Paper 63

Getting Started with Macro

Ian Whitlock, Westat

Abstract

The macro language is a powerful tool, but it can be dangerous in the hands of the naive. This tutorial takes the first steps in building a strong foundation to understanding of the SAS® macro facility.

Macro variables are introduced as parameters to a SAS program. Then %INCLUDE is added, and the power of this combination demonstrated. Macros are introduced as a better means of packaging code, and the consequences are explored with examples. Design issues and understanding are emphasized.

The consequences of adding the DATA step functions CALL SYMPUT and CALL EXECUTE to the programmer's repertoire are considered. As an extension, very simple PROC SQL code is introduced to make macro variables holding lists. The power of this new tool is demonstrated with examples.

If you restrict the use of macro to the above tools you have a dramatically more powerful language than SAS alone provides, and debugging is no harder than without these tools. In this way you are prepared to step into the world of SAS macro with the appropriate confidence that you are the master of the language and not vice versa.

Introduction

This beginning tutorial is designed to help you get started with using the macro facility and learning how to design programs.

In the first section macro variables are introduced and it is shown how to use them as parameters to a program. Then %INCLUDE is used to turn this into a powerful design technique which can save much time in coding and debugging.

The next section extends the idea to using macros as better method of packaging code than shown in the previous section. The second example illustrates the use of the DATA step subroutine CALL SYMPUT to communicate between two steps of the macro.

The third section introduces the DATA step subroutine CALL EXECUTE as method of repeatedly invoking a macro with different parameter values.

At this point none of the traditional complexity of the macro language has been used, hence debugging is still as simple as it is in plain SAS. The fourth section explains how to make macro decisions with %IF an introuduces simple statements like %PUT, %GLOBAL and %LOCAL. After some simple uses, a relatively hard problem is discussed and solved while still keeping the macro code to a minimum.

By the end you should have better ideas about designing programs and how to use the macro facility without harm. After some experience, you should be ready to tackle the macro facility with a sound understanding of the basic principles introduced in this tutorial.

Macro Variables

Let's begin with macro variables. A macro variable holds some text as its value. It is often created with a %LET statement. For example,

```
%let state = FL ;
```

Note that there are no quotes around the value, FL, since all values are always text. The text is not SAS data, but rather part of a SAS program. The variable is referenced with an ampersand (&) in front of the name.

If text is to be recognized as part of a SAS program, then the macro instructions must be distinguished with special symbols. The %-sign is used to start macro instructions and some macro objects such as macro functions. The &-sign is used to indicate the value of a macro variable. For example:

```
title "Data from &state" ;
proc print data = mstr ;
   where state = "&state" ;
run ;
```

Double quotes are required around the reference, &STATE, in the WHERE statement because this is used as a SAS literal, and so it must be quoted. The quoted material in single quotes is treated as a single entity and sent directly to the SAS compiler, hence that quoted material is never seen by the macro facility and any macro references are not resolved. Double quotes are treated differently. They are tokens, so they go to the macro facility along with the material inside. It is the SAS compiler that sees double quotes as making a literal value, not the macro facility.

Two important ideas are present in the little piece of code shown above.

- The common code problem

  How can you guarantee the title reflects the subsetting WHERE statement?

Make a macro variable to hold the text and then reference it in each appropriate place.

- The parameter problem

  How can you pass control information to a block of code so that one can change the code and the way in which the block acts by merely changing one value?

  Make a macro variable to hold the parameter value and then reference it in each appropriate place.

Usually the ideas go together as two views of the same thing.

To see the significance of the second idea, let's extend the example a little. Suppose you have to write a program to update a master file with survey data that is collected state by state. You have just finished the code for Florida.

```
/* Update program for FL */

filename fl "c:\survey\fl.dat" ;
libname lib "c\survey" ;

data fl ;
   infile fl ;
   input .... ;
run ;

proc sort data = fl ;
   by id ;
run ;

data lib.mstr ;
   update lib.mstr fl ;
   by id ;
run ;

title "Updated Surveys for FL" ;
proc print data = lib.mstr ;
   where state = "FL" ;
run ;
```

The boss approves your code, and now you are ready to duplicate it 49 times, changing "FL" to each of the other state abbreviations. Wait, you ask yourself, "What about parameters? Can that idea help here?" So you try:

```
/* Survey state update program */
filename &st "c:\surv\&st..dat" ;
libname lib "c\surv" ;

data &st ;
   infile &st ;
   input .... ;
run ;

proc sort data = &st ;
   by id ;
run ;

data lib.mstr ;
   update lib.mstr &st ;
   by id ;
run ;

title "Updated Surveys for &st" ;
proc print data = lib.mstr ;
   where state = "&st" ;
run ;
```

Now you have one parameterized program instead of 50. But how are you going to run it? What about %INCLUDE? Suddenly the light hits. Write a driver, i.e. a program to manage the whole process using a %LET to control the state and a %INCLUDE to execute the code. So you write:

```
/* Survey update driver */

filename progs "c:\programs" ;
%let st = AL ;
%inc progs (stateupd) ;
%let st = AK ;
%inc progs (stateupd) ;
%let st = AZ ;
%inc progs (stateupd) ;
%let st = AR ;
%inc progs (stateupd) ;
...
```

One detail we slid over - what is that second dot in the FILENAME statement in the state update program?

```
filename &st "c:\surv\&st..dat" ;
```

The question is - how does a reference to a macro variable end? In the first instance a space ends the reference since a space cannot be part of the name of the variable. But we don't want a space in the filename. SAS uses the dot as an optional ender to the reference and the macro facility eats the dot as an ender. Using the dot to end a macro variable reference is optional, but when present, the first dot is always an ender. Thus we need two dots, one to end the reference, &ST, and one to separate the filename from the extension.

Using the dot as an ender means that a parameter doesn't have to refer to a complete object. For example suppose we have a Florida edits data set LIB.FLEDITS. We can reference the data as

```
lib.&st.edits
```

The first dot is not an ender, because there is no preceding reference to a macro variable, so it is in the final value for the compiler and separates the libref from the member name. In contrast, the second dot ends the reference, &ST; hence it is not part of the final value for

the compiler. Consequently the compiler sees FLEDITS as the member name.

Having seen the power of parameterization in a toy example, let's consider a large scale real life problem. A survey was conducted asking young people and their parents how they perceived certain problems and how they placed a value on those problems.

The variables are indicated in Table 1 below. A programmer was asked to write the program for the following cross tabs:

1. Parental evaluation versus Youth evaluation
2. Parental perception versus Youth perception
3. Parental evaluation versus Parental perception
4. Youth evaluation versus Youth perception
5. Demographic versus all of the others

And correlations:

1. Parental evaluation versus Youth evaluation
2. Parental perception versus Youth perception

Parental Peception	Parental Evaluation	Youth Peception	Youth Evaluation
PACASHED	PICASHED	YACASHED	YICASHED
PACIVCAR	PICIVCAR	YACIVCAR	YICIVCAR
PAHIQUAL	PICNTRY	YAHIQUAL	YICNTRY
PAHITECH	PIHIQUAL	YAHITECH	YIHIQUAL
PALEADER	PIHITECH	YALEADER	YIHITECH
PAMATURE	PIHOME	YAMATURE	YIHOME
PAMENTAL	PIINNOV	YAMENTAL	YIINNOV
PAPHYS	PILEADER	YAPHYS	YILEADER
PAPOTEN	PIMATURE	YAPOTEN	YIMATURE
PAPROUD	PIMENTAL	YAPROUD	YIMENTAL
PASELCON	PIPHYS	YARMBOSS	YIPHYS
PASTEP	PIPOTEN	YARMCOUN	YIPOTEN
PATRAIN	PIPROUD	YARMCOW	YIPROUD
PAWIDE	PISELCON	YARMDAD	YISELCON
		YARMFARM	YISERCOM
		YARMFMIL	YISERPAR
		YARMFNO	YISTEP
		YARMMOM	YITRAIN
		YARMSTUD	YIWEEKEN
		YARMTEAC	YIWIDE
		YASELCON	
Demographics		YASTEP	
YPESIM		YATRAIN	
E13TO24		YAWIDE	
EAGE		YDMARITL	
ECALCAGE		YEDLEV	

Table 1 - Survey variables.

In this example, lists play an important role. Let's make each list a macro variable and see how the program might go.

```
%let pp =
    PACASHED PACIVCAR PAHIQUAL PAHITECH
    PALEADER PAMATURE PAMENTAL PAPHYS
    PAPOTEN  PAPROUD  PASELCON PASTEP
    PATRAIN  PAWIDE ;
%let pi =
    PICASHED PICIVCAR PICNTRY  PIHIQUAL
    PIHITECH PIHOME   PIINNOV  PILEADER
    PIMATURE PIMENTAL PIPHYS   PIPOTEN
    PIPROUD  PISELCON ;
%let yp =
    YACASHED YACIVCAR YAHIQUAL YAHITECH
    YALEADER YAMATURE YAMENTAL YAPHYS
    YAPOTEN  YAPROUD  YARMBOSS YARMCOUN
    YARMCOW  YARMDAD  YARMFARM YARMFMIL
    YARMFNO  YARMMOM  YARMSTUD YARMTEAC
    YASELCON  YASTEP   YATRAIN  YAWIDE
    YDMARITL YEDLEV ;
%let yi =
    YICASHED YICIVCAR YICNTRY  YIHIQUAL
    YIHITECH YIHOME   YIINNOV  YILEADER
    YIMATURE YIMENTAL YIPHYS   YIPOTEN
    YIPROUD  YISELCON YISERCOM YISERPAR
    YISTEP   YITRAIN  YIWEEKEN YIWIDE ;
%let dem =
    YPESIM E13TO24 EAGE ECALCAGE ;
%let data = in.survdata ;

title2
"Parent Eval vs Youth Eval";
proc freq data = &data ;
   table (&pi) * (&yi) ;
run ;

proc corr data = &data ;
   var  &pi ;
   with &yi ;
run ;

title2
"Parent Percep vs Youth Percep";
proc freq data = &data ;
   table (&pp) * (&yp) ;
run ;

proc corr data = &data ;
   var  &pp ;
   with &yp ;
run ;

title2
"Parent Eval vs Parent Percep";
```

```
proc freq data = &data ;
   table (&pi) * (&pp) ;
run ;

title2
"Youth Eval vs Youth Percep" ;
proc freq data = &data ;
   table (&yi) * (&yp) ;
run ;

title2 "Demo vs all of the others" ;
proc freq data = &data ;
   table (&dem) * (&pi &pp &yi &yp) ;
run ;
```

Now why was the data set name made a parameter? Before the job could be run, the specifications changed. Do all of the above for three data sets:

- Black and Hispanic
- Other
- All

Here is the driving program:

```
/* pgm1.sas - see revised memo 1 */
libname in 'c:\surv\dat' ;
filename pgm "c:\surv\pgm";

*** list macro variables as above ***

data blckhisp othr ;
   set in.survdata ;
   if race in (1 2) then
      output blckhisp ;
   else output othr
run ;

title "Survey - all" ;
%let data = in.survdata ;
%inc pgm ( pgm2 ) ;

title "Survey - Black/Hispanic" ;
%let data = blckhisp ;
%inc pgm ( pgm2 ) ;

title "Survey - Other" ;
%let data = othr ;
%inc pgm ( pgm2 ) ;
```

When the code above is executed, the log does not show the resolved value of the macro variables. To see these values, use the system option SYMBOLGEN. Later on, the option may generate too much information, but for SAS programs with just a few macro variable references, it is appropriate.

By now you should begin to see the power of combining two simple ideas - macro variables with %INCUDE to package a program in parts so that some of the parts may be reused.

Macros

The combination of %INCLUDE and macro variables is powerful, but there are problems:

- The interface between parts is not absolutely clear.
- The macro variables are known throughout the entire program and may be changed incorrectly in any part.
- The system soon becomes cumber-some when there are many parts, or the nesting of %INCLUDEs exceeds one level.

All of the above tend to limit the size and complexity of programs that can be built with these tools. What we need is a better method of packaging code than the %INCLUDE statement can provide.

The macro concept provides that better tool. A macro is a unit of code with parameters. A macro begins with the %MACRO statement, and ends with a %MEND statement. For example, let's take our first problem of insuring that a print from PROC PRINT includes a title naming the DATA set. The macro is called TESTPRNT because the problem is particularly important for printing data when testing a program. Imagine trying to study five unlabeled or incorrectly labeled prints of critical data sets in some large program. The macro has three parameters DATA naming the data set, OBS specifying how many records to print, and TL specifying the title line for the print.

```
%macro testprnt
   ( data = &syslast ,
     obs = 90 ,
     tl = 3
   ) ;

   title&tl
     "Data = &data (obs=&obs)" ;
   proc print
        data = &data (obs=&obs) ;
   run ;
   title&tl ;

%mend testprnt ;
```

The values to the right of each parameter provide default values that need not be specified when the macro is invoked, if they are acceptable.

As the code is read, everything is stored away for future use. SYSLAST is an automatic macro variable created by the SAS system whose value is the last created SAS data set. Note that the value of SYSLAST is not stored at this

time, it is the reference to SYSLAST that is stored. In the same way any references, &DATA and &OBS, within the macro are not resolved at this time. The time of storing a macro is known as macro compile time. It is the time the code is read from the %MACRO statement down to the corresponding %MEND statement. No SAS code is executed at this time. The code is simply stored for future reference or invocation.

In general it is a mistake to put one macro definition inside another. Remember the inner macro is not compiled when the outer macro is compiled, since resolution does not take place at this time. Instead, it is compiled each time the outer macro is executed.

The macro can then be invoked by placing a %-sign in front of the macro name and following it with parentheses plus any parameters that will not take their default values. For example,

```
%testprnt ( )

%testprnt (data = mydata )

%testprnt (data = final ,
           obs = 500 ,
           tl = 5 )
```

are all legitimate invocations of TESTPRNT. Note that there are no semi-colons at the end of a macro invocation. If there were a semi-colon, it would be sent to the SAS compiler as a semi-colon. If there were no preceding SAS statement without a semi-colon, then it would be treated as a null statement, which might be harmful (for example when it splits an IF statement from and ELSE statement).

At the time of invocation, all references are resolved, and the PROC PRINT code is generated and sent to the SAS supervisor for execution. This time is known as macro execution time. It is exceedingly important to understand the difference between macro compile time, when the code is simply stored away for future use, and macro execution time, when SAS code is generated.

There are several features of the design of this simple macro that should be noted. The parameter names DATA and OBS are carefully chosen to match the SAS use of these terms. The default value for DATA is chosen to match the SAS defaults. These two features make it much easier to use the macro. You can already guess at the parameter names and their default values. In the case of OBS, I did not choose MAX for the default because typically test prints are limited. Hence it was more important find a default natural to the purpose of the macro than to dogmatically choose the SAS default, MAX. For the same reason TL=1 was not chosen. Typically a program has at least one title line used throughout the program and possibly more used for each section of the program. It would be most annoying to have to reinstate these titles every time the macro is called; hence TL=1 is a poor choice. TL = 3 is a good choice because an extra blank line is not too bad, and TITLE3 is far enough down that there should be few occasions when it is not acceptable. Another important point is the line after the PROC PRINT clearing the extra title line. (With the use of SCL functions in the macro facility, it is possible to eliminate the TL parameter by determining the first available title line; but this technique is beyond a beginning tutorial on the macro facility.)

The choice of parameters is important for flexibility and ease of use, but it should also document the interface between the macro and the rest of the program. In other words, it is a contract - if the program invoking the macro provides the required parameter values, then the macro will generate the code to complete the assigned task. A big mistake, made by many programmers, is to leave secret understandings not spelled out in the parameter list. Some programmers even use no parameters leaving all communication to global macro variables.

For debugging macro execution, it is important to use the system option, MPRINT. This option will show the SAS code generated by the macro. Now there is no need for the option SYMBOLGEN because the generated resolved code is shown by MPRINT.

Since we have used nothing but SAS code and parameters, debugging is not any harder than debugging SAS code. If there is a syntax error, it is because we wrote the wrong SAS code or because we have the wrong value for a parameter, and that value is showing in the code.

Parameters should be carefully chosen to provide useful flexibility with good defaults for simple use. The macro itself should be chosen to handle an important task. Typically the task occurs repeatedly in a project or large program.

As a second example, consider the common problem of making a stratified random sample from a frame; i.e., given a SAS data set with a stratum variable, make a random subset so that each value of the stratum variable is equally represented in the sample.

What should the parameters be? First, we need DATA to name the input DATA set. Second, we need to know the name of the stratum variable. Third, we need a means of conveying the sample size. Fourth, we need to know the name of the output data set. What about the sampling process? Should it be repeatable? Perhaps we should also provide a seed parameter for RANUNI. What should the seed be? Zero is the default for RANUNI, so it should be our default.

The process consists of sorting by the stratum variable, counting each value, and then merging these counts with the sorted file to randomly choose the subsets. The

sampling step consists of the process for each stratum value:

- Start with a count, _FREQ_ of the available records for a stratum and a count, RNDWANT, of the subsample for this stratum.
- Generate a uniformly random number between 0 and 1 for each record and choosing the record whenever the number generated is smaller than the ratio, RNDWANT/_FREQ_
- Reduce _FREQ_ by one and if the record is chosen, then also reduce RNDWANT by one.

Here is an implementation of the plan with an extra step to provide a random seed when the default SEED = 0 is used.

```
%macro rndsamp
    ( data = &syslast ,
      out = _DATA_ ,
      stratum = stratum ,
      rate = 0.1 ,
      seed = 0 ) ;

  proc sort data=&data
            out=rndtemp;
    by &stratum ;
  run ;

  proc summary data=rndtemp nway;
    class &stratum ;
    output out = rndcnt ;
  run ;

  data _null_ ;
    if &seed = 0 then
    do ;
      seed = int ( 1000000000000
                * ranuni (&seed))
      ;
      seed = mod ( seed,
                   1000000000 ) ;
      call symput ( 'seed' ,
                  put (seed, 9.));
    end ;
    else
      seed = &seed ;
    put "NOTE: Sampling based on"
        seed = ;
  run ;

  data &out ( drop=rndwant
                   _freq_);
    merge rndtemp
          rndcnt (keep=&stratum
                       _freq_)
    ;
    by &stratum ;
    if first.&stratum then
    do ;
      rndwant =
         ceil (&rate * _freq_) ;
    end ;
    if ranuni (&seed) <
       rndwant/_freq_ then
    do ;
      output &out ;
      rndwant +(-1) ;
    end ;
    _freq_ + (-1) ;
  run ;

  %mend  rndsamp ;
```

A new technique for assigning a value to a macro variable is required here because we wish to allow the user to be able to repeat a run made with SEED = 0. First we generate a seed based on 0, write a message on the log so the user can use this seed again, and then we have to communicate the seed value to the main DATA step creating the sample.

CALL SYMPUT is a DATA step subroutine that interacts with the macro facility. Given a macro variable name, and a value, the subroutine causes the macro variable to be assigned the value. Note that %LET is not good enough because a %LET would act during the compilation of the DATA step and not during its execution the way CALL SYMPUT does. In our case the new seed value is not known at compile time.

CALL SYMPUT provides a new way to turn data into code. Thus it provides an important tool in writing flexible programs.

Repetition

Often one wants to do something repeatedly. The macro facility does provide a %DO-loop instruction, but we are looking for ways to avoid the complexity of the full macro language. Often these loops are based on the values of a variable in a SAS data set. Here it is particularly helpful to use the DATA step/macro interface function, CALL EXECUTE.

For example, suppose we wish to make test prints of every SAS data set in a library using the macro, TESTPRNT, developed above. SASHELP.VTABLE provides a SAS view with the members of each SAS library, so we have the looping situation mentioned. For each MEMNAME value where LIBNAME is restricted to the given library, we want to call %TESTPRNT. We use a macro, LOOK, to handle this task. LOOK should have a parameter, LIB, to specify the library, a parameter OBS, to

specify how many observations should be printed, and a parameter, TL, to specify where the data set title can be written. For each given member, we want to call

```
%testprnt ( data = &lib..member,
            obs = &obs,
            tl = &tl )
```

The parameter OBS is used both in LOOK and TESTPRNT. The "OBS" on the left side is the TESTPRNT parameter, and it is assigned the value that &OBS (reference to the LOOK parameter) has. When communicating parameter values to a helping macro, it is often necessary to obtain these values from parameters in the calling macro, and it is easiest to use the same names because they refer to the same ideas although they are different parameters owned by different macros.

CALL EXECUTE passes a character string (limited to 200 bytes in Version 6) to the macro facility for immediate execution during the DATA step. Typically the macro facility generates SAS code which is then dumped into the input stack for SAS execution when the DATA step ends. In our case the character string is the invocation of %TESTPRNT and it consists of literal values concatenated with DATA step variable values. Here is the code.

```
%macro look ( lib = work ,
              obs = 90 ,
              tl = 3 ) ;
  %let lib = %upcase(&lib) ;

  data _null_ ;
    set sashelp.vtable ;
    where libname = "&lib" ;
    call execute
      ( '%testprnt '            ||
        "   ( data="|| memname ||
        "    , obs=&obs"       ||
        "    , tl=&tl)"
       ) ;
  run ;

%mend  look ;
```

The macro function %UPCASE is used to create a standard basis of comparison on uppercase letters. Double quotes are used in the character strings containing parameters of LOOK so that the macro references, &OBS and &TL will be evaluated once during the compilation of the DATA step. Single quotes around the macro reference, %TESTPRNT, are essential. Without them the macro would be invoked once during the DATA step compile. With them, an invocation is passed to the macro facility via CALL EXECUTE each time an observation passes the WHERE statement during execution of the DATA step. From the point of view of the compiler, the character string is concatenated character literals with the exception of the variable, MEMBER, coming from SASHELP.VTABLE.

There are two more important points in the use of CALL EXECUTE. One, a macro invocation cannot be split between several calls to EXECUTE. Two, macro variable values must be available without the execution of SAS code, since the SAS code does not execute until after the DATA step is finished. This means one cannot assign macro variable values via the SYMPUT function or PROC SQL in a macro invoked by CALL EXECUTE.

Now let's turn to a significant application using CALL EXECUTE. You have a large survey data base consisting of about 60 SAS data sets corresponding to the tables of data base produced with the Blaise system. There are 300 variables spread over these sets, which refer to "Other Specify" type questions. For example,

For breakfast do you prefer:
1. cereal
2. toast
3. egg
4. other _____

You are assigned to produce a report giving for each respondent the non-blank values of these "Other Specify" variables. The data sets cannot be merged by the respondent's ID, because many of the tables have repeating records for a person. When they do, there is a variable to indicate to which line the "Other Specify" value applies, but the variable may have different names in different data sets.

Since the report could be easily produced from a SAS data set with the variables RESPID, VARIABLE, ROW, and VALUE; your real job is to produce this data set. For one data set in the data base the program is simple. You might use:

```
data temp
       ( keep = respid variable
                row value ) ;
  length variable $ 8 value $ 80;
  set atable
      ( keep = respid seqno
          frstothr secnothr ...);
  array othrval (n) frstothr
                    secnothr ...;
  if there is a seqno then
     row = seqno ;
  else
     row = . ;

  do i = 1 to dim ( othrval )
     if othrval ( i ) ^= ' ' then
     do ;
        call vname ( othrval(i),
                     variable ) ;
```

```
        value = othrval ( i ) ;
        output othrspec ;
      end ;
  end ;
run ;
```

Conceptually, the problem is now simple:

- Add a PROC APPEND statement to the above code to collect the data into one data set
- Put the code in a macro

```
%GETOTHR ( lib =,
           mem =,
           seq = )
```

- Invoke the macro via CALL EXECUTE in a DATA step reading a control data set listing the "Other Specify" variables for each member of the data base.

There is a technical problem that the list of "Other Specify" variables can grow to a length of over 200 characters so it cannot be the value of a parameter in %GETOTHR when invoked by CALL EXECUTE. We will avoid the problem by making just one list variable, OTLIST, with the understanding that %GETOTHR will refer to this macro variable. (In general it would be better to pass the name, OTLIST, as a parameter in %GETOTHR, but that requires the use of indirection and is not a good topic for a beginning tutorial on macro.) There is, of course, a lot of hard work making the control data set containing the member names and their "Other Specify" variable names.

```
data _null_ ;
   set control ;
   by memname ;
   if first.memname then
   do ;
      cnt = 0 ;
      call execute
        ( '%let otlist = ;' ) ;
   end ;
   cnt + 1 ;
   call execute
     ( '%let otlist = &otlist '||
       name || ';' ) ;

   if last.memname ;

   call execute
     ('%getothr(lib='||"&lib"||
           '  ,mem='||memname||
           '  ,seq='||seqvar )'
     ) ;
run ;
```

The last task is to put it all together in a system of two macros, %DRIVER and %GETOTHR, but first we will take a quick look at the macro instruction, %IF, and apply it to our TESTPRNT macro.

Decisions

The macro facility is really a programming language where one programs, i.e. writes instructions, for writing SAS programs. So far, the only instructions we have covered are %LET, %MACRO and %MEND together with the two DATA step subroutines, CALL SYMPUT and CALL EXECUTE. The simple instruction, %PUT, is important for debugging. It writes whatever follows the %PUT up to the first semicolon to the SAS log. For example, we might echo the parameters of %TESTPRNT by beginning with:

```
%put Parameter values ;
%put DATA=&data ;
%put obs=&obs ;
%put tl=&tl ;
```

In general, I prefer using %PUT statements to SYMBOLGEN because I can focus their use on a debugging problem.

So far the macro %TESTPRNT doesn't live up to its name because the print is always executed, even when we are finished with debugging. The answer is to make a decision whether to generate the PROC PRINT or skip it. Let's make a variable, DEBUG, at the top of every program using %TESTPRNT having the value YES or NO. When &DEBUG is YES the print code should be generated, and when it is NO there should be no code generated. Here is the new version of %TESTPRNT.

```
%macro testprnt
   ( data = &syslast ,
     obs = 90 ,
     tl = 3
   ) ;
   %global debug ;

   %if %upcase(&debug) = YES
   %then %do ;
      title&tl
       "Data = &data (obs=&obs)";
      proc print
         data = &data (obs=&obs);
      run ;
      title&tl ;
   %end ;

%mend testprnt ;
```

With this small addition the macro becomes much more useful. When we want prints, we have them, and when we don't, they are gone. Moreover, this decision is

controlled by the simple assignment of YES or NO to the macro variable DEBUG. The macro still does not have all the features of PROC PRINT, but they could be built in by adding more parameters and then making decisions on whether to (or how to) generate the corresponding feature's code.

A few simple decisions can add greatly to the flexibility and use of a macro, but they also begin to lead us into more difficult programming.

(The %GLOBAL statement just says that the variable is defined in the global environment, out side the macro. Strictly speaking it isn't necessary because we already agreed to assign a value at the top of the program, i.e. in the global environment. The advantages are that the program will work, albeit without test prints, if the assignment is forgotten; and that it tips off the reader that the variable DEBUG is assigned outside of the macro. It is also a good idea to use a %LOCAL statement to announce any variables that should be restricted to the macro.)

Let's apply decisions one more time, using the survey task discussed in the previous section. Suppose we spice up the task, by requiring that the report be restricted to the list of respondents specified in a macro variable IDLIST.

There are really two decisions for %GETOTHR:

- Should the sequence variable, ROW, be a constant or defined by a variable?
- Should ID be in &IDLIST or is there no restriction?

Note that in both cases, the decision is about a parameter and need only be decided once. For simplicity, we previously made the first decision using the code:

```
if there is a seqno then
     row = seqno ;
  else
     row = . ;
```

Here, the decision is made over and over on each record, but in fact, a given DATA set either has a variable to indicate the sequence or it doesn't. This indicates that the decision is being made at the wrong time. It is made during SAS execution time, when in fact we should decide whether to generate the code:

```
row = seqno ;
```

or the code

```
row = . ;
```

Hence we need a macro %IF statement to decide which code to generate.

```
%if &seq = %then
%do ;
   row = . ;
%end ;
%else
%do ;
   row = &seq ;
%end ;
```

Note that only one of the two assignment statements will be in the DATA step, and we are deciding which one to use during the compiling of the step. This means the decision is made once at compile time and not once per iteration of the DATA step loop during execution of the step. In general it is more efficient to make decisions as early as possible and more flexible to postpone them as long as possible.

The second decision can be based on whether the parameter, IDLIST, is empty or not.

```
%if &idlist ^= %then
%do ;
   if id in ( &idlist ) ;
%end ;
```

When IDLIST is not empty, we generate a subsetting IF statement. Otherwise we don't need a subsetting IF statement.

With this preparation we have only to package the steps above with minor changes as macros.

```
%macro driver ( lib = ,
                control = ,
                idlist =
               ) ;

  data _null_ ;
    set &control ;
    by memname ;
    if first.memname then
      call execute
        ( '%let otlist = ;' ) ;
    call execute
      ( '%let otlist=&otlist '||
                  name || ';' ) ;

    if last.memname ;

    call execute
      ('%getothr(lib='||"&lib"||
            "  ,mem="||memname||
            "  ,seq="||seqvar ||
            "  ,idlist=&idlist)"
      ) ;
  run ;

%mend driver ;

%macro getothr
          ( lib = ,
```

```
            mem = ,
            seq = ,
            idlist =
          ) ;

  data temp
       ( keep = respid variable
                 row value ) ;
    length variable $ 8
           value $ 80 ;
    set &lib..&mem
      ( keep = respid &seq
                &otlist) ;

    array othrval (*) &otlist ;

    %if &idlist ^= %then
    %do ;
       if id in ( &idlist ) ;
    %end ;

    %if &seq = %then
    %do ;
        row = . ;
    %end ;
    %else
    %do ;
        row = &seq ;
    %end ;

    do i = 1 to dim ( othrval )
      if othrval ( i ) ^=' ' then
      do ;
        call vname ( othrval(i),
                     variable ) ;
        value = othrval ( i ) ;
        output temp ;
      end ;
    end ;
  run ;

  proc append base = othrspec
              data = temp
  run ;

%mend getothr ;

%driver ( lib = surv ,
          control = surv.cntl ,
          idlist = "00123"
                   "00890"
         )
```

Conclusion

By now you should be convinced that some pretty hard and general programs can be written with relatively little use of real macro code. So, it is time to stop this tutorial. Go get some experience, and then you will be able to introduce still more macro instructions in your macro code without making the code unreadable and probably with less macro debugging problems than one who has not learned to take the first step before trying the full fledged macro language.

The author may be contacted by e-mail at

whitloi1@westat.com

or by mail at

Ian Whitlock (RA1356)
1650 Research Boulevard
Rockville, MD 20850

Questions are welcome.