WHO SHOULD TEACH, WHAT TO TEACH, HOW TO TEACH: AN ILLUSTRATION USING THE SAS® MACRO FACILITY AS AN EXAMPLE

Erika M. Stone
Carolina Population Center
The University of North Carolina at Chapel Hill

ABSTRACT

This paper addresses two questions: first, the general question of how to provide effective instruction in advanced SAS programming to a given group of SAS users and, second, the specific question of how to introduce the complex subject of the SAS macro facility.

The proposed solution to the first question is to encourage experienced programmers, working in the same environment as the intended audience, to develop and teach classes. The benefits of teachers who are programmers belonging to the same group they are teaching are many, and go beyond the savings in costs associated with providing outside instructors.

The second question is addressed by presenting excerpts from a programmer-designed introduction to the macro facility. The setting is a university research environment; the audience consists of students and programmers who use SAS to process and analyze survey data. The emphasis is on getting across two concepts: the role of the macro processor as a text generator and the idea that the macro facility can help us write better programs—programs that are systematically debuggable and testable, shorter, clearer, and more general. Simple examples, drawn from situations familiar to the audience, are used to illustrate key ideas. They are presented in a light fashion against a background of a running dialog between the programmer and the macro processor.

INTRODUCTION

How to best provide SAS instruction, especially in more advanced programming topics, is a perennial question in groups within an academic setting where budget constraints are ever-present. At the Carolina Population Center, part of The University of North Carolina at Chapel Hill, we have been gradually developing special topics classes based on the belief that experienced programmers who like to teach can make effective teachers. The benefits of having classes taught by programmers belonging to the same group as the students they teach reach beyond the savings in costs incurred by hiring outside instructors.

This paper illustrates an intuitive way of introducing the SAS macro facility. It is drawn from a tutorial presented at the first SouthEast SAS Users Group conference (Stone, 1993). The same tutorial has been given as an introduction to the SAS macro facility to an audience of graduate students and programmers at The University of North Carolina at Chapel Hill. It is aimed at SAS programmers of an intermediate level who have either not used the macro facility at all or used it only a little.

THE SETTING AND THE STRATEGY

Programmers, students and faculty at the Carolina Population Center work with survey data. Data comes to the Center from all over the world, either as the result of original data collection efforts on projects or from large national and international surveys which have been released to the public for research purposes. Data sets range in size from several thousand observations to millions of observations. Recognizing the superior ability we have with SAS to manipulate, transform, combine, and analyze data, the data on all major projects at this Center is subject to SAS processing.

The SAS macro facility can be approached from different angles. The emphasis in a first introduction is best placed on understanding the role of the macro processor as a text generator and showing what the macro facility can do for you. Specifically, I focus on three benefits of the macro facility that relate directly to the writing of better programs. These are the possibility of 1) writing shorter programs by reducing the amount of keyed text, 2) writing programs that are systematically debuggable and testable through a modular approach, and 3) writing program code which is general, flexible, and reusable. The result can be programs that run correctly in less programming time and who isn’t interested in that?

Given the particular programming environment at the Carolina Population Center, simple but real examples are chosen from survey data since this is the most familiar context for the audience. The examples are presented in a light mode, arising from situations familiar to the audience. In each situation there is a choice—to use the macro facility or not. The macro processor, in a dialog with the programmer, advises on how to decide.

Before presenting specific examples, however, the essential principles by which the SAS macro facility functions as a text generator are explained, beginning with the difference between programs with and programs without macro activity. The sections which follow are excerpts from the tutorial referred to above and developed as an introduction to the macro facility.

HOW TO THINK OF THE SAS MACRO PROCESSOR AS A TEXT GENERATOR

Programs with and without macro activity

A program that does not make use of the SAS macro facility is said to be "without macro activity." Such a program does not request work from the macro processor, one of the two components, along with the macro language, of the
When you submit a program without macro activity to the SAS system, you think primarily of communicating with the SAS compiler—that part of the system that checks the syntax of your SAS language statements and translates your program, one DATA or PROC step at a time, into a form the computer can execute. While you have no choice in the matter of dealing with the SAS compiler, whether or not you deal with the SAS macro processor is up to you to decide.

In a program with macro activity, recognizable by the use of the symbols % and &; you communicate not only with the compiler, but also with the SAS macro processor. You can think of the macro processor as a part of the SAS system separate from the compiler and between you and the compiler. With macro language statements, you can request the macro processor to help you build the program code that the compiler will see. The principles by which the SAS macro processor functions as a text, or code, generator can be illustrated by very simple examples.

Letting the macro processor generate code

Suppose you want a 2-way frequency table for the variables A, B in the SAS data set ALPHA stored in a SAS data library referenced by IN. Let's look at three different ways you could write your program and achieve the same result—the first (Version 1.) without macro activity, and the second and third (Version 2. and Version 3.) with macro activity:

Version 1. Program without macro activity

```sas
PROC FREQ DATA=IN.AlPHA;
TABLES A*B;
RUN;
```

Version 2. Program with macro activity

```sas
%MACRO Tbl;
PROC FREQ DATA=IN.AlPHA;
TABLES A*B;
RUN;
%MEND Tbl;
%Tbl
```

Version 3. Program with macro activity

```sas
%MACRO Tbl(member);
PROC FREQ DATA=IN.&MEMBER;
TABLES A*B;
RUN;
%MEND Tbl;
%Tbl(ALPHA)
```

In Version 1., the compiler compiles the two statements of the PROC step and the program is executed. In Version 2. and Version 3., we achieve the same result by requesting the macro processor to generate the code that the compiler will see.

In Version 2., the complete PROC step is packaged into a unit of code called a "macro" to which the name TBL is given. The macro language statement %MACRO TBL; begins the definition of the macro named TBL, and the statement %MEND TBL; ends it. The body of the macro in this example consists of pure SAS language statements.

When the program in Version 2. is submitted to the SAS system, the percent sign, followed immediately by the word MACRO, activates the macro processor, causing it to compile the macro and store it for possible later use. The %TBL statement triggers the macro processor again, this time requesting it to execute the macro TBL. The execution of the macro TBL causes the SAS language statements which form the body of the macro to be inserted at the point where the macro is called, or invoked, by the statement %TBL and thus placed in the stream of text directed to the compiler. The compiler never sees the statements addressed to the macro processor; instead, it sees the text generated by the macro processor, namely the code:

```sas
PROC FREQ DATA=IN.ALPHA;
TABLES A*B;
RUN;
```

In Version 3., the macro TBL is written with one parameter named MEMBER. A parameter is a special case of a macro variable. Such variables belong to the macro language and their values are strings. In the body of the macro TBL the symbol &MEMBER refers to the current value of the macro variable MEMBER. The current value of MEMBER is determined from the statement that invokes, or calls, the macro. That is, the statement `%TBL(ALPHA)` asks the macro processor to execute the macro TBL using the string ALPHA as the value of the macro variable MEMBER. The result is that the code the compiler sees is again exactly the same as in the first two versions of the program.

Macros written with parameters, as in Version 3. above, are flexible. They can be invoked many times in one program with different values for the parameters. In this case macro TBL could be invoked more than once in the same program to produce a sequence of 2-way frequency tables for the variables A, B in different SAS data sets in the same SAS data library.

The three programs above, which result in identical executable code, differ in how much work the macro processor is requested to do. In the first example, the macro processor is ignored and hence does no work; in the second and third examples the macro processor is engaged to generate the code—in this case all the code—that will be compiled and executed. It is this text-generating capability of the macro processor that gives us the potential to write not only more interesting, but also better, programs, as we will see in the sections which follow.

THE POTENTIAL OF THE SAS MACRO FACILITY ILLUSTRATED

Introducing the characters

All the examples which follow come from work with survey data. We will introduce them in a sequence of scenarios in...
which there will be three characters: you and your program, the SAS macro processor (MP), and, in the background, the SAS compiler. In each situation you will have a choice: to use the macro facility or not. The SAS macro processor, endowed with a power of speech we all wish it had in real life, will tell you which way to go.

First example: breaking up a tedious task

In this example you will see macros used in their simplest form—i.e., to package portions of pure SAS code—in order to systematically construct, test, and debug a DATA step which is conceptually simple but extremely tedious.

Imagine that you have obtained raw data from abroad, data at the community level for a country you are studying. You want to convert this data as quickly as possible into a SAS data set. Conceptually, the task is as simple as can be, yet on closer examination it has the following characteristics which make it more tedious than you like.

There are seventeen records per case (per community) and close to one thousand variables. The variable names have been selected to facilitate easy referencing of the variables in the survey documentation. The problem is that you have about one thousand names to key and doing so will be very tedious since they look like this:

```
SA1 SA2 ... SA7_9 SA10 13 ... SA87 89 ... SA94 96 ...
SB10_12 SB94 96 ... SG31 32 SG33 ... SG34 35 ...
SP15 16 SP17_18 ... SP94 95 ... SQ25 SQ26 SQ27 ...
```

Your task to create a SAS data set from this data boils down to writing a tedious and error-prone INPUT statement.

You: How can I proceed in a systematically debuggable way?

MP: Break it up! Break it up!

You: How?

MP: Focus on the variables coming from one record of data at a time.

You read in the "SAS Guide to Macro Processing" (which you discover stands beside the "SAS Language Reference" on your bookshelf) how you can package code into a macro. You decide to package up the variable names and column locations from one record at a time into separate macros, one per record, and test your INPUT statement in stages. Your macro for the variables from the first record, which you name LINE1, looks like this:

```
%MACRO LINE1;
SA1 SA2 ... VNUM & 3-4 VNAME & 5-6 SA7_9 7-9 SA10 13 ...
SA14 16 14-16 SA17 19 17-19 SA20 22 21-22 SA23 ...
SA25 27 SA26 28 SA29 30 29-30 SA31 32 31-32 SA33 ...
SA34 35 SA36 37-38 SA39 40 39-40 SA41 42 ...
SA43 44 45 44 45 46 47 ...
SA48 49 50 51 52 53 54-55 SA56 57 56-57 SA58 ...
%MEND LINE1;
```

You test a partially constructed INPUT statement, reading only the variables from the first record (of seventeen records) for each case, by invoking the macro LINE1:

```
OPTIONS MPRINT;
DATA VILL88;
FILE IN;
LENGTH DEFAULT=4;
INPUT
#1 %LINE1
#17 ;
RUN;
PROC CONTENTS;
PROC FREQ;
```

As a result of the MPRINT system option, you can see in the SAS log exactly what the macro processor does for you. The macro processor generates code which is inserted into your DATA step at the point of invocation of your macro and before the compiler compiles your DATA step. The code in this case consists of the body of the macro LINE1 and appears in the SAS log following the call to macro LINE1:

```
DATA VILL88;
FILE IN;
LENGTH DEFAULT=4;
```

```
OPTIONS MPRINT(LINE1): SA1 SA2 2 VNUM & 3-4 VNAME & 5-6
SA7_9 7-9 SA10 13 10-13 SA14 16 14-16 SA17 19
SA26 28 SA29 30 29-30 SA31 32 31-32 SA33 34
33-34 SA35 36 35-36 SA37 38 37-38 SA39 40
39-40 SA41 42 41-42 SA43 44 43-44 SA45 46
SA47 ...
```

```
RUN;
```

Using the Proc Contents listing and the frequency tables produced by your run, you correct any bugs in this part of the INPUT statement. After debugging your program to this point, you add a second macro named LINE2 which pack-
ages up the names and column locations for the variables from the second record. To test the next part of the INPUT statement, you comment out the first macro invocation and run the DATA step as follows:

```
DATA VILL88;
INFILE IN;
LENGTH DEFAULT=4;
```

```
INPUT
   #1 %LINE1 */
   #2 %LINE2
   #17
RUN;
```

```
PROC CONTENTS;
PROC FREQ;
```

You move systematically through the variable names and their column positions building the INPUT statement and testing and debugging as you go along. Finally you are ready to create the permanent SAS data set with the following program:

```
/* * * MACRO DEFINITIONS * * */

%MACRO LINE1;
   .
%MEND LINE1;

%MACRO LINE2;
   .
%MEND LINE2;

%MACRO LINE3;
   .
%MEND LINE3;

%MACRO LINE17;
   .
%MEND LINE17;

/* END MACRO DEFINITIONS */

DATA OUT.VILLS8;
INFILE IN;
LENGTH DEFAULT = 4;
```

```
INPUT
   #1 %LINE1 */
   #2 %LINE2
   #17
RUN;
```

```
PROC CONTENTS;
PROC FREQ;
```

Note that the program code is in two sections—first the macro definitions and then a main part in which the macros are invoked. The INPUT statement is constructed in stages using macros as building blocks. The main motivation for engaging the macro processor in this task is to break up a tedious INPUT statement and to test and debug it systematically, focusing on one part of the problem at a time.

Second example: taking advantage of a pattern in names

In this example the macro processor is used to generate variable names and thus reduce the amount of keyed program text.

Your task is to make a large selection of variables from an existing SAS data set in which the variables have awkward names (through no fault of your own). This is a large national survey for which the data has been released as a SAS data set with variable names selected to relate to a survey instrument. Again, your task is conceptually simple but it is tedious and error-prone in the following sense: you want to insert 193 variables names into the KEEP = option in the SET statement and the variable names look like this:

```
RTIID F17E1BM F17E1BY F17E1EM F17E1EY F17E2BM F17E2BY F17E2EM F17E2EY F17E3BM F17E3BY F17E3EM F17E3EY F17E4BM F17E4BY F17E4EM F17E4EY ..... F11SB4BM F11SB4BY F11SB4EM F11SB4EY ..... F12DB1BM F12DB1BY ..... 
```

The problem is you don’t really want to key all those names. You hear the macro processor:

```
MP: Try to find a pattern! Try to find a pattern!
You: What kind of pattern?
MP: That’s up to you.
```

So you stare at the variable names and patterns begin to emerge. Logically, besides the variable RTIID, the variables happen to fall into three groups. They are the respondent’s job related dates (128 variables), unemployment related dates (32 variables), and school attendance dates (32 variables). In the first group of 128 variables the following pattern repeats itself three more times with 8, 9, and 10 in place of 7 in the third character position of the variable names:

```
```
You: OK, now that I see patterns, what next?

MP: You have to tell me how to generate the patterns.

You: How do I do that?

MP: With the SAS macro language.

So you search through the "SAS Guide to Macro Processing" again. . . . After some time you come up with a sequence of %DO loops and you want to see if the macro processor understands your macro language instructions to generate the variable list. You begin to key the first nested %DO loops inside your DATA step like this:

```
DATA ACTDATA;
SET INSAS.FU6NLS;
%DO J=7 %TO 10;
%DO I=1 %TO 8;
    FJ&J.E&I.BM FJ&J.E&I.BY
    FJ&J.E&I.EM FJ&J.E&I.EY
%END;
%END;
%MEND JOBS;

%MACRO NOJOBS; /*generate 32 variable names*/
%DO I=1 %TO 8;
    FJ&I.B&I.BM FJ&I.B&I.BY
    FJ&I.B&I.EM FJ&I.B&I.EY
%END;
%MEND NOJOBS;

%MACRO SCHOOLS; /*generate 32 variable names*/
%DO J=19 %TO 20;
%DO I=1 %TO 4;
    FJ&J.B&I.BM FJ&J.B&I.BY
    FJ&J.B&I.EM FJ&J.B&I.EY
%END;
%MEND SCHOOLS;
```

DATA ACTDATA;
SET INSAS.FU6NLS(KEEP = RTIID %JOBS %NOJOBS %SCHOOLS);

The main motivation for engaging the macro processor in this task is to reduce the amount of keyed text. In fact, except for the variable RTIID, you don't have to key a single one of the other 192 variable names that you do not want to key. The macro processor generates the variable names from your macro language instructions and inserts the names into the KEEP= option at the point of invocation of your macros and before the compiler compiles the DATA step.

Third example: a useful reusable macro

In this example a macro with parameters is used as a building block for a DATA step and results in reusable code.

Your SAS data set contains the birth dates of children born to the respondents in a survey of women and you want to examine child spacing; that is, the lengths in months of the intervals between successive births. Each observation in your data set has the month (1 to 12) and year (2 digits) of birth of up to five children in variables M01 YR1 M02 YR2 M03 YR3 M04 YR4 M05 YR5. You want to do what some sociologists like to do, i.e., define new variables DOB1 DOB2 DOB3 DOB4 DOB5 which are the children's birth dates in century months.

To get century months, we apply the rule: century month = 12 x year + month, where year takes values from 0 (for 1900) to 99 and month from 1 to 12. Easy enough. For the first birth, the assignment statement DOB1 = 12 *YR1 + M01; defines the century month DOB1. But then there are the 2nd, 3rd, 4th, and 5th births; and what about dates of marriages, separations, and divorces which you may also want to express in century months? Should you repeat the assignment statement as many times as you need to (that's one solution) or . . . .

You hear the macro processor:

MP: Use a macro with parameters! Use a macro with parameters!

After consulting the "SAS Guide to Macro Processing" once more, you write the macro CM with three parameters and a DATA step which invokes it five times, once for each birth date:

```
OPTIONS MPRINT;
%MACRO CM(MONTH,YEAR,CMDATE);
%*convert MONTH, YEAR to century;
%*month date CMDATE;
&CMDATE=12*YEAR + &MONTH;
%MEND CM;
```

DATA CMBIRTHS;
SET IN.BIRTHS;
%CM(M01,YR1,DOB1)
%CM(M02,YR2,DOB2)
%CM(M03,YR3,DOB3)
%CM(M04,YR4,DOB4)
%CM(M05,YR5,DOB5)
RUN;
The MPRINT system option shows you the work of the macro processor. With the first call to the macro CM, the variable name MO1 becomes the value of the parameter MONTH, the variable name YR1 becomes the value of the parameter YEAR, and the new variable name DOB1 becomes the value of the parameter CMDATE. Analogous substitutions of variable names for parameters occur with each succeeding invocation of the macro CM. In the SAS log you see the code generated by the macro processor:

```
DATA CMBIRTHS;
SET IN.BIRTHS;
%CM(M01,YR1,DOB1)
MPRINT(CM): DOB1 = 12*YR1 + MO1;
%MCMO2,YR2,DOB2)
MPRINT(CM): DOB2 = 12*YR2 + MO2;
%MCMO3,YR3,DOB3)
MPRINT(CM): DOB3 = 12*YR3 + MO3;
%MCMO4,YR4,DOB4)
MPRINT(CM): DOB4 = 12*YR4 + MO4;
%MCMO5,YR5,DOB5)
MPRINT(CM): DOB5 = 12*YR5 + MO5;
RUN;
```

With new confidence from your successes in communicating with the macro processor, you remember that you can take advantage of all kinds of patterns in code and request the macro processor to generate the code. You examine the pattern in the five statements which call the macro CM in the program above and write a second macro BIRTHDTS to generate these calls. The following program, now consisting of two macros followed by a much shorter DATA step, achieves the same result as the previous version:

```
OPTIONS MPRINT;
%MCONFROUTY,CMDATE);
%generate calls to macro CM to get;
%century months for birth dates;
&CMDATE = 12*&YEAR + &MONTH;
%MEND CM;
%MCONFROUTDTS;
%generate calls to macro CM to get;
%century months for birth dates;
%DO I = 1 TO 5;
  %CM(MO&I,YR&I,DOB&I)
%MEND;
%MEND BIRTHDTS;
DATA CMBIRTHS;
SET IN.BIRTHS;
%BIRTHDTS
RUN;
```

Now you remember that you also want to convert to century months the respondents' dates of marriages, separations, and divorces which are stored in a separate SAS data set. You again hear the macro processor:

```
MP: Store your macro CM outside of your program so that other programs can use it!
```

By lifting your macro CM out of your program and storing it in a separate file, you can make it available to any program. For example, if you store your macro CM as the member CM in a partitioned data set under MVS named USTONE.SESUG93.MACLIB, it can be invoked from any program by using the FILENAME and %INCLUDE statements like this:

```
FILENAME IN 'USTONE.SESUG93.MACLIB';
%INCLUDE IN(CM);
DATA CMBIRTHS;
SET IN.BIRTHS;
%CM(MO1,YR1,DOB1)
RUN;
```

The macros CM and BIRTHDTS above are examples of building blocks that can be used in a program to reduce the amount of keyed text when you need to do the same processing more than once. With appropriate macros, the part of the program in which the macros are invoked can be written with greater simplicity.

**SUMMARY**

Encouraging the teaching of classes by programmers who work in the same environment as the intended audience can be an effective way to provide instruction, especially in advanced SAS topics. As an illustration, this paper presents portions of a tutorial designed to introduce the topic of the SAS macro facility in an intuitive way (Stone; 1993).

**REFERENCES**


**ACKNOWLEDGMENTS**

I appreciated the support of the Carolina Population Center in the preparation of this paper. Special thanks to numerous CPC friends, and to my husband Alan, who either read an earlier version or listened to a presentation of it. Thanks also to Sara Curran and other CPC graduate students who, in asking questions and even providing one of the examples, unwittingly encouraged me to attempt to write a paper on this complex topic in the first place.

SAS is a registered trademark of SAS Institute Inc., Cary, NC. © indicates USA registration.