

An Introduction to SAS® Macro

H. Ian Whitlock, Westat Inc.

Introduction

We will answer the following questions about SAS macro using examples.

- What is it?
- Why use it?
- How do you use it?

The paradigm of SAS macro thinking is introduced with minimal intrusion from syntax.

The intended audience may have no special knowledge of SAS macro or they might be familiar with the macro syntax but do not naturally think in SAS macro. Significant experience with basic SAS programming will be helpful for understanding the problems presented. Even the macro sophisticate has something to gain in philosophy, because the emphasis on the various phases of SAS processing and how they influence each other is not usually presented in discussions of SAS macro.

What Is It?

The Macro Facility is a processor, i.e. a "machine" that transforms code into a final form for presentation to the SAS compiler.

The Macro Language is a programming language. The programs manipulate character strings of SAS code.

INPUT:

MACRO (instructions)
Partial SAS code (data).

OUTPUT:

Complete SAS code (data).

A significant part of programming is the art of making decisions and knowing when to make them. Macro provides the ability to make compile time decisions.

To emphasize different decision times, suppose some data are stored in the SAS data set IN.DAYS and we need to write a program to access the data for a particular day of the week. On weekdays we want the data for Sunday and on weekends we want the data for Friday.

For weekdays we might make an execution time decision using the DATA step code:

```
SET IN.DAYS ;
```

```
IF WEEKDAY = 1 ;
```

We could make this a coding time decision by splitting the file into subsets IN.DAY1 and IN.DAY6. Then a coding time decision would use the DATA step code:

```
SET IN.DAY1 ;
```

For a compile time decision we would like on weekdays, to show the compiler

```
SET IN.DAY1 ;
```

and on weekends to show it

```
SET IN.DAY6 ;
```

But this means we must be able to manipulate the code like data. Compile time decisions are often implemented by hiring a junior programmer to edit the code just before running the program. SAS macro provides an automated solution, since it allows one to write instructions for modifying SAS code just before compiling.

The macro concept is almost as old as programming languages! Why then is its effect on SAS programming so much more dramatic than on traditional languages?

Traditional languages have only one compile time, hence only one time for macro processing. In SAS, however, each step is compiled and executed before the next step is considered, hence there are many "compile times" in a SAS program. It is the intertwining of compile and execute steps in SAS that presents a much richer structure for macro opportunities than traditional languages can provide.

WHY USE IT?

Macro can be used to:

- Turn system information into code.
- Isolate common source code.
- Parameterize code.
- Package code.
- Perform conditional execution of steps.
- Generate repetitive code.
- Communicate between steps.
- Write code that changes with the data.
- Write reusable code.
- Control system design.

The remainder of this paper is a series of examples demonstrating a "how" corresponding to each "why".

System Information - Example 1

Write a SAS program to put today's date in the title of a database report on SESUG speakers. For example,

```
SESUG Speakers as of May 15, 1994
```

But the date should be whatever day the program is run on the database of speakers. This code does the job.

```
TITLE "SESUG Speakers as of &SYSDATE";
```

Macro variables come in two types:

Automatic variables created by the SAS system -
SYSDATE, SYSLAST, etc.

User created variables -
%LET VARNAME = VALUE ;

Macro variables are resolved (i.e. evaluated) by placing an ampersand in front of the name. Why is an ampersand needed? Instructions must be separated from data! In a DATA step SAS character data is quoted to distinguish it from code. (Numeric literals do not need to be distinguished since SAS code elements cannot begin with a number.) Macro instructions must also be distinguished from macro data. But plain SAS code is the data that macro instructions operate on. If the data elements (i.e. SAS code) cannot be distinguished, then the macro instructions must be. Macro instructions and elements are distinguished by beginning them with % or &. (Using quote marks around all SAS code would be a poor way to design a language to operate on SAS code.) The ampersand tells the SAS Macro Facility:

- SYSDATE is a variable.
- Replace the variable with its value.

Common Code - Example 2

Consider the SAS code:

```
.....
TITLE "DATA = IN.MYDATA LIMIT 100 OBS" ;
PROC PRINT DATA = IN.MYDATA (OBS=100);
RUN ;
.....
```

where the dots indicate a block of code which has been left to your imagination.

Note that the data set name and the observation limit are repeated. This means that when changes are made, these values could be changed in one place and not another. Used programs always change. Is there a way to guarantee any change made in one place will also be made in the other? Yes, if they have a common source! Macro variables provide an answer.

```
%LET DATA = IN.MYDATA ;
%LET OBS = 100 ;

.....

TITLE "DATA = &DATA LIMIT &OBS OBS" ;
PROC PRINT DATA = &DATA (OBS=&OBS);
RUN ;

.....
```

This example illustrates another principle:

Lift common code out of setting and move to the front.

Parameters are a closely related concept.

Parameters - Example 3

Survey data are collected state by state, stored, and processed separately. How should the code be set up?

First step:

- PLAN!
- Make sure all data set names and filerefs contain a state code.

In this case we want to use STATE as a parameter. The same program will be run over and over just changing the value of STATE.

The code might look like the following.

```
%LET STATE = TX ;

.....

DATA SURV.TRANS&STATE ;
  INFILE "RAW&STATE.DAT" ;
  INPUT ..... ;
RUN ;

DATA SURV.EDIT&STATE ;
  UPDATE
```

```

SURV.MSTR&STATE
SURV.TRANS&STATE
;
BY KEY;
RUN;

```

.....

Here the macro variable STATE is used in a more complex manner than in the previous examples. Here it is used to construct data set names in which the state is only part of the name. When used in this manner, there is no space between the constant text and the reference &STATE. If there were a space, then the compiler would see the code as two separate tokens and not as one name.

Why does

```
INFILE "RAW&STATE..DAT" ;
```

have two dots? When constant text must follow a macro variable reference without any space separator, the macro facility needs to know where the variable name ends and the constant text begins. A dot is used as a separator which will be eaten up by the macro facility. Hence

```
&STATE and &STATE.
```

mean the same thing when a separator follows them. The special macro dot separator allows you to make references like

```
&STATE.1
```

which would resolve to TX1 when STATE has the value TX. For our INFILE statement we want

```
INFILE "RAWTX.DAT" ;
```

Here the dot is a separator between the filename and its extension. Two dots were needed in the macro statement - one to indicate the end of the macro variable and one to remain to separate the extension.

Packaging Code - Example 4

The title/print problem was left half finished. Typically, there isn't one test print; but many of them. All should be handled the same way. What now? The solution in example 2 would yield many macro variables and a nightmare situation ensuring each received the correct value. What went wrong? We didn't capture the common element at the right level. The common element should have been making a titled PROC PRINT. The concept of a macro provides the needed tool. A macro can be thought of as a package of code with parameters. Here is the example.

```
%MACRO TPRINT (DATA=_LAST_, OBS=500);
```

```
TITLE3
```

```
"DATA=&DATA LIMIT &OBS OBS";
```

```
PROC PRINT DATA=&DATA (OBS=&OBS);
```

```
RUN ;
```

```
TITLE3 ;
```

```
%MEND TPRINT ;
```

To use the macro, i.e. to submit the code package to the SAS compiler, one invokes the macro by placing a % sign in front of the name of the macro followed by the parameter assignment. For example, the macro might be invoked several times as follows:

```
TITLE "BIG PROJECT #000000" ;
```

.....

```
%TPRINT ( DATA = IN.MYDATA , OBS = 100 )
```

```
%TPRINT ( DATA = IN.YOURDATA )
```

.....

```
%TPRINT ( DATA = OUT.RESULT , OBS = 90 )
```

Note that when a macro parameter is not assigned a value by the invocation, then the default value given in the %MACRO statement is used.

Adding Decisions - Example 5

There is a problem with the title when the default value (_LAST_) for DATA is used. Consider the invocation:

```
%TPRINT ( )
```

Now the title is:

```
DATA = _LAST_ LIMIT 500 OBS
```

Remember that macro is very literal: if you say the value is _LAST_, then the value will be precisely those six characters. The macro facility has no understanding of SAS code. Of course the SAS compiler knows the meaning in the PROC PRINT statement, but it would not change any text in a title. We need to get at the name of the data set when the default value is used. The macro facility provides an automatic variable SYSLAST for this purpose. But we cannot use

```
%MACRO TPRINT ( DATA = &SYSLAST, ... )
```

since the value of SYSLAST is not wanted when the macro is compiled; it is wanted when the macro is invoked. We need to make a decision - was the macro invoked with the default value

```
DATA = _LAST_
```

or was it invoked with some other value. If it is the default value then we need to change the value of DATA to &SYSLAST. The %IF - %THEN statement is used to make macro decisions.

```
%MACRO TPRINT (DATA=_LAST_, OBS=500);
  %IF &DATA = _LAST_ %THEN
    %LET DATA = &SYSLAST;
  TITLE3 "DATA=&DATA LIMIT &OBS OBS";
  PROC PRINT DATA=&DATA (OBS=&OBS);
  RUN;
  TITLE3;
%MEND TPRINT;
```

Note that the macro variables in the statement

```
%IF &DATA = _LAST_ %THEN
  %LET DATA = &SYSLAST;
```

are not evaluated when the macro is read (i.e. compiled). This code is acted upon when the macro is invoked, hence &DATA and &SYSLAST resolve to their values at macro invocation time, not macro compile time.

Conditional Execution of a Step - Example 6

We can use %IF - %THEN statements in more significant ways. Add a feature to %TPRINT to allow turning off all prints made by the macro. Why? When testing we want the prints, but in production we don't. Remember it is important to not change the code after testing, so we don't want to rewrite the macro or take out all invocations of the macro. We will define a macro variable, DEBUG, at the top of code with the value, YES, to mean make the prints. Now changing the value of DEBUG should cause elimination of the prints. The macro is defined as follows.

```
%MACRO TPRINT (DATA=_LAST_, OBS=500);
  %GLOBAL DEBUG;
  %IF &DEBUG = YES %THEN
    %DO;
      %IF &DATA = _LAST_ %THEN
        %LET DATA = &SYSLAST;
      TITLE3 "DATA=&DATA LIMIT &OBS OBS";
      PROC PRINT DATA=&DATA (OBS=&OBS);
      RUN;
      TITLE3;
    %END;
%MEND TPRINT;
```

In this manner one can conditionally execute SAS steps such as PROC PRINT.

The statement

```
%GLOBAL DEBUG;
```

is used to tell the macro that DEBUG is a variable defined outside the macro in the global environment. There are two types of macro variables

local - only known and existing during the execution of the macro

global - known and existing throughout a SAS program after the variable is created.

The distinction between local and global variables is important and the failure to understand it is responsible for many macro errors. But it is beyond the scope of this paper to explain more.

Repetitive Code - Example 7

The University of Higher Learning (UHL) stores the list of honor students for each year in a separate flat file. The dean wants a listing of the honor rolls from 1976 to 1985. First let's indicate what we want:

```
DATA YEARyy;
  INFILE INyy;
  INPUT NAME $20. GPA;
  RUN;
```

```
TITLE "UHL Honor Roll for 19yy";
PROC PRINT DATA = YEARyy;
RUN;
```

We might tell a junior programmer to duplicate the above ten times and replace YY with the appropriate year value. To implement this as macro code, we need the concept of a do-loop. SAS macro provides it in the form of the %DO statement.

The macro solution is better because the code is generated when it is used. Consequently there is only one copy to modify when a change is required.

Here is the macro.

```
%MACRO HONORLST;
  %DO YEAR = 76 %TO 85;
    DATA YEAR&YEAR;
    INFILE IN&YEAR;
    INPUT NAME $20. GPA;
    RUN;

    TITLE "UHL Honor Roll for 19&YEAR";
```

```

PROC PRINT DATA = YEAR&YEAR ;
RUN ;
%END ;
%MEND HONORLST ;

```

The macro is invoked

```
%HONORLST
```

Note this macro was written without any parameters as a setup for the next example.

Changing Requirements - Example 8

Now, what happens when the dean wants reports from 1981 to 1990? Or, how about the last five years?

The principle:

- Used macros always grow parameters.
- Moreover, the parameters multiply with use.

It is best to identify and use parameters as early as possible. Here the default values have been chosen for consistency with the previous requirements.

```

%MACRO HONORLST ( FIRST=76, LAST=85 );
  %DO YEAR = &FIRST %TO &LAST ;
    DATA YEAR&YEAR ;
    INFILE IN&YEAR ;
    INPUT NAME $20. GPA ;
    RUN ;
    TITLE "UHL Honor Roll for 19&YEAR" ;
    PROC PRINT DATA = YEAR&YEAR ;
    RUN ;
  %END ;
%MEND HONORLST ;

```

Note there is a slight difference in the invocation of the macro without parameters and the one with.

```
%HONORLST /* without parameters */
```

```
%HONORLST ( ) /* using default parameters */
```

This difference means a change is needed in any code that used the previous version. Such changes should be avoided. If you cannot identify any reasonable parameters then it is best to create one and not use it. For example,

```

%MACRO HONORLST ( DUMMY = ) ;
  /* same code as example 7 */
%MEND HONORLST ;

```

Communication Between Steps - Example 9

An economist at UHL wants to plot the gross national product (GNP) by month. But he wants a reference line for the average GNP in the plot. We might start with instructions for our junior programmer:

```

Do a PROC SUMMARY.
Read the report.
Then write the PROC PLOT code.

```

Macro could provide a better solution, if we can turn the data value produced by PROC SUMMARY into a macro variable.

CALL SYMPUT is the DATA step routine which turns a DATA step value into a macro variable value. The call has the form

```
CALL SYMPUT ( 'macname' , 'macvalue' ) ;
```

where 'macname' and 'macvalue' are character literals, or

```
CALL SYMPUT ( NAME , VALUE ) ;
```

where NAME and VALUE are character variables, or even

```

CALL SYMPUT
  ('VAR' || LEFT(PUT(_N_,3)), VALUE) ;

```

In the last case the character string is constructed.

Using CALL SYMPUT for our problem yields:

```

/* CREATE DATA INFORMATION */
PROC SUMMARY DATA = USDATA ;
  VAR GNP ;
  OUTPUT OUT = AVG MEAN = ;
RUN ;

/* CONVERT TO MACRO INFORMATION */
DATA _NULL_ ;
  SET AVG ;
  CALL SYMPUT ( 'AVG' , GNP ) ;
RUN ;

/* USE MACRO VARIABLE IN CODE */
PROC PLOT DATA = USDATA ;
  PLOT GNP * MONTH / VREF = &AVG ;
RUN ;

```

Data Driven Code - Example 10

Produce a PROC PRINT with the variables in alpha-numeric order. We need a VAR statement, but the VAR statement depends on the data set printed. PROC CONTENTS knows the variables and their alpha-numeric order.

Traditionally CALL SYMPUT is used here to create arrays of macro variables, and a %DO-LOOP is then used to write the VAR statement. But CALL EXECUTE provides a simpler and faster answer.

CALL SYMPUT exports data values to macro variables.

CALL EXECUTE exports data values directly to the macro facility for macro execution. Any SAS code generated is dumped into the input stack for execution after the DATA step is finished.

Remember the old technique:

Write code to a temporary file.
Then use %INCLUDE to execute it.

Here it is in the form of CALL EXECUTE.

```
DATA _NULL_;
  SET NAMES;
  CALL EXECUTE
    ( 'PROC PRINT DATA = ' ||
      NAME ||
      '; RUN;'
    );
RUN;
```

We can use this technique to solve the problem.

```
%MACRO APRINT ( DATA = _LAST_ );
  %IF %UPCASE(&DATA) = _LAST_ %THEN
    %LET DATA = &SYSLAST;
  PROC CONTENTS
    DATA = &DATA
    OUT = __CONTNT ( KEEP = NAME)
    NOPRINT;
  RUN;

  DATA _NULL_;
    IF _N_ = 1 THEN CALL EXECUTE
      ("PROC PRINT DATA = &DATA; VAR");
    SET __CONTNT END = EOF;

    CALL EXECUTE (' ' || NAME);
    IF EOF THEN
      CALL EXECUTE ("; RUN;");
  RUN;
%MEND APRINT;
```

Decisions - Example 11

Remember our first problem - make a compile time decision to read IN.DAY1 on weekdays and IN.DAY6 on weekends. We are now in a position to solve it. We use the automatic variable

SYSDAY, which holds the day of the week, and the fact that Saturday and Sunday begin with an 'S'.

```
%MACRO GETDAILY ( DUMMY = );
  DATA DAILY;
    SET
      %IF %SUBSTR(&SYSDAY,1,1) = S %THEN
        IN.DAY6;
      %ELSE
        IN.DAY1;
    ;
    .....
  RUN;
%MEND GETDAILY;
```

Note how the %IF - %THEN - %ELSE statements are made in the middle of the SET statement. This is permissible and the way many people would code this macro. But each time SAS code and macro code are intertwined in this fashion it becomes a little harder to read. How can one separate the two types of code? Remember the principle we began with:

Use a macro variable to lift code out of context.

Using it, we have:

```
%MACRO GETDAILY;
  %LOCAL DSN;
  %IF %SUBSTR(&SYSDAY,1,1) = S %THEN
    %LET DSN = IN.DAY6;
  %ELSE
    %LET DSN = IN.DAY1;

  DATA DAILY;
    SET &DSN;
    .....
  RUN;
%MEND GETDAILY;
```

Reusable Code - Example 12

No introduction to macro is complete without the classic NOBS macro. It creates a macro variable whose value is the number of observations in a given data set.

```
%MACRO NOBS ( DATA = _LAST_, VAR = NOBS );
  /* USER MUST DECLARE &VAR */
  DATA _NULL_;
    CALL SYMPUT ("&VAR", LEFT(PUT(NOBS,8)));
  STOP;
  SET &DATA NOBS = NOBS;
  RUN;
%MEND NOBS;
```

The NOBS macro is usually written with a fixed variable name NOBS, forcing that variable to be global so that it can be known outside the NOBS macro. I prefer the form given here because it lets the consumer of the macro decide what to call the variable. It also does not force the variable to be global, but at the price that the consumer is now responsible for providing the variable.

A consuming macro might look like

```
%MACRO TEST ( DATA = _LAST_ );
  %LOCAL NUMOBS;
  %NOBS ( DATA = &DATA , VAR = NUMOBS )
  %IF &NUMOBS = 0 %THEN
    %DO;
      /* special code for empty data set */
    %END;
  %MEND TEST;
```

Finished TESTPRNT - Example 13

Throughout these examples I have been driving toward a macro to make test prints when developing SAS code. Here is the macro TESTPRNT in a finished form. It supports all options of PROC PRINT and has the capability of not making any prints during production runs. It uses the global variable DEBUG for control.

```
%macro testprnt (
  data = _LAST_, /* name of data set */
  obslim = 500, /* number obs printed */
  report = NO, /* yes - unconditional prnt */
  tl = 3, /* data set name title line */
  options = , /* print options */
  var = _ALL_, /* variable list */
  pageby = , /* page by variables */
  sum = , /* sum variables */
  sumby = , /* sumby variables */
  id = , /* id variables */
  by = , /* by variables */
  format = , /* variable format. etc. */
  contents = NO /* yes - produce contents */
);

%global debug; /* yes -> test print wanted */
%local i;
%if %upcase(%substr(&report,1,1)) = Y or
  %upcase(%substr(&debug,1,1)) = Y %then
  %do;
    %if %length ( &obslim ) > 0 %then
      %let obslim = ( obs = &obslim );
    %if &tl > 0 %then
      %str(title&tl "data = &data &obslim");
    %if %upcase(%substr(&contents,1,1)) = Y %then
      %str ( proc contents data = &data ; run ; );
```

```
proc print data = &data &obslim &options ;
  by &by ;
  id &id ;
  sum &sum ;
  var &var;
  %if %length(&pageby) > 0 %then
    %str(pageby &pageby);
  %if %length(&sumby) > 0 %then
    %str(sumby &sumby);
  %if %length(&format) > 0 %then
    %str(format &format);
run ;
%if &tl > 0 %then
  %str(title&tl);
%end ;
%mend testprnt;
```

This example is not much harder than any of the preceding ones. But it does use several new macro functions. I leave it as a jumping off point for your continuing study of SAS macro.

Conclusion

Macro can be applied at many different levels to solving SAS programming problems. At the very least one should be able to use macro variables as program parameters and invoke macros written by others. Simple uses of CALL SYMPUT and CALL EXECUTE provide the next level up. Then comes the writing of macros. The examples in this paper use macros to manage SAS code. This permits a clear separation of SAS code from macro code. It is the easiest form of macro to learn and the least error prone. Finally one can write macros to generate parts of steps or parts of statements and macros consisting of pure macro code.

Whatever level is chosen, it is well worth the SAS programmer's time to learn and apply SAS macro to his or her problems.

The author can be contacted by mail at

H. Ian Whitlock
Westat
1650 Research Boulevard
Rockville, MD 20850-3129

or by e-mail at

Whitloi1%westat@mcimail.com

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.