Paper 175-2008

Add SAS® Macros to Your Programming Skills: Achieve More, Write Less Code

Peter Crawford, Crawford Software Consultancy Ltd, UK

ABSTRACT

SAS® macros allow your program to do things you cannot achieve in other ways. Without needing the substantial leap to SAS/AF®, SAS macros empower the programmer's program. SAS macros will raise the functionality of your programming to a higher level. This tutorial provides a practical introduction to the concepts with examples and fun.
What you will need to bring:
1 confidence that you really want this extra level of programming
2 awareness of the non-macro solution needed for your first macro
3 determination, because there is much to learn
4 open-mindedness, because the learning will surprise you .

INTRODUCTION

Early stages in learning the SAS® Macro Language:
Introducing the first steps in learning how to program your program.
I assume you can already write SAS® programs, and you know you need to step up to the next level. The tutorial introduces the concepts in the following sections:

1. Introduction to concepts
2. avoiding macros – macro facilities available without writing %MACRO
3. simple macros (are not necessarily small)
4. what macro quoting is for
5. dirty macros and clean macros (why – best behaviour)
6. beginning to understand macro environments
7. write less code
8. feedback

This tutorial introduces the main concepts you'll need to understand to use the SAS Macro Language.
It should not take the place of proper SAS Macro Language training courses.

INTRODUCTION TO CONCEPTS – 1

SAS macros are about processing syntax. We want this when plain syntax does not seem to do enough for us.

DEFINITION

A SAS macro is a collection of SAS program statements which compile into an object that can be invoked by its name, like

```
%print_empty( data= my.data_set, empty_msg= No data to report  this time )
```

DETAILS

That is the definition I use here.
However, there is much more to base SAS macros.
A SAS macro can include syntax that is not part of the SAS Macro Language. That will be clarified by examples in section 3 of this Tutorial. Some features of the SAS Macro Language can be used without defining a macro. That will be demonstrated in the next section.
Here is the definition of a very simple macro. Its purpose is to produce a message when The PRINT procedure produces no report because the data set to be printed is empty.

```
%MACRO print_empty( data=, empty_msg=, out= conts );
 %IF %anyobs( &data ) %THEN %DO;
    PROC PRINT DATA= &data;
    RUN;
 %END;
 %ELSE %DO;
   DATA _NULL_ ;
     FILE PRINT ;
     PUT / "&empty_msg" /;
```

```
      RUN;
    %END ;
   %MEND  print_empty ;
```

The macro `%print_empty()` performs a PROC PRINT if the data set is not empty, otherwise it produces a message. Note that this macro invokes macro %anyobs to indicate that the data set has observations.

DEFINITION

The SAS® Macro Language provides functions, statements and options to support SAS® System programming languages.

DETAILS

There are many features in the Macro Language. Although this tutorial doesn't describe them all, you will see the main features and their purpose, to help you advance with confidence and learn more.

AVOIDING MACROS – 2

Some facilities of the SAS Macro Language are available without writing %MACRO.
Passing a parameter through a program is supported like :

```
   %LET period_ending = 31-Dec-2007 ;
```

Throughout the program, where ever the reporting period end date is needed, it is available as this macro variable, as in these two examples :

```
   TITLE2 "Reporting at &period_ending"  ;
   WHERE transaction_date LE "&period_ending"d ;
```

The Macro Language statement `%LET` made the macro variable named "period_ending". It gave it a value that looks like a date constant. It was used in a TITLE statement and a WHERE statement.

A very useful and very simple base SAS language facility is SUBSTR(). The Macro Language equivalent is almost identical :

```
   TITLE2 "reporting at %SUBSTR( &period_ending, 4 )"  ;
```

puts just the month and year into the title.

Two more functions are %**EVAL**() and %SYSFUNC.
The first of these %**EVAL**(), performs integer arithmetic, including logic tests;

```
   %LET reports_left = %EVAL( &reports_left - 1 ) ;
   %LET  finished    = %EVAL( &reports_left < 1 ) ;
```

The **%SYSFUNC**() function is a really useful feature because it allows the Macro Language to invoke (most) base SAS functions. For example :

```
   %LET run_started = %SYSFUNC( DATETIME(), DATETIME ) ;
```

places a time stamp in macro variable &RUN_STARTED.

```
   %LET week_begins = %SYSFUNC( INTNX( WEEK.2, "&SYSDATE9"D, 0 ), DATE9 ) ;
```

places Monday's date of this week, into the macro variable &WEEK_BEGINS.

```
   TITLE2 "reporting  %SYSFUNC( PUTN( "&week_begins"d, WEEKDATE29. ))" ;
```

places the macro variable's date into the title line in the expanded WEEKDATE. format.

ONLY WITHIN %MACRO

Two common Macro Language statements which only work within %MACRO are %IF blocks and %DO blocks.
In many situations the SAS® Macro Language achieves all the flexibility needed, without writing a %MACRO.

SIMPLE MACROS (ARE NOT NECESSARILY SMALL) – 3

By simple macro I mean one with simple logic.
A simple macro has a %MACRO statement at the beginning and a %MEND at the end. When invoked, the simple macro submits the text that appeared between the %MACRO and %MEND.
However, you are here to discover more than this ~ but remember the fundamentals.
Also in the category of "simple macros that are not small" are those where the language allows the macro to invoke a large package of SAS program using the values of parameters. This example demonstrates :

```
   %MACRO long_program( infile=, out=, dest= )
                       /DES= 'load and report' ;
```

```
    DATA inpt( LABEL= ' descriptive label about the data' COMPRESS= YES
                KEEP= a long list of many variables . . . . .;
      LENGTH a $2 long $50 list $2000 of many variables 8 ;
      INFILE &infile  DSD TRUNCOVER         ;
      INFORMAT many latest YYMMDD14. ;
      INPUT  . . . . . . . . . . . . . .  ;
      IF various - conditions - are - met THEN DO;
             FILE &dest ;
             PUT "messages about &infile on row " _N_ ;
      END;
    RUN;
    PROC SORT DATA= inpt OUT= &out ;
      BY some sort keys ;
    RUN;
    TITLE "long program report summary" ;
    PROC MEANS ;
    RUN ;
   %MEND;
```

The process is invoked by simple syntax

```
   %long_program( infile= indd11, out= keep.basic, dest= rept02 )
```

It is a simple macro because it contains the minimum amount of Macro Language = %MACRO and %MEND.

SHORTEST

The shortest macro (imho) introduces some sophisticated concepts. Because it is brief I add it here

```
   %MACRO now( fmt= DATETIME23.3 )/ DES= 'timestamp' ;
   %SYSFUNC( DATETIME(), &fmt )
   %MEND  now ;
```

It looks brief but it contains enough "lesson" to fill a Coders Corner paper(1) The main point here is that when it is invoked, `%NOW()` uses the SAS Macro Language to derive new information (current date and time) and pass the results to the submit buffer.

What makes a macro complex is the amount of Macro Language logic that it contains. The Macro Language can process many kinds of information. However, generally it exists to submit code.

WHAT MACRO QUOTING IS FOR – 4

Since the Macro Language only processes text, it doesn't quote text in the way that text constants are quoted in the base SAS programming language. The language allows this fragment of a macro

```
      %IF &state EQ TX %then %DO ;
                                  something big
                             %END ;
```

When you think about this, consider how Nebraska and Oregon abbreviations might be interpreted. After the macro variable &STATE is replaced with NE, the macro compiler will see

```
      %IF NE EQ TX %THEN %DO ;
```

This is a problem for the macro compiler so it will probably not do what you want.

This is not the only way in which the values in a macro collide with the Macro Language. We know now that the SAS Macro Language is for "programming" our code, at some future time you will discover this collision problem, and how much it needs to be solved.

The solution in the SAS Macro Language is **macro quoting.** I leave that topic for a more advanced stage of learning, but you must be aware of the issue. The simplest solution here is not macro quoting, but base SAS quoting - looking like

```
      %IF "&state " EQ "TX " %THEN %DO ;
                                        something more
                                   %END ;
```

It will work adequately, as long as the values in the macro variable &STATE can be relied upon.

The Macro Language provides functions to control the level of protection and the timing of protection release.

The most robust protection can be used when you are completely unable to control the values in the macro variables and must program "defensively". You need this, for example, when programming for parameters coming from a web input form. Then, remember to look up the documentation for the `%SUPERQ()` function. Here, the fragment of the macro would look like

```
        %IF %SUPERQ(state) EQ TX %THEN %DO ;
                                           something more
                                      %END ;
```

DIRTY MACROS AND CLEAN MACROS (WHY – BEST BEHAVIOUR) – 5

This has little to do with soap and water.
A clean macro will leave no more trace of its presence than the text it submits. That means when the macro completes and closes. It will leave behind no unwanted extras :

- no more macro variables
- no extra libnames assigned
- no extra datasets left in the work library
- no permanent changes to the environment

except of course, when this is the purpose of the macro.

A "dirty" macro leaves these traces for others to "clean up".

A macro might be invoked many times in a SAS session. When this "clean-up" issue is ignored, like the straw and a camel's back, any of these "extras" from a "dirty" macro might bring down your system.
So, make sure your macro executes on its best behaviour.

The Macro Language can capture option settings with the base SAS function GETOPTION(). Of course that is available through %SYSFUNC().
Where a macro assigns files and libraries, it can keep track, and release the assignments when no longer needed.
In a similar way, any data set created can be added to a "hit list" to be deleted before the macro finishes.
A useful technique is to build the list as the macro progresses.
Another, is to adopt the OUT= _DATA_ convention. This is very similar to a DATA step statement DATA with no output dataset name. _DATA_ uses the WORK.DATAn convention. After each step creating a data set named in this DATAn convention, use code like:

```
            %LET latest= &SYSLAST;
            %LET hitlist = &hitlist &SYSLAST ;
```

Then your code can refer to that data set by name &latest or any particular reference name you need, and the hitlist can easily remove it later.

Of course the first issue in writing macros is to have a %LOCAL statement near the beginning. There you place the names of variables you will only use in your macro. As far as possible, do this.
There will be times when it isn't possible, but make your best effort Your macro wil be on its "best behaviour" and the users of your macro will be recommending you as a source of quality.

BEGINNING TO UNDERSTAND MACRO ENVIRONMENTS – 6

What happens inside a macro need not have any impact once the macro is complete. However, you may need many items inside your macro. Some of these macro variables will hold parameters. Others will hold counters for a loop. Some might capture the values of system options before you temporarily change the option value. These macro variables are referred to as Local macro variables. This is contrasted with Global macro variables.
The local environment is convenient. All local macro variables will be deleted when the macro's environment closes. Unfortunately, there is no easy way to remove macros and formats from the environment after they are compiled.

Although environments are considered Local or Global, this does not mean 2 environments. There may be many environments. There will be one from each nesting of macros. Although we cannot write to an explicit environmant, other than the Global and the Local, when a macro executes, anywhere in the hierarchy of calls, assigning a value to a macro variable, works like

- if the macro variable is a Local name, then use Local environment
- if name already exists in the current mix of inherited environments, use the name that exists in whichever environment it exists
- if name does not already exist, and a local environment exists, add the name to the Local environment
- if name does not already exist and no local environment exists, add the name to the calling environment.

This complexity becomes helpful when your solution needs to invoke a macro to return a value, that you want delivered to your own environment – say result_string; here is sample code that uses this approach

```
   %MACRO demo2( parm1, parm2 )/ DES='use vars  neither local not global' ;
     %* inherit the environment of mVar &result_string or it will be local;
        %LET result_string = none ;
        %collect_results( a,b,mean ) ;
         %IF result_string EQ none %THEN %DO;
                  %* handle no results situation ;
         %END;
         %ELSE %DO;
                  %* use results information in &result_string  ;
         %END;
       %* we now lose that result_string if it was not inherited as this closes;
       %*    or it is already updated in the environment that called this;
   %MEND  demo2 ;
```

And the macro invoked to collect the result_string, might look like

```
   %MACRO collect_results( lib, mem, stat ) /DES= 'get requested stats' ;
      %LOCAL old_opt work_hitlist ;
      %LET   old_opt= %SYSFUNC( GETOPTION( _LAST_, KEYWORD ));
     %* now collect &stat from all numerics in &lib..&mem ;
      PROC MEANS NOPRINT DATA= &lib..&mem ;
      OUTPUT OUT= _DATA_ &stat=  ;
      RUN;
     %LET work_hitlist= &work_hitlist %SCAN(&SYSLAST,2,.) ;
      PROC TRANSPOSE ;
      RUN;
     %LET work_hitlist= &work_hitlist %SCAN(&SYSLAST,2,.) ;
      PROC SQL NOPRINT ;
     %* now transfer all stats into a macro variable
        prefixed by the name of the numeric variable, and : ;
         SELECT compbl( _name_ !!':' !! put( col1, best8.) )
           INTO :result_string SEPARATED BY ', '
           FROM &SYSLAST
          WHERE _name_ NOT IN( '_FREQ_', '_TYPE_' )
                ;
      %LET result_string = (&SQLOBS) &stat.s &result_string ;
      QUIT;
      %PUT _USER_;
     %* clean out hitlist from work library ;
      PROC DATASETS MT= DATA NOLIST ;
         DELETE &work_hitList ;
         RUN;
      QUIT;
     %* reinstate old options ;
      OPTION &old_opt ;
   %MEND  collect_results ;
```

PARAMETERS ARE NOT GLOBAL

It is very convenient that named parameters in a macro are "local". They provide a convenient way to establish locals while giving them a default value. Add that to the flexibility the parameters provide.

WRITE LESS CODE – 7

The general idea with macros starts as a convenient way to repeat code ~ but we have %include anyway.

OK, so a macro's parameters allow each execution to be that little be different ~ but without %MACRO we still have macro variables that can vary for each call %include.

What %MACRO will allow you to achieve, is the logic at run time.
That is %IF tests which decide what code to submit, compile and run. Not available outside %MACRO are iterative %DO blocks. Imagine that you have a list of tasks to execute. Each task is well described by its name, and probably the rundate. A macro can work on the list of names.

```
%LET task_list = this THAT the other ;


%DO task_n =1 %TO 10 ;
   %LET task_name = %SCAN( &task_list, &task_n ) ;
```

Any additional parameters will probably be able to be derived from the task-name.
The relevant program suite to be executed will be able to be identified by its name and the relevant start-up command can just as easily be made available

CONCLUSION

Macros become a natural way once you have mastered these concepts.
Use your skills with DATA step processing to organise your programs in logic conditions and loops. Macros will carry around the information your processes need, and just as quietly, they can clean up and leave behind only the traces you want to leave..

REFERENCES

(1) Crawford, P. "The Big Introduction from the Smallest Macro". Paper presented at the SAS Users Group International Conference. San Francisco, CA. March, 2006

ADDITIONAL READING

Read any introductory books and guides to SAS macros. The internet will provide a large list. Use both SAS On-line Doc and the book form of SAS 9.1 Macro Language: Reference which can be found in Product Documentation at http://support.sas.com.

CONTACT INFORMATION

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

Peter Crawford
Crawford Software Consultancy Ltd
31 Sefton Road, Croydon, CR0 7HS, UK
+44(0)7802732254
Peter.Crawford@blueyonder.co.uk