If you have chosen to delve more deeply into the macro facility of the SAS® System, you probably started down the path by writing a few simple macros, and then noticing in the documentation that there are quite a few more capabilities than you are exercising. But, the intrepid journeyman macro programmer faces a variety of pitfalls: a bewildering set of abstruse quoting functions, multiple ampersand constructions, and a syntax which is just similar enough to regular SAS programming to get you into trouble.

This advanced tutorial will piggyback on (and assume prerequisite knowledge of syntax-oriented material contained in) other previous SUGI tutorials on the macro facility, but will focus on elucidating the specific macro constructs and techniques ("idioms of expression") used by guru-level macro programmers. Our major vehicle to impart these ideas will be to go through a case study in constructing a reasonably advanced macro, where we will broaden our scope to encompass implementation issues, including style, readability, and maintainability. These issues will focus the spotlight on some of the buts of advanced macro programming, discussing both the pros and the cons of certain techniques.

INTRODUCTION

A recent excellent and already influential book on the C++ language, Advanced C++: Programming Styles and Idioms [Coplien, 1992] makes the point that knowing the syntax of a programming language is not enough:

Programming language syntax shapes our thinking to a degree, but what we learn in the "owner's manual" about syntax alone only gets us started. Most of what guides the structure of our programs, and therefore of the systems we build, is the styles and idioms we adopt to express design concepts.1

Coplien goes on to note that traditional ways of teaching programming languages tend to focus on syntax and basic semantics, neglecting for the most part what he calls "fundamental building blocks," or reusable "expressions" of programming semantics.

The same observation also pertains to most SAS System manuals as well as to most SUGI tutorials, of course: they tend to focus on the basic (or even advanced) hows of wrestling with this huge baggy monster that we call the SAS System. Meanwhile, the new user (or the user who is new to a particular appendage of that huge baggy monster), faced with an intimidating pile of SAS manuals, is largely left to figure out his or her own idioms, ways to put the syntax together to create useful work. As I have noted elsewhere, the manuals are intended more as a general reference than as a programming instruction tool. They tend not to discuss the issues involved in fitting procedure and data step chunks together into a viable program.

The primary purpose of this tutorial, then, will be to propose some principles which are intimately familiar to any expert macro programmer, but which tend to appear confusing or mysterious to the uninitiated. The idioms of expression we will focus on here are:

- inline macros
- Interstep communication using CALL SYMPUT
- macro variable "arrays"

At the same time, however, we will look at issues of style and structure when macro programming. I will attempt to point out wise and unwise methods of proceeding. And in fact, some of the unwise ways imply something quite drastic: I will argue that much of the macro facility, ingenious though it may be, should be avoided when other and better techniques exist, techniques that we will discuss.

As a vehicle for showing alternative techniques as well as sound structuring principles, we will focus in the latter half of the paper on two separate, yet functionally identical, versions of a utility macro, DSDUMP, designed to list the contents and sample observations of every data set within a given data library. Programming this relatively simple functionality, the purpose of which is understandable by all, will illustrate most of the basic building blocks with which most useful macro programming can be accomplished. At the same time, we will discuss reasons for choosing one version over another, reasons that in a larger, "real world" example would be all the stronger. So the secondary purpose of this tutorial, then, will be to propose some principles of good macro design, and to put forward the notion that you should choose not to use some of the macro facility's capabilities.

BASIC MACRO IDIOMS

Inline Macros

The following examples illustrate what can be done with what I call "inline macros," or macros that generate what amounts to a single SAS expression, suitable for direct inclusion in a calculation, function call, or whatever. These inline macros become almost like extensions to the language, serving in essence as reserved words. Like most macros, when used carefully and when well documented, they can make SAS code more readable and maintainable, in that they provide a "block
be concerned with exactly how the macro is achieving its results.

The pair of macros that follow are simple examples of inline macros; they provide tools, akin to system-provided functions, for calculating the beginning date of a particular month and the ending date. These might be used, then, in subsequent calculations, such as:

```sas
PAYDAY = %begmonth() + 10;
PARTYDAY = %endmonth(SASDATE = OLDDATE);
```

Here is the text of the two macros:

```sas
%macro begmonth
(sasdate = date(),
 month = month(&sasdate)
);
mdy(&month,
 1,
 year(&sasdate)
)
%mend begmonth;

%macro endmonth
(sasdate = date(),
 month = month(&sasdate)
);
mdy((&month + 1) - (12*(&month=12)),
 1,
 year(&sasdate) + (&month = 12)
) - 1
%mend endmonth;
```

The simpler of these two macros, %begmonth, contains nothing more than a call to the SAS function MDY, with appropriate parameters. If the user supplies nothing as a parameter, the macro makes use of the current date. If we need to calculate the first day of the month in which a particular date falls, we supply the date (either via the name of a variable or via a SAS date literal, such as '05OCT56D') in the date parameter; witness the second example above. The month parameter need never be specified at all, since it logically must correspond completely to the month of the specified date; it is provided for reasons of potential efficiency in both of the macros, so that if a variable already exists containing the appropriate value, it may be provided by the programmer so that the macro is not forced to spend the necessary CPU time to recalculate it.

The second macro, %endmonth, has identical parameter guidelines and is, in fact, used in exactly the same manner as %begmonth. But it uses a couple of tricks to adhere to the requirement that it be usable inline, within a SAS expression. Since that requirement means that it cannot make use of IF/THEN logic, for example, %endmonth has to come up with alternative ways of doing conditional calculation.

The basic technique used to calculate the last day of a given month, as anyone knows who has thought about this common problem, is to calculate the first day (as a SAS date, of course) of the subsequent month, and then simply subtract one. Of course, if the base month is December (or month number 12), the subsequent month is January, or month number 1. The macro makes use of the fact that a conditional statement (e.g., MONTH = 12) will, in an arithmetic expression, evaluate to either 0 or 1, based on whether it is false or true. It uses the results of that evaluation as a multiplier by 12, to determine that we should subtract 12 from the result if we have come up with a 13 when we added 1 to the month. Of course, if we haven't come up with 13, we will multiply 12 by 0, and end up subtracting nothing. In the same manner, we will either add or fail to add 1 to the base year depending on whether the base month was December.

All of this somewhat contorted processing needs to take place within the bounds of a single SAS expression, so that we may adhere to the inline requirement and allow this macro to be used as a kind of SAS function. Admittedly, we may lose some efficiency (via possible double calculation of the month) if it turns out that we use the default MONTH parameter to the macro, rather than supplying the name of a variable that happens already to contain the month. However, it is quite possible that the SAS compilation process, now or in the future, will detect and optimize away that inefficiency, and in the meantime, we have won a great deal by simplifying the programming process through the construction of generic, reusable code.

Interstep Communication Using CALL SYMPUT

One of the chief powers granted through the macro facility is the ability to execute entire steps (DATA steps or PROCs) conditionally, by testing the value of macro variables. Those macro variables may be parameters to the macro, or they could be set via %LET, or they could be created in earlier steps through one of the chief "idioms of expression" available to the macro programmer: the use of CALL SYMPUT. This technique allows you to take information (data values) from within a DATA step and use those values actually to determine the flow of the rest of your overall SAS program, such as whether or not particular steps get executed at all, or what parameters are used within given PROCs. The macro facility would be considerably less useful if it lacked this ability to enable such interstep communication.

A simple example: if you maintain a counter within a DATA step (e.g., of all observations that meet a particular criterion), and then want to use that counter in subsequent steps, the typical idiom would look something like:

```sas
SET BLAHBLAH END=EOF;
IF EOF THEN CALL SYMPUT("COUNTER", COUNTER);
```

In all subsequent steps, the macro variable &counter will be available, containing the value that the COUNTER variable has accumulated by the end of the DATA step.

A typical mistake of new macro programmers, please note, is to forget that this macro variable is only available after the current step (the one containing the CALL SYMPUT) has executed.
We chose, in this case, to call that macro variable &counter, but we could have called it anything we wish, or we could have supplied, as the first parameter to CALL SYMPUT, the name of a data step variable that contained the name we wanted the macro variable to be. That means that we can build macro variable names on the fly, based on information in the DATA step. Our second, more complex example illustrates such a construction:

```
LENGTH VARNAME %8;
VARNAME = 'MEM' || LEFT(PUT(COUNTER, 3.));
CALL SYMPUT(VARNAME, MEMNAME);
```

In this case, we are building a macro variable name of the form &MEMn, where n is determined by the value of the variable COUNTER at runtime. Note that you do need to perform a LEFT() function on the numeric variable COUNTER (and in this case, we also PUT it explicitly into character format), so that the "splicing" happens correctly and the result in VARNAME is the name of a legal SAS macro variable (i.e., no spaces!). Ability to do this on-the-fly construction anticipates our next major idiom, the concept of macro variable arrays.

**Macro Variable "Arrays"**

The principal piece of knowledge that separates the expert macro facility programmer from the novice is the ability to manipulate what I term macro variable "arrays." Of course, arrays do not exist in the macro language as a syntactical entity, hence groups of variables (e.g., &MEM1, &MEM2, &MEM3, and so on) must be dealt with via a common programming convention, which allows them to be thought of as logical arrays. Much of what is typically done, in fact, in complex macros that interface tightly with SAS data sets and DATA steps, consists of the programmer putting information from within the DATA step into such macro variable "arrays" and then making use of that information later on.

Let's go at this one backwards, first showing the idiom of expression, then explaining what it all means. Here's a typical construct using a macro variable array:

```
%do counter = 1 %to &total;
  PROC PRINT DATA = &FILE&counter;
  TITLE "Contents of &FILE&counter";
  RUN;
%end;
```

This code presupposes the existence of a set of macro variables, named &FILE1, &FILE2, &FILE3, etc., with the total count of such variables contained in the macro variable &TOTAL. Contained in each of those &FILEn variables, in this particular case, are the names of SAS data sets on which this code will run PROC PRINT.

How does the code work? The shell is obviously a simple macro %DO loop, which will generate however many PROC PRINTs are reflected in the value of the &TOTAL variable. When the SAS supervisor, parsing through the program code, reaches the DATA = &FILE&counter statement, it will pass control to the appropriate routine to try to resolve the macro variable reference, using the standard algorithm: First, it resolves the & at the front of the reference to a single &. Then, it attaches the characters FILE to that, followed by the current value of the variable &COUNTER. The result, for the first pass through the loop, will be &FILE1, for the second pass, &FILE2, and so on. When it finishes the initial substitution in each case, it determines that it is not yet done, since the result still contains an ampersand, so it will resolve the &FILE1, for example, to get the value of that macro variable. Of course, the same thing happens in the TITLE statement, just as long as you remember to use double quotation marks, not single quotation marks.

**MACRO STYLE GUIDELINES**

As I have written elsewhere [Kretzman, 1988], style is surprisingly important in the design of macro tools, especially if they are intended to be widely used. In all but the rarest cases, all macro tool parameters should be coded as keyword parameters (e.g., BLAHBLAH=YES) rather than just plain positional parameters. This practice allows the ready use of defaults, permits the parameters to be used in any order in the calling program, and is considerably more readable both in the macro definition and in the macro call. It imparts a measure of self-documentation to the macro, especially if the parameter names are well chosen.

But how do you determine what should be a macro parameter? The purpose of a generalized macro tool is to serve as a black box, and preferably as a black box for all similar, yet not necessarily identical, needs. You want to be able to adjust the behavior of the black box in minor ways without having to alter (or even be aware of) the code within it. That means that one of your design tasks is to determine what kinds of things you would want to change in the code to make it behave in slightly different ways: a file name, a constant, a switch setting to turn certain behavior on or off. It is not necessary that the typical usage of the macro entail specifying each and every parameter; in fact, many parameters might rarely be used, and would be coded with suitable defaults, "just in case." A typical example is the DEBUG= parameter which is present in many or most of the macros I write. If that parameter is coded to anything other than an N (which is the default), then diagnostic messages are produced, along with test PROC PRINTs, etc.

In addition to having a meaningful, self-documenting name, of course, each macro parameter should be fully commented in the macro definition, describing its purpose and its typical and default values. In all but the rarest cases, macro parameters should be given reasonable default values, so that the user is never forced to specify a parameter unless the default behavior is to be changed.

If a macro is to be most useful, it is critical that it be documented rigorously, both internally (in the form of comments) and externally. I recommend especially the establishment of a generalized "header" block of comments, attached to the front of every macro, containing the name, author, revision dates, calling guidelines, and so on.

Every macro should take special care to "clean up" after itself, deleting any work data sets that it may have generated. In addition, these work data sets should be given names that are unlikely to conflict with those of the user's program: using the underscore as an initial character is a good convention to achieve this goal. One recommended macro parameter, typically coded as DELETE=Y, could be a switch that prevents this cleanup if the user (usually the macro developer) so desires, so that testing may take place.
THE DSDUMP MACRO: GENERAL APPROACH

Our major vehicle to illustrate some of the idioms of expression we have covered so far will be to go through a case study in constructing a reasonably advanced macro, where we will consider further implementation issues, including style, readability, and maintainability. We will do this by constructing two functionally identical versions of the case study macro, one which uses some of the idioms of expression discussed earlier, and one which avoids some of those idioms in favor of other techniques.

The simple objective of both versions of this illustrative macro will be to print the contents and selected number of observations for each of the data sets in a given SAS data set library. In both cases, we will use PROC CONTENTS, with an output data set, to generate an electronic list of the data sets in the specified data set library. Then, we will generate SAS code (either via the macro facility, as in Alternative A, or directly to a flat file, as in Alternative B) that performs the PROC PRINT and PROC CONTENTS on the individual data sets.

Although the second alternative avoids some of the macro techniques used in the first alternative, both versions are macros, using the "Teflon and Tang" kind of style that I have proposed here and elsewhere. For example, each alternative has a well-defined set of parameters, with appropriate defaults, meaning that the user could invoke the macro with no parameters specified at all, and, assuming that the libname DATA was defined, would get viable results rather than an error message.

Note that neither version happens to make use of inline macros; once such macros are established, of course, they may be used anywhere that SAS functions and other expressions may be called, so I saw no purpose in intentionally molding this case study just to make use of them.

THE DSDUMP MACRO: ALTERNATIVE A

The first version of the DSDUMP macro uses the macro facility to generate the SAS code that will actually perform the PROC CONTENTS and the PROC PRINT on each of the data sets that it finds. DSDUMP is defined with five keyword parameters, with the following defaults:

```
%macro dsdump
  (libname=DATA,
   numobs=10,
   format=Y,
   delete=Y,
   title='DSDUMP Macro')
```

These keywords, used throughout the macro in the form of macro variables, enable the user to achieve a great deal of flexibility from this black box: you can process any data library, print any number of observations, use or not use the internal data set formats, choose whether or not to "clean up" after the processing of the macro, and place any desired title on the contents and prints of the respective data sets. Any combination of these settings may be employed without having to touch the code within the block box.

Here is the text of the macro, with a detailed discussion following.
When this step is complete, we have a bunch of macro vars:

- MEM1 fred
- MEM2 harry
- MEM3 betty
- MEM4 jane

-------------------

DATA NULL;
SET ALL END=EOF;
BY MEMNAME NOTSORTED;

LENGTH VARNAME $8; * This var will hold the current macro var name;
IF LAST.MEMNAME; * Get only one occurrence per member;
RETAIN CTR 0; * Counter for macro variables to be generated;
CTR + 1;
VARNAME = 'MEM' || LEFT(PUT(CTR, 3.));
* put member name into this macro var;
CALL SYMPUT(VARNAME, MEMNAME);
* preserve total # of members in a macro variable too;
IF EOF THEN
CALL SYMPUT("TOTALMEM", LEFT(PUT(CTR,3.));
* do ctr = 1 %to &totalmem;

RUN;
%do ctr = 1 %to &totalmem;

PROC CONTENTS
DATA=&libname..&MEM&ctr;
TITLE1 &title;
TITLE2 "Contents of &libname..&MEM&ctr";
RUN;

PROC PRINT
DATA=&libname..&MEM&ctr(OBS=&numobs);
%if %upcase(&delete) eq Y %then
%do;
FORMAT_NUMERIC;
FORMAT_CHARACTER;
%end;
TITLE1 &title;
TITLE2 "Contents of &libname..&MEM&ctr";
RUN;
%end;
%end dsdump2;

The first thing the macro needs to do is generate a SAS data set containing a list of members contained in the referenced libref, and it does so via a PROC CONTENTS with an output data set. The next task is to get all those members into an array of macro variables, which is done within a DATA step by using building macro variable names of the form MEMn and giving them the value of each member name by using SYMPUTs. The same counter that is used to compose the current macro variable name (of the form MEMn) is then written, at the last observation, into another macro variable called &TOTAL. When we exit this DATA step, then, we have a full symbol table, consisting of a set of member names in the variables &MEM1, &MEM2, &MEM3, etc., along with a variable called &TOTAL containing how many of those member variables there are. In short, we have performed some interstep communication. We can now use this information in subsequent steps.

Now that we have those macro variables, we can call PROC CONTENTS and PROC PRINT from within a macro %DO loop, counting from 1 to the value of &TOTAL. The only tricky part is telling the PROCs what member we’re talking about. This requires dereferencing the macro variable array, as discussed earlier, using the construct &&MEM&ctr. Look again at one example of this:

PROC CONTENTS
DATA=&libname..&MEM&ctr;
TITLE1 &title;
TITLE2 "Contents of &libname..&MEM&ctr";
RUN;

Note that we also make use of several of the supplied keyword parameter variables, such as &libname and &title. The “extra” dot contained in the DATA=&libname..&MEM&ctr reference is required, for the macro facility regards a dot at the end of a macro variable reference as a delimiter, and we ourselves need a dot in the generated SAS code data set reference, of the form DATA MEMBER.

The rest of the %DO loop calls PROC PRINT, with some conditional processing depending on whether we have turned the “switch” on for using the data set formats (the default). After all the data sets are processed, then depending on the setting of the DELETE switch, the macro cleans up after itself, deleting the work data set it generated in the initial PROC CONTENTS on the referenced libname.

Now here’s the style issue: as macros get more and more contorted and involved, they tend to make ever greater usage of constructs such as the example shown in the box above: DATA=&libname..&MEM&ctr. The more complex the need, the
more your code acquires triple ampersands, quoting functions, multiple dots, etc., all necessary to "get it to work." Readability and maintainability can become significant problems as complexity increases. Producing certain patterns of SAS code via the macro facility requires the use of arcane quoting functions, each designed to handle yet one more loophole that has emerged as people use the macro facility to its limit. Worst of all (and I have experienced this again and again, moving from Release 5 to Release 6.03, then to Release 6.06), macro functions and features can have subtly different behavior from release to release, meaning that the more complex a macro is, the more likely it is to break during a release upgrade.

For these reasons, I believe that one should Just Say No to overuse of the macro facility. If you can't tell the macro quoting functions apart without pages of explanation and examples, for example, maybe they should be avoided altogether. The astonishing truth is that avoiding them is possible. The next example illustrates that functional objectives can generally be met even if one is limited to the most basic macro coding techniques: simple interstep communications via SYMPUT, use of basic macro parameters, and the use of a simple key technique: write SAS code to a flat file and then %INCLUDE it.

THE DSDUMP MACRO: ALTERNATIVE B

This second version of the DSDUMP macro manages to avoid using any macro variables other than those defined as keyword parameters by the macro itself. Rather than going through a separate step of loading macro variables with the names of the members discovered in the referenced libref, this version deals with those names directly, writing out SAS code to a flat file. The key insight here is that any SAS program can write another SAS program, and it doesn't need to use the macro facility to do it. Here's the text of the macro:

```
* Purpose: Prints contents and selected obs of all data sets within a data base
* This version uses a flat file for output rather than pure macro concepts.
*****************************************************;
* ----- THE DSDUMP MACRO ---- *
* PROGRAM NAME: DSDUMP *
* AUTHOR: Peter Kretzman *
* DATE CREATED: 3/15/88 *
* LAST REV : DESCRIPTION *
* DATE BY OF REVISION*
* ----------------------------------- *
* Purpose: Prints contents and selected obs of all data sets within a data base *
* This version uses a flat file for output rather than pure macro concepts. *
*****************************************************;
%macro dsdump
   (libname=DATA, /* LIBNAME of data base to
   numobs=10,  /* # of obs to print for each data set */
   format=Y,  /* Use the formats in data set? */
   tempcode=TMP CODE, /* Data set name for temp. code */
   delete=Y, /* Delete the work code when done? */
   title='DSDUMP Macro' /* Title for PROC PRINT (in tick marks) */
) ;

%*****************************************************;
%* Get a list of all members in the referenced libref. *
%* --------------------------------------------------- *
%*****************************************************;
PROC CONTENTS DATA=&libname..ALL_ MEMTYPE=DATA OUT= ALL(KEEP=MEMNAME) NOPRINT
RUN;

%*****************************************************;
%* Now write out code to a flat file. Each data set in the referenced libref will get a *
%* PROC CONTENTS and a PROC PRINT. *
%* ----------------------------------- *
%*****************************************************;
DATA _NULL_;
   FILE "&tempcode" NOTITLES;
   SET ALL;
   BY MEMNAME NOTSORTED;
   IF LAST.MEMNAME;
   * Get only one occurrence per member;
   PUT "PROC CONTENTS DATA=&libname.. " MEMNAME " ;;";
   PUT "title1 &title; ;"
   PUT "title2 'Contents of &libname.. " MEMNAME " ;";
   PUT "RUN; ;"
   PUT "PROC PRINT DATA=&libname.. " MEMNAME " (OBS=&numobs) ;;";
   %if %-upper-case(&format) ne Y %then %do;
      PUT "FORMAT NUMERIC; ;"
      PUT "FORMAT _CHARACTER_; ;"
   %end;
   PUT "title1 &title; ;
   PUT "title2 'Contents of &libname.. "
```

181
MEMNAME ";";
PUT ";RUN;";
RUN;
%*****************************************************
%*
% Now run the code that was just
%* generated.
%*
%*****************************************************
%include ";tempcode"
%
%if %upcase(&delete) eq Y %then
%do;
x "erase &tempcode"
PROC DATASETS LIBRARY=WORK
NOLIST;
DELETE _ALL;
RUN;
%end;
%mend dsdump;

This version differs from the first version mostly in the middle area, where instead of creating SAS macro variables and then going through a %DO loop, we write out code to a flat file. You'll notice that another keyword parameter has been added to the macro, allowing the caller to specify, if so desired, exactly where the temporary flat file code will be placed. Again, this would be a parameter that would be used mostly for testing.

The chief advantage of operating in this manner is simplicity of approach. Although constructs such as PUT "PROC CONTENTS DATA=dbname." MENNAME ";"; are not exactly unmuddy, they are still easier to code and understand, and significantly more robust to changes or release upgrades, than their pure macro equivalents. What's more, debugging the results will be considerably easier, since the generated code is real code, with associated line numbers at run time, and so on. If you have doubts as to what was generated in a particular instance, you can always go to the intermediate scratch flat file and peruse the code for yourself. With the macro facility, even given the excellent debugging enhancements and options that are now part of the SAS System, you're operating relatively blind.

CONCLUSION

Ever notice that "macro" and "macho" differ by only one letter? I once had a programmer proudly show me his intricate macro code, code that actually wrote macros conditionally and which was laden with triple ampersands and replete with quoting functions. Macro programming has almost become the macho side of SAS programming: the more complex, unreadable, and unmaintainable your macro code is, the more self-worth you feel as a programmer, having wrestled the bear down and then made him dance.

But like any powerful tool, macro programming requires steadfast discipline to use effectively. I believe that one should choose to exercise such discipline, avoiding in one's code the sublime and the ridiculous, in the interests of readability and maintainability. And in this case, that means gauging carefully just when, and to what extent, you will use the more abstruse areas of the macro facility.

Comments, suggestions, and questions are welcome. The author may be reached at the following address:

Peter Kretzman
McCaw Cellular Communications, Inc.
P.O. Box 97050
Kirkland, WA 98033-9780
Voice: (206)628-1344
FAX: (206)628-1300

In addition, the author may be contacted via internet, using the address mcgp1peter@cs.washington.edu

BIBLIOGRAPHY


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