

Why Write a Macro? Then Again, Why Not?

John W. Davison, Jr., Factotum Inc., Leesburg, VA

ABSTRACT

One of the most useful SAS ® tools for designing and programming applications is the SAS macro language. It can be used in the full range of coding situations, from a shorthand form of an expression in a single SAS statement to a container for an entire application executed by a single macro invocation. With this much power it is easy on the one hand for one to become wary and on the other hand to use it in all situations. This paper offers no hard and fast rules for macros but suggests some considerations and guidelines used by the author in deciding whether and how to write a macro or use another facility of the SAS language to create an effective application structure.

INTRODUCTION

WHAT THIS PAPER IS ABOUT

First, this paper is not a tutorial on macros, nor is it meant to be a collection of macro tips and tricks. It is intended rather to examine and compare the facilities of the SAS macro processor and to discuss some considerations for their use. Rather than offer didactic rules for such use I will suggest some questions to consider and offer the guidelines that I try to follow in my own application development. These are not very rigid and should be applied in the context of whatever stylistic rules that may be in effect in your own installation.

OTHER SOURCES OF INFORMATION

There are many well qualified authors and presenters on programming with the SAS macro language. Among the first of these is Art Carpenter, whose *Complete Guide to the SAS Macro Language* is nearly encyclopedic. Besides being a near exhaustive treatment of the macro language and its application, with a wealth of examples, its bibliography affords a comprehensive survey of the literature including SAS documentation and SUGI and other SAS User Group papers as early as 1988. Many of the earliest papers have been collected in *SAS Macro Facility Tips and Techniques, Version 6* which is no longer listed in the SAS Press catalog but copies can often be found in many established SAS installations.

OVERVIEW AND HISTORY

SYMBOLIC PROGRAMMING AND MACROS

The concept of a macro is one of the oldest in the practice of automatic machine computing. It arose very early after the introduction of symbolic assembler language, which is specific in each case to a particular machine. The notion of a macro is a further step in the process of translating program code from the logic of the problem as understood by a human to a sequence of machine operation codes for execution. This has proven useful not only in its original context but extends to high level language compilers and application systems like SAS. Moreover there is a general purpose macro language, M4 that is part of every UNIX distribution and commercial and open source packages of POSIX compliant utilities. M4, and perhaps others, can be used to write macros not only for scripting languages such as the UNIX shells, PERL, or REXX but for more general purposes as managing "boiler plate" text in the general context of document preparation.

SIMPLE CHARACTER SUBSTITUTION

The simplest form of macro processing is merely substitution of character strings for symbolic tokens embedded in the source text of an application. The token is first defined by a statement that equates its name to a character string or more extensively delimited in SAS by the "%MACRO *name* ;" and "%MEND" statements. Wherever the name is encountered thereafter, denoted in SAS by a leading "%" or "&," in the body of the code the processor replaces the instance with the defined value.

SIMPLE CHARACTER SUBSTITUTION

In any modern macro processor this facility is extended by provision for passing symbolic parameters to be replaced in the code by their own transient values so the same code can be used in different situations, and by introduction of a logic capability to facilitate alternative versions of the generated code. These facilities and the addition of functions to the macro language it is possible to write macros that are sufficiently complex to be in every sense programs at another level.

THE SAS MACRO FACILITY

The SAS macro processor is a component of Base SAS ® that is essentially automatic in its operation. As the SAS ® System transforms the code of an application into DATA and PROC steps and executes them the occurrence of "%" or "&" as the initial character of a text string invokes the macro processor to define a macro or macro variable, include text input from an external file which may be the keyboard in an interactive environment, or to substitute a previously defined macro variable or the body of a macro for an instance of its name.

MACRO VARIABLES

As stated above the simplest form of macro processing is to assign a variable name or literal string to a symbol for substitution wherever the symbol occurs. A macro variable can represent any valid string from a single character literal to a list, code expression, or control statement which may embody other macro variables that have already been defined; however, some characters such as semicolons must be qualified by macro functions so the processor can distinguish their meaning in the macro language from that in the SAS language in general. Such macro functions can be used only in the context of a macro definition.

The resolution of macros and macro variables takes place entirely during code translation and generation before it is compiled. This may seem to be an obvious point but even after all my years of experience I still stumble over this in practice. The definition of a macro variable may itself include the doublet "&&" which resolves to a single "&" which must itself be resolved in turn. The processor resolves all such definitions that it can in one pass through the code; if any remain in the resulting code the macro processor is again invoked, continuing until all have been resolved or an error condition results from the inability to do so. Thus the definition of a macro variable can itself be defined variably by use of double and triple "&" symbols to delay resolution.

INCLUDE FILES

Macros afford an effective way to organize a program into discrete steps that are defined individually and invoked in sequence in the body of the application; however, code for these steps often does not have any content that varies beyond at most simple substitutions that can perhaps be made through globally defined macro variables. In such instances the command "%INCLUDE *fileref*;" may be sufficient to incorporate an external file or even previous lines into the code. In the interactive environment and during translation the external file may even be the keyboard. In my experience this seems to work only with complete statements, at least in the case of stored files. Also, the %INCLUDE facility permits no logical or complex operations.

MACROS

A macro as defined by the delimiters "%MACRO name" and "%MEND" can be any text, subject to the notational considerations mentioned, ranging from a simple string for substitution to one that defines a complex process that amounts virtually to a program. A macro definition with the operations and functions that it contains is not executable in itself but directs execution of the macro processor, including logical decisions to direct flow of code generation. As I will show later in my examples it is in fact possible to write code that approaches or accomplishes to the entire logic of the application short of actually processing the data involved.

COMBINATIONS

The various facilities described can be combined in many ways. My own practice, among other things, is to set apart the macros of an application in individual files and as part of the initialization process have a single module, say "domacros.sas", that is a sequence of statements of the general form "%INCLUDE *macro_fileref*(*macro_filename*).". I go even further; in setting up the application I also assign the names of all the directories containing code and data to macro variables whose values suggest the contents for portability and use those in filename and LIBNAME statements.

STORED PROCESSES AND SCL

Although they do not properly fall into the heading of macros, both stored processes and SCL offer facilities that are similar in concept for the organization and control of SAS applications.

STORED PROCESSES

Stored processes have long been a part of SAS and now appear to be playing a major role in the SAS ® Enterprise Guide. A stored process is any SAS DATA step that is translated and partially compiled using the "stored" option in its DATA statement and stored for subsequent execution. In the SAS Enterprise Guide environment it appears to be the preferred if not default form of an application. I must confess that I am still learning to modify my approach to SAS development to work more fully or entirely in the SAS Enterprise Guide, only because my experience makes me so proficient in older methods and old habits die hard. I do urge those who are just learning their practice to take full advantage of the facilities offered in the new environment.

Stored processes impose some special considerations in application development. They are defined only for DATA steps (PROC steps are already compiled). They may not include global statements and macro processing is resolved during compilation of stored processes. Therefore it is necessary to find alternatives to macro parameters. Perhaps the most effective way to accomplish the purpose is to use "call symput ()" in an earlier DATA step, perhaps NULL, to define a global variable for subsequent use in a later DATA step using "symget ()" or a PROC step using "&name." Macros can, of course, be defined before a stored process is compiled and used in its definition.

SYSTEM CONTROL LANGUAGE (SCL)

SCL code can only be prepared using the facilities of SAS/AF® or SAS/FSP® and most developers in those areas probably have their methodology well established already but it is included here for completeness. Unlike a stored process a macro variable can be defined with a %LET statement or a SAS DATA step called by a SUBMIT block; however, there is little practical difference between doing so and using "call symput ()" and "symget ()." As with stored

processes macros may be defined in the preparation of SCL code but it seems more advantageous to store shared code blocks and include them during SCL program definition.

MACROS IN APPLICATION DESIGN

The SAS macro processor provides an effective tool for organizing and clarifying one's logic and easing the labor of maintaining an application. At the same time it can become a powerful tool for code obfuscation for two reasons. One is the complex notation of the complete macro language and the other the developer's need to distinguish actions and events processed during code generation from those that are part of the run time application logic.

As a control mechanism the SAS macro facility can invoke application steps for execution in sequence while the layout of the application displays only the instances if "%INCLUDE *filtered*" or "%macro_name" in outline form. Macros must of course be defined before they are invoked and the logic of the application is not affected by whether such definition occurs in a preparatory step during initialization that defines all macros at the same time or, immediately before the first reference. If the AUTOCALL option is in effect the first instance of macro call itself is sufficient to invoke the macro processor to generate the macro code; however, care must be taken that when the code of the macro is changed either explicitly or by changing the value of a global macro variable the MRECALL option does not cause the earlier generated code to be recalled in a later reference, ignoring the change. My own preference is to define all macros during initialization both for clarity and to leave little to chance, and in a production application to store compiled macros in a catalog wherever practicable.

In the course of an application conditional logic within program steps can set global macro variables to contain "%macro_name" or "%INCLUDE *fileref*" statements to be executed when their later appearance is evaluated, or ignored if the logic sets their values to blanks. Although in SAS command statements and code generation the symbols "%" and "&" must be qualified by macro functions, in a DATA step body their inclusion in literals with single quotes (apostrophes) has the effect of hiding them so they can be concatenated into the desired text string, as in:

```
data a_data_step ;
... some statements ...;
call symput ("do_this_1", cats('%', 'INCLUDE fileref') );
call symput ("do_this_2", cats('%', 'macro_name') );
call symput("do_not_do_this", cats(' ' ) );
run;
... more statements ... ;
&do_this_1 ;
&do_this_2 ;
%do_not_do_this ;
```

In complex applications conditional and iterative statements in the macro language can direct the flow of program control; however, this can be a mixed benefit. Even well designed macros are not easy to read with their profusion of "%" and "&" characters which are processed more easily by a machine than a human. Quote problems can present difficulties for both machine and human, and their logic is often much harder to follow than that of a DATA step. This is further complicated by the need to shift the viewpoint between the logic of the macro and that of the application itself. A badly designed macro can make maintenance very nearly impossible. Consequently the macro facility offers a powerful tool for both effective control structures and code obfuscation.

In my present engagement much of my work is to rewrite applications that were originally developed during the 1980 decade in an IBM mainframe environment to load quarterly banking report data into SAS tables for use in analysis and reporting systems by examiners and financial analysts. Eventually the transfers took the form of downloading data to a departmental server. In the early years disk space and other limitations required the data for each quarter to be archived as separate tape images and the local data to be stored in quarterly folders. The applications typically were designed to load one or more quarters in a given run. Over time changing conditions introduced new variables and dropped others so not all were present in all source data tables. The applications were therefore written as macros in which extensive conditional statements generated sequences of code modules that varied from quarter to quarter to be processed within the same job. This was done using macro functions iteratively to parse the date representing each quarter and use its value in conditional macro statements to select the code statements specific to the quarter being processed. The success of these design techniques led to their adoption in the development of interactive reporting systems where they have been used to set up the same basic report in response to requests for different underlying conditions. In many of these applications close examination reveals that the program logic is contained entirely in the macro processing with the actual SAS code for each quarter being little more than a highly customized DATA step or PROC SQL query.

These applications are very effective in their use of the facilities of the SAS macro language, with the essential virtue that each job is done correctly. The principal problem, apart from inconsistencies of developer style, is that on the one hand over the years macro programming techniques that were often the only way to control the transfer of data from tapes with different record structures to tables with a common structure on disk became unnecessary when the

common source changed to a DBMS based data warehouse. On the other hand facility with the macro language led developers to use it in ways that may sometimes not have been as effective as other approaches.

My presentation includes examples that are abstractions of actual code fragments drawn from these applications. I need to emphasize that it is not my intent to present any of them as "wrong" in any sense. In general they are well constructed examples of macro programming and many embody techniques that were necessary in the versions of SAS available when they were written but have since been rendered obsolete by functions available in SAS version 9. In the spirit of not fixing what is not broken they have continued in use until other considerations may require a complete redesign of the application, and some that are of general utility are still used today because they work and there is no other justification for their redesign. My purpose, as stated before, is to suggest questions that a developer might ask in deciding what approach to take when planning a project. In at least one case I will show some of the changes I intend to make when the time does come to replace one that is most widely used,

This macro was written in 1997 in SAS version 6 to create a table containing user specified columns from among the very extensive number of banking variables stored in the quarterly working tables. Parenthetically, the organization of quarterly folders on the server, reflecting the original collection of quarterly tapes, is so fundamental to the structure of nearly all such applications that to change would require a major redesign of nearly everything in common use. This example illustrates several of the questions that I suggest a developer should consider when starting a new design. The macro definition opens with:

```
%macro MAKETABLE( var      =
                    , where  =
                    , qtr    = 0
                    , table  = risview
                    , dset   =
                    , dir    = dfltdir
                    , begqtr = 0
                    , endqtr = 0
                    , begyr  = 0
                    , endyr  = 0
                    )
;

```

and provides for user specification of the list of variables to be extracted, a WHERE condition for selection, an optional number of quarters to extract, used with a beginning date, the name of the table and directory from which to extract the data, and beginning and ending quarters or years to be extracted. The macro definition begins with parameter validation in the form of several conditional macro statements of the form:

```
%IF %LENGTH(&VAR) = 0 %THEN %DO;
    %PUT ERROR: NO VARIABLES SELECTED;
    %GOTO ENDMAC;
%END;
;

```

including some with complex logic to ensure that the beginning and ending dates are provided in some consistent combination and to count the number of quarters to be processed since the size of the workspace will be kept within bounds by extracting data for one quarter at a time and appending it to the output table. Here is the first place where I would suggest an alternative approach. Rather than invest the effort in performing validation in the macro language I would write a null DATA step, primarily to improve readability but also to accept the dates involved as SAS date literals rather than a YYYY year or the YYMM format in use to identify quarters, and do comparisons and calculations using SAS date functions and formats, some of which were not available in version 6. Subsequent steps of the macro will parse the date strings to get the month and year to build conventional directory names, and even then SAS date functions could have been used to extract parts of a SAS date value. I originally intended to include the code that does this but finally gave up because I could not convey its flavor and at the same time trim it to reasonable size for presentation. All of its logic is processed in the macro language, beginning with definition and call of a macro that counts the quarters if the number is not supplied as a parameter and creates as many YYMM strings and assigns them to macro variables for use in output directory names. Apart from the fact that it is deprecated practice to define a macro within another macro, in this instance it is unnecessary because its only purpose is a macro %DO loop which can be used in the outer macro in the first place. The fact is that almost all of the code that constitutes the control structure of this macro is functionally identical to code in PERL, REXX, or VBASIC. My alternative for similar control of repeated execution of a process over an arbitrary number of quarters reduced to the following, where &begdate (not shown) and &enddate are global variables containing SAS date values for the first and last quarters, &curdate is the next quarter to be processed, &dsbase is a global variable containing the destination folder name up to the variable portion supplied by &cycle, and &process contains the name of the process to be executed repeatedly:

```
%macro getdata ;
    %do %while (%eval(&curdate) le %eval(&enddate)) ;

```

```

data _null_ ;
  curdate = input(symget('curdate'), f5.) ;
  dsbase = symget('dsbase') ;
  cycle = put(curdate, yymn4.) ;
  dsname = cats(dsbase, cycle) ;
  call symput('cycle', cycle) ;
  call symput('dsname', dsname) ;
run ;

%include source(&process) ;

data _null_ ;
  curdate = input(symget('curdate'), f5.) ;
  curdate = intnx('qtr', curdate, 1, 'e') ;
  call symput('curdate', put(curdate, f5.0)) ;
run ;
%end ;
%mend getdata ;

```

The values of &begdate, &enddate, &dsbase, and &process are set through the user interface or a larger containing application. They of course could as well, perhaps better, have been passed as parameters and used symbolically rather than called with symget(), but during my initial development I wanted maximum clarity for my own benefit.

Many of the applications discussed here require different expressions to be evaluated for the same banking analysis variable at different periods. An example of the requirement is when changes to banking regulations enable changes in the types of loan that may be offered, changing the definition of a class of risk exposure used to evaluate a bank's condition. The solution adopted has been to write the statement for each such item as a separate macro of the form:

```

%macro eval_var_n ;
  %if %eval(&cycle) %le eval("30jun97"d)
    %then (sum(of var1, var2, var3, ... ))
    %else %if %eval(&cycle) %le eval("31dec02"d)
      %then sum(of var1, var2, ... -var4)
    %else sum(of var1, var3, ... -var4) ;
%mend eval_var_n ;

```

and inserting these on the right hand side of each statement evaluating such a variable with multiple time dependent definitions.

A variation on the theme of using the same basic code in an application that takes differing detailed forms is found in an interactive system that generates reports that are similar in form but different in content in response to requests that refer to varying categories, such as bank size or region:

```

proc sql ;
  create table &output as
  select var1,
         var2,
         %if &category = cat1
           %then %do ;
             var3,
             var4
           %end ;
         %else %if &category = cat2
           %then var5 ;
  and similar conditional structures in the rest of the macro body ;

```

The point here is that the code that varies with the category does not occur simply once in the body of the macro but repeatedly throughout. It probably would have been simpler to code each variation completely, perhaps going to the extent of writing each section of constant code lying between sections of variable code as a macro, and also each of the variable sections as a macro and string them together either using the conditional statements %if ... %then ... %else ... to select each variable section as needed or to give them names systematically that each end in a macro variable containing a value keyed to the category to be selected.

CONCLUSION

The SAS macro facility provides a selection of powerful tools both for application organization and control and for program obfuscation. For their most effective use developers should become familiar with the strengths of each and consider carefully which is most appropriate to each situation that may arise. In my opinion one should also try as much as possible to keep things simple.

RECOMMENDED READING

Burlew, Michells, *SAS Macro Programming Made Easy*, SAS Press, November 1998

Carpenter, Art, *Carpenter's Complete Guide to the SAS Macro Language (Second Edition)*, SAS Press, March 2004

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

John W. Davison, Jr.

Factotum, Inc.

15265 Montresor Road

Leesburg VA 20176-5811

Work Phone: (202) 874-4586

Fax:

Email: johndavison@earthlink.net

Web:

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.