %put %nrstr(        Today:) %sysfunc(putn(&tday,date9.));
      %put %nrstr(PrevDateBegin:) %sysfunc(putn(&prevDateBegin,date9.));
      %put %nrstr(   PrevDateEnd:) %sysfunc(putn(&prevDateEnd,date9.));
%mend;
```

This macro creates global macro variables for the beginning and ending dates for the desired period as well as the ability to specify the base date of those calculations.

```
36          /* Previous 3 months from current date */
37          %computeDates(-3,-1)
21614 21519 21608
        Today: 06MAR2019
PrevDateBegin: 01DEC2018
  PrevDateEnd: 28FEB2019
38          /* Previous 6 months from a specified date */
39          %computeDates(-6,-1,today='01jan2019'd)
'01jan2019'd 21366 21549
        Today: 01JAN2019
PrevDateBegin: 01JUL2018
  PrevDateEnd: 31DEC2018
```

Output 8: Calculating Date Ranges

PROTECTING SPECIAL CHARACTERS WITH MACRO QUOTING

Macro quoting of special characters is needed to protect our processing and prevent errors. A special character is any character that is not part of a valid SAS name, which includes only letters, numbers, and underscores. Here is the list of special characters that might need to be quoted, depending on usage:

```
& % ' " ( ) + − * / < > = ¬ ^ ~ ; , #  blank
     AND OR NOT EQ NE LE LT GE GT IN
```

There are 3 major quoting functions:

%STR and %NRSTR	Mask special characters and mnemonic operators in **constant text at macro compilation.**
%BQUOTE and %NRBQUOTE	Mask special characters and mnemonic operators in a **resolved value at macro execution.**
%SUPERQ	Masks all special characters and mnemonic operators **at macro execution but prevents further resolution of the value.**

Table 1: Macro Quoting Functions

%STR: PROTECTING TEXT AT MACRO COMPILATION

Using an example of sending macro values to a SAS/CONNECT session, we can see why protection is necessary. We first connect to our server:

```
signon cnxn1;
```

```
NOTE: AUTOEXEC processing completed.

NOTE: Remote signon to CNXN1 complete.
```

Output 9: Connecting to remote server

Now we want to send a specific path value to the remote session to identify where the output should be created, so we use the %SYSLPUT macro statement to create a new macro variable in the remote session:

```
%syslput path=/shared/data;

rsubmit cnxn1;
   %put NOTE: Path = &path;
   ods pdf file="&path/RonConnectDemo.pdf"
            style=statistical;
```

This code produces this SAS log showing an error in %SYSLPUT:

```
129  %syslput path=/shared/data / remote=cnxn1;
ERROR: Unrecognized option to the %SYSLPUT statement.
130
131  rsubmit cnxn1;
NOTE: Remote submit to CNXN1 commencing.
1        %put NOTE: Path = &path;
WARNING: Apparent symbolic reference PATH not resolved.
NOTE: Path = &path
2        ods pdf file="&path/RonConnectDemo.pdf"
WARNING: Apparent symbolic reference PATH not resolved.
3                 style=statistical;
NOTE: Writing ODS PDF output to DISK destination
      "/shared/sas/sasconfig/compute/Lev1/SASApp/&path/RonConnectDemo.pdf", printer "PDF".
NOTE: Remote submit to CNXN1 complete.
```

Output 10:

The error was caused by the special character '/', and because this is explicit code we can use the %STR macro quoting function to mask the issue.

```
%syslput path=%str(/shared/data);

rsubmit cnxn1;
   %put NOTE: Path = &path;
   ods pdf file="&path/RonConnectDemo.pdf"
            style=statistical;
```

```
132  %syslput path=%str(/shared/data);
133
134  rsubmit cnxn1;
NOTE: Remote submit to CNXN1 commencing.
4        %put NOTE: Path = &path;
NOTE: Path = /shared/data
5        ods pdf file="&path/RonConnectDemo.pdf"
6                 style=statistical;
NOTE: ODS PDF printed no output.
      (This sometimes results from failing to place a RUN statement before the ODS PDF CLOSE
      statement.)
NOTE: Writing ODS PDF output to DISK destination "/shared/data/RonConnectDemo.pdf", printer
"PDF".
NOTE: Remote submit to CNXN1 complete.
```

Output 11:

Having protected the special characters, we have success in passing the macro value.

%BQUOTE: PROTECTING TEXT AT MACRO EXECUTION

Next, we will look at passing the value from a local macro variable, which means the value must be resolved at execution. First, try using %STR as earlier:

```
%let mypath=/shared/data;
%syslput path=%str(&mypath);

rsubmit cnxn1;
   %put NOTE: Path = &path;
   ods pdf file="&path/RonConnectDemo.pdf"
            style=statistical;
```

```
137  %let mypath=/shared/data;
138  %syslput path=%str(&mypath);
ERROR: Unrecognized option to the %SYSLPUT statement.
139
140  rsubmit cnxn1;
NOTE: Remote submit to CNXN1 commencing.
1        %put NOTE: Path = &path;
WARNING: Apparent symbolic reference PATH not resolved.
NOTE: Path = &path
2        ods pdf file="&path/RonConnectDemo.pdf"
WARNING: Apparent symbolic reference PATH not resolved.
3                 style=statistical;
NOTE: Writing ODS PDF output to DISK destination
      "/shared/sas/sasconfig/compute/Lev1/SASApp/&path/RonConnectDemo.pdf", printer "PDF".
NOTE: Remote submit to CNXN1 complete
```

Output 12:

The same error we had before of an ‘Unrecognized option to the %SYSLPUT statement’ shows us that %STR did not protect the special characters properly. This is a case where we need %BQUOTE to protect a value at macro execution:

```
%let mypath=/shared/data;
%syslput path=%bquote(&mypath);
```

```
rsubmit cnxn1;
   %put NOTE: Path = &path;
   ods pdf file="&path/RonConnectDemo.pdf"
            style=statistical;
```

```
141  %let mypath=/shared/data;
142  %syslput path=%bquote(&mypath);
143
144  rsubmit cnxn1;
NOTE: Remote submit to CNXN1 commencing.
4        %put NOTE: Path = &path;
NOTE: Path = /shared/data
5        ods pdf file="&path/RonConnectDemo.pdf"
6                 style=statistical;
NOTE: ODS PDF printed no output.
      (This sometimes results from failing to place a RUN statement before the ODS PDF CLOSE
      statement.)
NOTE: Writing ODS PDF output to DISK destination "/shared/data/RonConnectDemo.pdf", printer
"PDF".
NOTE: Remote submit to CNXN1 complete.
```

Output 13:

Problem solved with the %BQUOTE function!

%SUPERQ: PROTECTING UNUSUAL THINGS

Messages from a database, via a SAS/ACCESS LIBNAME statement or pass-through queries are best handled using %SUPERQ, as they can sometimes contain special characters. Values that are processed with %SUPERQ are usually not intended to receive further processing, as the quoting stays with the value unless specifically removed by the %UNQUOTE function.

When using %SUPERQ, the macro argument that is passed is the name of the macro variable with no ampersand. Here's an example:

```
%put %superq(SYSDBMSG);
```

CONCLUSION

As you can see, there is a lot more to the SAS Macro Language than macro variables, symbolic substitution, and building SAS code. I hope you continue to learn more of what macros can do and then build new macros to accomplish more and more tasks to make you more productive.

APPENDIX

This is the complete ReadDirectory macro definition:

```
%macro readdirectory(folder);
      /* Verify the folder exists */
      %let rc = %sysfunc(fileexist(&folder));
      %if &rc = 0 %then
      %do;
            %put WARNING: Folder &folder does not exist!;
            %return;
      %end;
      %let fileref = _readdir;
      filename &fileref "&folder";
      run;
      %let did = %sysfunc(dopen(&fileref));
      %put NOTE: Directory ID is: &did;
      %if &did <= 0 %then
      %do;
            %put WARNING: Could not open folder &folder;
            %return;
      %end;

      %global _dcount;
      %let _dcount = %sysfunc(dnum(&did));
      %put NOTE: &_dcount files found in folder;

   %do i = 1 %to &_dcount;
      %let file = %sysfunc(dread(&did,&i));
            %global file&i;
      %let file&i=&file;
      %end;

      %let did = %sysfunc(dclose(&did));
      filename &fileref clear;

%mend readdirectory;
```

CONTACT INFORMATION

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

Ron Coleman
HLS Customer Advisory
SAS Institute Inc.
Ron.Coleman@sas.com