

Five Easy Macros

David Sommer, Optimal Systems, Inc.

Most SAS® users believe they should know more about macros, but that macros are difficult. They are right on both counts. This tutorial introduces the writing of SAS macros by illustration. I discuss five macros which I use often in application development.

What is a Macro?

Fundamentally, a macro is a statement of the way you want *text* to be generated. *Text* may be any SAS statements. Consider this macro and an example invocation of it.

```
%MACRO CRIMERPT(WHO, WHAT, WHEN, WHERE,
WHY);
DATA NULL;
PUT "The notorious &WHO committed yet"
  " another &WHAT &WHEN.."
  " Police reported that &WHO"
  " is responsible for &WHAT"
  " at &WHERE.."
  " &WHO's motive is &WHY..";
RUN;
%MEND CRIMERPT;
```

```
%CRIMERPT(John Dillinger, bank robbery,
late Thursday night,
First Bank of Cicero,
'to get rich quick')
```

The result in the SASLOG:

The notorious John Dillinger committed yet another bank robbery late Thursday night. Police reported that John Dillinger is responsible for bank robbery at First Bank of Cicero. John Dillinger's motive is 'to get rich quick'.

NOTE: The DATA statement used 0.25 seconds.

0. Open Code Macrovariables

This is "macro 0" because it isn't a macro at all! But the easiest useful thing to learn about the SAS macro facility is the way a macrovariable can be defined in open code, that is, not inside a macro, and then used anywhere following the definition.

The following statements might appear at the beginning of a SAS program:

```
%LET ALPHA = ABCDEFGHIJKLMNOPQRSTUVWXYZ;
%LET NUMERIC = 0123456789;
%LET MYLIB = 'C:\MYSTUFF\PLAYPEN';
%LET MYINPUT = TEST3;
%LET MYOUTPUT = SUMTEST3;
```

The first two are useful strings in certain dataset coding. For example, if you wanted to test that

variable NAME is entirely alphabetic, you could use this:

```
IF VERIFY(UPCASE(NAME), "&ALPHA") THEN PUT
NAME "is not alphabetic.";
```

This is exactly the same as if we wrote:

```
IF VERIFY(UPCASE(NAME),
"ABCDEFGHIJKLMNOPQRSTUVWXYZ") THEN PUT NAME
"is not alphabetic.";
```

The MYLIB, MYINPUT, and MYOUTPUT might be used in statements such as:

```
LIBNAME THEBANK "&MYLIB";
PROC SORT DATA=THEBANK.&MYINPUT
OUT=THEBANK.&MYOUTPUT;
RUN;
```

This is convenient when a program may have many references to certain datasets or files. The program may be written "generically" and can then be changed consistently by just altering a few lines at the beginning.

1. A Print Macro

Our first real macro illustrates some of the flexibility in the macro language:

```
%MACRO DSPRINT(DATA=, TITLE=, VAR=,
FIRSTOBS=1, OBS=1000000000);
** Utility macro to print a dataset. If
TITLE is missing, will make title using DATA
name. If VAR missing, all variables print-
ed.;
PROC PRINT DATA=&DATA(FIRSTOBS=&FIRSTOBS
OBS=&OBS) NOOBS;
%IF %QUOTE(&TITLE) EQ %THEN
%STR(TITLE "DATASET &DATA"););
%ELSE %STR(TITLE "&TITLE"););
%IF %QUOTE(&VAR) NE %THEN
%STR(VAR &VAR););
RUN;
%MEND DSPRINT;
```

A few invocations might look like this:

```
%DSPRINT(DATA=FREDDY, VAR=A B C)
%DSPRINT(DATA=CATHY,
TITLE=Catherine the Great,
OBS=200)
```

A few things worth noting about this macro. First, we are using *keyword* rather than *positional* parameters. Keyword parameters allow us to assign default values of parameters as we have done here. Here, one need not specify the FIRSTOBS and OBS parameters if the entire dataset is to be printed. (If FIRSTOBS is omitted, then 1 is used as the first

observation; if OBS is omitted, then 1000000000 is used as the last observation to be printed.)

Second, *macro logic* is used to handle various conditions. Thus, if no TITLE is given, then one is made up. And if the VAR parameter is not specified, then no VAR statement is generated.

2. A Little Killer

The next macro isn't complicated, nor does it do anything you couldn't do easily without it. But it generates a few lines of code that are needed often. And defining a macro like this tends to make the handling of similar situations consistent. Here is the MYABORT macro:

```
%MACRO MYABORT(MSG,DS);
  /* MSG is abort message. DS is Y if a DATA
  NULL is needed. When used within a data
  step, may be invoked like
  "IF ... THEN %MYABORT(message)"
  since a "DO ... END" structure is generat-
  ed.;
  %IF %QUOTE(&DS) EQ Y %THEN
    %STR(DATA NULL _);
  %ELSE %STR(DO);
  PUT "***** ABORT, MESSAGE = &MSG *****" ;
  ABORT RETURN 100;
  STOP;
  %IF %QUOTE(&DS) EQ Y %THEN %STR(RUN);
  %ELSE %STR(END);
%MEND MYABORT;
```

This macro has the flexibility to be used either inside a datastep (the usual case), or invoked from another macro. In large production applications, something like this is needed many times to deal with error conditions.

I'd like to point out a little subtlety here. The conditional "%IF %QUOTE(&DS) EQ Y...." is written the way it is because if the user *doesn't give the DS parameter* then the simpler coding "%IF &DS EQ Y...." would fail – it would resolve as "%IF EQ Y...." which is bad syntax. But with the %QUOTE, the macro compiler is able to interpret the %IF as comparing a null string with Y.

3. Multi-Platform Acrobatics

I often write applications that must run on several platforms: OS/2, Windows, CMS, VSE, and MVS for example. I avoid a lot of trouble by centralizing all the special code needed for each platform in a single macro. I call this macro SETFILES because most, but not necessarily all, of the differences are in FILE or LIBNAME statements. Of course, there is a

different version of this for each application. Here's one that I use for a program that runs on CMS or various personal computer platforms. The macrovariable _ROOT used in this is defined with a %LET statement, as shown here.

```
%LET _ROOT = C:\CUSTOMER;
%MACRO SETFILES;
  %GLOBAL MINUMLNG; /* Min Numeric Length;
  %IF &SYSSCP EQ CMS %THEN %DO;
    FILENAME SASMACRO "&FNAME SASMACRO A";
    FILENAME VSEJOB "&FNAME VSEJOB A"
    LRECL=80;
    FILENAME MESSAGE "&FNAME MESSAGE A";
    FILENAME LISTING "&FNAME LISTING A"
    LRECL=&LISTRECL;
    LIBNAME DATADICT "&DATADICT A2";
    %LET MINUMLNG = 2;
    * Output set to fname LISTING;
    PROC PRINTTO PRINT=LISTING NEW;
  %END;
  %ELSE %DO; /* For OS2, WIN, UNIX;
    FILENAME SASMACRO "&_ROOT\&FNAME..SMC";
    FILENAME VSEJOB "&_ROOT\&FNAME..VSE"
    LRECL=72;
    FILENAME MESSAGE "&_ROOT\&FNAME..MSG"
    LRECL=72;
    FILENAME LISTING "&_ROOT\&FNAME..LST"
    LRECL=&LISTRECL;
    LIBNAME DATADICT "&_ROOT\&DATADICT";
    %LET MINUMLNG = 3;
  %END;
%MEND SETFILES;
%SETFILES
```

As shown here, SETFILES is usually invoked immediately after it is defined. In this example, SETFILES defines some FILENAME and LIBNAME. It also defines a macrovariable MINUMLNG that requires a different value on different platforms. Also, the PROC PRINTTO is done only on one platform (CMS).

4. A Time-Saving Macro

If you find yourself repeatedly writing tedious code, you should ask if a macro could save some of the work. I often need to compare a character variable against several different character constants. Example: BEAST is a variable to be compared against constants COW, DUCK, HORSE, GOAT, GOOSE, SHEEP, DOG, CAT. In SAS 6.06 and beyond, the IN operator makes this reasonably easy:

```
IF BEAST IN ("COW", "DUCK", "HORSE", "GOAT",
"GOOSE", "SHEEP", "DOG", "CAT") THEN ....
```

Now I find typing a string like that tedious. Instead I use a macro that simplifies this to:

```
IF %INCHK(BEAST, COW DUCK HORSE GOAT GOOSE
SHEEP DOG CAT) THEN ....
```

Note that in typing the long string of animals I don't have to use the " and , anymore. Also, if that list of

barnyard animals is used more than once, then defining it as a global macrovariable saves even more effort:

```
%LET BARNYARD = COW DUCK HORSE GOAT GOOSE
SHEEP DOG CAT;
```

```
....
IF %INCHK(BEAST, %BARNYARD) THEN ....
```

Actually I first wrote the %INCHK macro when sometimes my programs were run under SAS 5.16. The IN operator is not available on that version, so I used a test against &SYSVER to determine SAS version, and then generated code that did or did not use the IN operator. Most of us now use version 6 SAS entirely, so here's the modern stripped-down version. You'll notice that INCHK itself uses another macro, CTOKEN (Count Tokens), which I include also.

```
%MACRO INCHK(V, C);
  /* Macro INCHK ('in check') generates code
  to check a character variable against a list
  of constants. Code generates 1 (true) if
  the variable is equal to any of the choices,
  otherwise, a 0 (false). Special convention
  - the string 'B' (quotes included) signifies
  the blank string. Note that otherwise
  quotes should not be used around the tokens.
  INCHK puts ( ) around the resulting expres-
  sion. This is so a prefix NOT may be used
  and the results will be as expected.;
  %LOCAL S D I N;
  %LET N = %CTOKEN(&C);
  %DO I = 1 %TO &N;
    %LET D = %SCAN(&C, &I, %STR( ));
    %IF %QUOTE(&D) EQ 'B' %THEN
      %LET S = &S, ' ';
    %ELSE %LET S = &S, "&D";
  %END;
  (&V IN(%SUBSTR(%QUOTE(&S), 2)))
  /* Removes leading ', ' and inserts
  code.;
%MEND INCHK;
```

```
%MACRO CTOKEN(LIST, SEP);
  /* This resolves to the number of tokens in
  LIST. SEP is the separator character - if
  absent, blank is used.;
  %LOCAL I RESULT;
  %IF %QUOTE(&SEP) EQ %THEN
    %LET SEP = %STR( );
  %LET I = 1;
  %DO %WHILE(%QUOTE(%SCAN(&LIST, &I,
    %STR(&SEP))) NE );
    %LET I = %EVAL(&I+1);
  %END;
  %LET RESULT = %EVAL(&I-1);
  %RESULT
%MEND CTOKEN;
```

There are several points worth noting in these macros.

First, note the use of the %LOCAL statement. This assures that the macrovariables named are *local* to the macro, and that the macro is not using macrovariables from a wider environment. Using %LOCAL is good practice, even when you are "pretty sure" that no

confusion can occur. Good macros tend to be used in contexts different from the original one.

Second, why did I use the macro CTOKEN in INCHK? I could have directly calculated N (the number of tokens in the list) directly in INCHK, or better yet, I could have used a %DO %WHILE instead of the %DO I = ... that would have avoided finding N altogether. I wrote it this way simply because CTOKEN is repeatedly useful as a service to other macros. It centralizes a function often needed. Note that macro CTOKEN is actually more general than needed here - an optional SEP parameter can specify a separator other than blank.

Third, the %SCAN and %SUBSTR macro functions operate very much like their datastep counterparts.

Finally, notice that in both macros the comments are about as extensive as the actual code! Macro coding is subtle, and extensive commenting is necessary so that others may understand. Actually, I find that I need extensive commenting so that I can understand what I wrote when I look at the macro even a few weeks later!

5. Faster Than a Speeding Binsearch

If you want to find whether a value is in an array, the ordinary technique is a linear search - just DO I = 1 TO N looking for the matching element. But if the array is long, and you check many elements against the array, this can use excessive CP resources.

If the array is *sorted*, then another technique is far more efficient: the Binary Search. Here's a simple macro that does it:

```
%MACRO BINSRCH(LIST, LNG, ELEMENT, INDEX);
  /* Searches array LIST, which is of length
  LNG, for ELEMENT. Returns the index of the
  match, or 0 if no match, in INDEX. LIST
  must be in sort. See MACRO SORT.;
  %LOCAL PFX;
  %LET PFX = %PFXGEN; /* Returns unique 4
  character string;
  %PFX.LO = 1;
  %PFX.HI = &LNG;
  /* Following is standard binary search
  algorithm;
  DO WHILE(%PFX.LO LT %PFX.HI);
    %INDEX = FLOOR((%PFX.LO + %PFX.HI) / 2);
    IF %ELEMENT GT %LIST(&INDEX) THEN
      %PFX.LO = %INDEX + 1;
    ELSE %PFX.HI = %INDEX;
  %END;
  IF %LIST(&PFX.LO) EQ %ELEMENT THEN
    %INDEX = %PFX.LO;
  ELSE %INDEX = 0;
%MEND BINSRCH;
```

The binary search algorithm itself is straightforward

enough - it occupies only 9 lines of code. One mysterious thing about this macro is its use of another macro, PFXGEN, to return "unique 4 character string." Why bother? Well, when BINSRCH is invoked in a datastep, if variables LO and HI were used (instead of &PFX.LO and &PFX.HI respectively), then this could cause trouble if the datastep itself were using variables LO or HI. Of course, the documentation could warn the user not to use LO and HI, but such warnings are often forgotten, so its better to try to write a bulletproof macro.

Suppose PFXGEN returns the string _01Q. Then &PFX.LO resolves to _01QLO, and &PFX.HI resolves to _01QHI. Here's the macro PFXGEN.

```
%MACRO PFXGEN;
  /* This macro returns prefixes of form _ccc,
  where the c are alphanumeric. Each invocation
  produces a new prefix. The global
  macrovariable PREFIX contains a counter, and
  is incremented each time used. It MAY be
  reset to null whenever a new datastep is
  begun, but this probably is not necessary,
  since there are 36x36x36 = 46,656 combina-
  tions that will be generated.;
  %GLOBAL PREFIX;
  %LOCAL A B C I J K;
  %LET PREFIX = %EVAL(&PREFIX + 1);
  /* Get the index of the first of the three
  characters. 1296 = 36**2;
  %LET I = %EVAL(&PREFIX/1296);
  /* Get the index of the second of the
  three characters;
  %LET J = %EVAL((&PREFIX-1296*&I)/36);
  /* Get the index of the third;
  %LET K = %EVAL(&PREFIX-1296*&I-36*&J);
  /* Get the three letters and form the
  resultant 4-character string;
  %LET A = %SUBSTR(&NUMERIC&ALPHA,
    %EVAL(&I+1),1);
  %LET B = %SUBSTR(&NUMERIC&ALPHA,
    %EVAL(&J+1),1);
  %LET C = %SUBSTR(&NUMERIC&ALPHA,
    %EVAL(&K+1),1);
  %A&B&C
%MEND PFXGEN;
```

In the BINSRCH macro, the macro SORT is mentioned. Here it is. It is not "easy," so I'll not go into details other than to mention that this is quite an efficient sort technique. On extensive tests, I've found it produces sorting times approximately proportional to $n\log(n)$, which is comparable to the best known sorts.

```
%MACRO SORT(LNG,COMPSWAP,OFFSET);
  /* General purpose sort. This sort method
  is a variant of shell sort. &LNG is the
  number of elements to be sorted. OFFSET is
  an optional parameter (if absent, assumed =
  1), that gives the first element to be sort-
  ed. %COMPSWAP is the name of a user-
  supplied macro that compares the element at
  the first parameter with that at the second,
  swaps them if out of order and returns the
```

third parameter=1 if the two elements are out of order, otherwise returns 0. User must supply COMPSWAP macro. COMPSWAP may include parameters, but then should appear in the invocation as %STR(name(parameters)). FAC1, FAC2, TRAN, and TEST may be adjusted for performance. However, FAC2 must be LE 0.5 and FAC1 must be GE FAC2. Also, CEIL(TRAN*FAC2) must be LT TRAN. TEST should be GT TRAN. The values given (0.7, 0.5, 10, and 100) were arrived at empirically, by experimentation with random data.;

```
  %LOCAL PFX ;
  %IF %QUOTE(&OFFSET) EQ %THEN
    %LET OFFSET = 1;
  %LET PFX = %PFXGEN;
  %LOCAL I FAC1 FAC2 TRAN TEST;
  %LET FAC1 = 0.7;
  %LET FAC2 = 0.5;
  %LET TRAN = 10;
  %LET TEST = 100;
  IF &LNG LE 1 THEN GOTO &PFX.SRT3;
  /* Nothing to sort if LNG le 1.;
  /* SPAN will be the distance between
  elements considered for swap;
  %PFX.SPAN = &LNG;
  %PFX.CHNG = 1; /* Records number of
    changes in a pass;
  %PFX.SRT1: /* Outer loop - adjust SPAN;
  /* Early in process, SPAN decreases by
  FAC1, later by FAC2;
  IF &PFX.SPAN GT &TRAN THEN
    %PFX.SPAN = CEIL(&PFX.SPAN*&FAC1);
  ELSE %PFX.SPAN = CEIL(&PFX.SPAN*&FAC2);
  %PFX.SRT2: /* This is start of main loop.
    In general, the same SPAN is
    used until it is not
    sufficiently productive.;
  %PFX.CHNG = 0;
  %PFX.UPPR = &LNG + &OFFSET - 1 - &PFX.SPAN;
  /* Upper limit for DO;
  /* Innermost loop - compares elements SPAN
  apart. Swaps if needed.;
  DO &PFX.LO = &OFFSET TO &PFX.UPPR;
    %PFX.HI = &PFX.LO + &PFX.SPAN;
    %COMPSWAP(&PFX.LO,&PFX.HI,&PFX.RES)
    %PFX.CHNG + &PFX.RES; /* RES returns 0
      if no swap, 1 if swap;
  END;
  /* Force SPAN adjustment while it is still
  GT 0.5*LNG;
  IF &PFX.SPAN GT 0.5*&LNG THEN
    GOTO &PFX.SRT1;
  /* Otherwise, if a large number of changes
  were found, or if at least one change and
  SPAN is small, then continue with same
  span.;
  IF &PFX.CHNG GT &TEST OR (&PFX.CHNG AND
    %PFX.SPAN LE &TRAN) THEN GOTO &PFX.SRT2;
  /* If not that, then so long as SPAN is
  GT 1 or there were changes, then adjust
  SPAN.;
  IF &PFX.SPAN GT 1 OR &PFX.CHNG THEN
    GOTO &PFX.SRT1;
  /* And if SPAN is 1 and no changes, we
  must be done sorting.;
  %PFX.SRT3: /* This label used when
    LNG LE 1 (nothing to do);
%MEND SORT;
```

Author information

David Sommer is the president of Optimal Systems, a consulting company specializing in SAS applications.

Optimal Systems, Inc.

10338 Colorado Road

Bloomington, MN 55438

voice: 612-831-5724

fax: 612-831-2765

Trademarks

SAS is a registered trademark of SAS Institute, Inc. in the USA and other countries. OS/2, CMS, VSE, MVS are trademarks of IBM Corporation.