ABSTRACT

SAS® programming frequently includes very repetitive sets of code in the form of a few lines of code within a DATA step, a single repeating DATA/PROC step or a repeating combination of DATA and PROC steps. This paper will present techniques for performing repetitive aspects of SAS programming using SAS macro programming techniques.

The discussion will start with a review of some basics of SAS macro, including macro "arrays", and build to more complex examples. It will also cover data and logic driven techniques.

These techniques dynamically generate SAS code and, in some cases, reduce the amount of code that needs to be explicitly included in a program.

The paper includes samples of code that can easily be applied to a wide variety of applications.

GENERAL MACRO RULES

- Macro variable values are always character data. Even numbers are characters. Arithmetic operations can be done, but only with special functions.
- After a macro variable is created (as will be described below) it is reference by prefixing it with an ampersand (i.e. &VAR1, &NAME)
- A macro procedure (%MACRO macro-name - %MEND) must be defined before the procedure can be used. This is frequently confusing since it distorts the sequential flow of the code. It's a good practice to use comments with "Step #" to document the sequence of steps in programs.
- Once a macro is defined, it is referenced by prefixing it with a percent sign (i.e. %A, %GET_NAME). Note that a macro execution statement (%GET_NAME) does not require a semi-colon after it; but it's a very good practice to use a semi-colon anyway. There are times when the missing semi-colon will cause problems.

DATASET VARIABLES vs MACRO VARIABLES

Let's assume that a dataset has 3 rows/observations and 3 columns/variables, for a total of 9 cells. To use all the data from this dataset as macro variables, it will take 9 macro variables to carry these 9 cells.

This is a key different between macro variables and dataset variable. A macro variable contains a single value of data, not a 'column' of data.

Macro programming gets around this difference by using a concept of an "implied array" that will be discussed later in this paper.

CALL SYMPUT COMMAND

The CALL SYMPUT command takes a single dataset value and assigns it to a macro variable.

The power of this command is that data that previously was available only to the single DATA or PROC step, now becomes available to all subsequent DATA and PROC steps. It also becomes available to macro programming outside of DATA and PROC steps. The full usefulness of the created macro variable will depend on the "scope" of the macro variable (see %GLOBAL and %LOCAL in the SAS OnlineDoc.)

SYNTAX:
CALL SYMPUT(argument-1,argument-2);

Argument-1 specifies a character expression that identifies a macro variable. If the macro variable does not exist, the routine creates it.

Argument-2 specifies a character expression that contains the value.

EXAMPLES:
CALL SYMPUT('MVAR',A);
Moves the value 'A' into macro variable &MVAR. Notice that the macro variable name is in quotes. The value isn't since it is always character.

CALL SYMPUT('VARNUM',PUT(_N_,BEST.));
Takes the current record number of the input dataset (_n_), converts it to a character and moves it into &VARNUM.

CALL SYMPUT('VAR'||TRIM(LEFT(PUT(_N_,BEST.))),TRIM(LEFT(NAME)));
The value of the NAME character variable is shifted to the left and the spaces on the right are deleted. This value is then assigned to VAR# where # is the current record number of the input dataset. The "||" is a "concatenation" symbol. In this case it means to append the number to 'VAR'. The result would be VAR1, VAR2, etc. and macro variables &VAR1, &VAR2, etc. would be created.

There is also a SYMGET function which returns the value of a macro variable during the DATA step execution. Note that this function does not include the word "CALL" in its syntax. Most of the time simply putting the macro variable in the code will suffice:
STNAME="&STATENM"; /* character field */
STCD=&STATECD; /* numeric field */

%DO-%END CONSTRUCT

Obviously in SAS, the default flow in a DATA step is - read one record, process the logic, go back to the top of the
DATA step and read another record, process the logic, etc. You don't even have to code it if you don't want to.

This is a loop.

In the DATA step, you can use the DO-END construct to loop through data. In the SAS Macro Language, you can use the %DO-%END construct to build SAS code.

**SYNTAX:**

```
%DO index=start %TO stop <%BY increment >;
    . . .  more statements . . .
%END
```

Index is a valid macro variable name. The names "I" for 'index' or "N" for 'number' are frequently used as the Index name. If the variable does not exist before the %DO statement, the Macro Pre-processor creates it in the Local Symbol Table.

Every time Index is used in a %DO construct it will be reinitialize. So it's a good practice to not use index variable names anywhere else in the program. Many programmers use single-letter variable names to distinguish index variables from other variable types and to simplify typing in the more complex statements.

Index changes each time the loop executes based on the Start, Stop and Increment values.

Start and Stop specify integers that control the number of times the portion of the macro between the %DO and %END statements is processed. These integers can be the result of expressions.

Increment specifies an integer that is added to Index before each iteration of the loop. By default, Increment is 1. Increment can be any positive or negative integer. If Increment is negative, Stop must be less than Start. Increment cannot be manipulated by logic inside the loop.

The first time the %DO statement executes, Index is equal to Start. As processing continues, Index changes by the value of Increment until Index is outside the range of integers defined by Start and Stop.

Index can be changed during processing. For example, based on conditional logic you might want to force the end of a loop before Stop is reached. This can be done by setting Index higher than Stop when a certain condition is met; this will then force the loop to end.

**MACRO "ARRAYS"**

These are not explicit arrays like those that are defined by the DATA step Array statement. Instead, Macro "arrays" are implied. This is a programming technique and a way of looking at the data.

If you have macro variables &VAR1, &VAR2 and &VAR3, then you can use a %DO-%END construct to perform some logic on all three variables. These 3 variables are not an array, but you can treat them as if they are.

Let's try a simple program. Use %LET to assign a value to a macro variable outside a DATA step. In fact, no DATA or PROC step is used in this program. It's a macro program. Notice that %A has a %MACRO and a %MEND statement. The "A" is a comment on the %MEND statement and it is not required. But it's good technique to put the name of the macro on the %MEND because when you have lot of macros in a program it will be easier to match the %MEND statement to the corresponding %MACRO statement.

```
%LET VAR1=A;
%LET VAR2=B;
%LET VAR3=C;
%MACRO A;
    %DO I=1 %TO 3;
        %PUT &&VAR&I;
    %END
%MEND A;
```

%PUT is similar to the DATA step PUT statement. It's also a global statement like a LIBNAME or OPTIONS statement. So it's generally used outside a DATA step.

But, you might ask, what is "&VAR&I"?

**DYNAMIC SUBSTITUTION**

It is important to understand that before a SAS program executes, the SAS Macro Pre-Processor makes several passes at the SAS program being executed. The Pre-Processor reads through the whole program, one or more times, converting macro code into text strings. Then it processes the global statements, DATA steps and PROC steps, as you would expect in any SAS program which did not have macro code in it.

Fundamentally, the only thing that SAS macro programming does is create more SAS code.

So what is "&&VAR"?

On the first pass, the Macro Pre-Processor converts && to & (another macro rule) and &I to 1, so the text becomes &VAR1. Does this look a little more familiar?

On the second pass, the Macro Pre-Processor substitutes the character value for the name of the macro variable. So %PUT &VAR&I becomes %PUT A. This is called "dynamic substitution".

Essentially the Macro Pre-Processor rewrites the program as:

```
%PUT A
%PUT B
%PUT C
```

The results in the SAS Log are:

A
B
C

Since all the code in this program was macro code, the whole program is processed in the Pre-Processor and there's nothing for the main SAS Processor to do.
Below we'll discuss how to debug macro code. When you use the SYMBOLGEN option, SAS will display in the log how each macro variable becomes a text string in the program. Take the time to use SYMBOLGEN and read each line in the log to see the substitution process.

**CALL EXECUTE (ARGUMENT)**

To see the value of the CALL EXECUTE statement consider this simple program:

```sas
%MACRO A;
  %PUT MACRO A;
  DATA A; /* #2 */
    PUT "WITHIN A";
  RUN; /* #2 */
%MEND;

DATA _NULL_; /* #1 */
  PUT "BEFORE A";
  %A;
  PUT "AFTER A";
RUN; /* #1 */
```

The results in the SAS Log are:

**MACRO A**

**BEFORE A**

NOTE: DATA statement used:  
   real time 0.00 seconds

**WITHIN A**

NOTE: The data set WORK.A has 1 observations and 0 variables.  
NOTE: DATA statement used:  
   real time 0.37 seconds

NOTE: SCL source line.  
   PUT "AFTER A";  
   ---  
   180

ERROR 180-322: Statement is not valid  
or it is used out of proper order.  
   RUN; /* #2 */

From within a DATA step, SAS allows you to execute a macro which contains part of a statement, whole statements, or multiple statements. But step boundaries are not permitted. When you have step boundaries in macros, they are applied to the last incomplete DATA or PROC step. It's similar to matching DO statements with END statements.

In this example, the code essentially becomes,

```sas
DATA . . . #1  
RUN; #1  
DATA . . . #2  
RUN; #2 =⇒ ends DATA #1  
RUN; #1
```

**DEBUGGING**

The techniques presented in this paper can be very difficult to debug at any experience level.

I highly recommend using this statement at the top of your code:

```sas
OPTIONS SYMBOLGEN MPRINT MLOGIC;
```

SYMBOLGEN (NOSYMBOLGEN) tells SAS to display the results of resolving macro variable references. Generally the line in the log will have the format:

```
SYMBOLGEN: Macro variable _ resolves to _
```

MPRINT (NOMPRINT) tells SAS to print the text that was sent to the SAS compiler as a result of macro execution. These lines in the log will be prefixed with MPRINT( ) where the name of the macro will be included in the parenthesis.

MLOGIC (NOMLOGIC) tells SAS to show messages related to initialization information, results of arithmetic and logical macro operations, and termination information. These lines in the log will be prefixed with MLOGIC( ) where the name of the macro will be included in the parenthesis.
The statement above generates a lot of detail in the SAS Log that shows how SAS is manipulating your code. Once you are satisfied with the code, comment out this line to reduce the size of the log. But don't delete it, you're bound to need it again some day.

STRATEGY
SAS Macro programmers always pass on this advice:

- Start small and simple. If you are going to eventually be processing a %DO-%END statement with 100 iterations, start with just 2 or 3 iterations. It saves time while you are still debugging.
- If at all possible take the DATA and/or PROC steps out of the macro. Write these steps separately until they are correct. And develop the macro logic separately using a %PUT statement to represent these steps, until the flow of the program is correct.

EXAMPLES

EXAMPLE 1
This code does a loop inside a loop:

```sas
DATA _NULL_
  DO I=1 TO 2;
    DO J=44 TO 45;
      PUT I= J=;
    END; /* END J LOOP */
  END; /* END I LOOP */
RUN;
```

Results in log:

```
I=1 J=44
I=1 J=45
I=2 J=44
I=2 J=45
```

You can do a lot of logical work with do loops in DATA steps, but so far the code has not generated any new SAS code, so it has not saved you any work. In addition, this example is not driven by the data.

NOTE: Be sure that you understand the pattern of the I and J variables in the loops before you proceed.

EXAMPLE 2
Now 2 macros are executed from the DATA step.

```sas
%MACRO A;
  PUT 'MACRO A: ' I= J= A= B=;
%MEND; /* END A */

%MACRO B;
  DATA _NULL_
    PUT 'MACRO B: ' A= B=;
  RUN;
%MEND B;

DATA _NULL_
  DO I=1 TO 2;
    DO J=44 TO 45;
      PUT I= J=;
      %A
      A=I;
      B=J;
      CALL EXECUTE('%B');
    END; /* END J LOOP */
  END; /* END I LOOP */
RUN;
```

Results in log:

```
I=1 J=44
I=1 J=45
MACRO A: I=1 J=45 A=1 B=44
I=2 J=44
MACRO A: I=2 J=44 A=1 B=45
I=2 J=45
MACRO A: I=2 J=45 A=2 B=44
MACRO B: A=. B=.
MACRO B: A=. B=.
MACRO B: A=. B=.
MACRO B: A=. B=.
```

There's almost too much to cover in this one step but the contrast of techniques is good to see all in one place.

First of all, by SAS macro rules, %A and %B must be defined before they are used in the DATA step. Note also that the %MEND statements are documented in two different ways.

%A is a simple PUT statement - a single line of a DATA step.

%B is a complete DATA step. This could be a mix of multiple DATA and PROC steps.

In this example, both macros, %A and %B, are executed in the one DATA step. And the "CALL EXECUTE ("%B");" statement execute for each iteration of I and J. But there's a big difference in the results.

Notice that the PUT statement with I and J, and the %A output, are displayed first in the results. These statements are within the current DATA step and they are executed at the step boundary.

The CALL EXECUTE statement adds code to the program and the new code executes after the current (first) DATA step as subsequent steps. So the output from these steps is displayed later in the log.

Notice also that the variables A and B are missing in the %B output. Remember another basic principle of SAS is that DATA step variables cannot be passed from step to step. Since the program is, as a result of the CALL EXECUTE statement and %B, now executing what could be called DATA steps 2 through 5, variables A and B are no longer available. If you use the OPTIONS statement which was described previously, this will become more obvious, but the output is too much for this paper.

EXAMPLE 3
Now, we'll pass some values to the new steps generated by %B:

```sas
%MACRO A;
  A=I;
%MEND A;
```
PUT 'MACRO A: ' I= J= A= B=;
%MEND; /* END A */

%MACRO B;
DATA _NULL_;%PUT MACRO B: C=&C D=&D;
E=&C;
F=&D;
PUT E= F=;
RUN;%MEND B;

DATA _NULL_; DO I=1 TO 2; DO J=44 TO 45;
PUT I= J=; %A
A=I;
B=J;
CALL SYMPUT('C',I);
CALL SYMPUT('D', TRIM(LEFT(PUT(J,BEST.))));
CALL EXECUTE('%B'); END; /* END J LOOP */
END; /* END I LOOP */
RUN;

Results in log:
I=1 J=44
MACRO B: C= 1 D=44
I=1 J=45
MACRO A: I=1 J=45 A=1 B=44
MACRO B: C= 1 D=45
I=2 J=44
MACRO A: I=2 J=44 A=1 B=45
MACRO B: C= 2 D=44
I=2 J=45
MACRO A: I=2 J=45 A=2 B=44
MACRO B: C= 2 D=45
E=1 F=44
E=1 F=45
E=2 F=44
E=2 F=45

Notice how %B has changed so all the variables could be displayed. Now %PUT from the %B executes when %B executes to build the new DATA step, not when the new DATA step is executed, so the "MACRO B" output line is seeming within the original DATA step. In other words, %PUT, and any other macro commands execute when the macro executes, not when a DATA or PROC step execute.

Notice also the affect of the TRIM and LEFT functions on the value of &D compared to &C. The TRIM and LEFT functions are very useful in macro coding since they manipulate text strings.

EXAMPLE 4
But sometimes we want to drive a process using input from a file - "data driven".

In this example, the program will build a data dictionary listing a series of data warehouses, the files in them, the fields in the files and all the characteristics of the fields.

The input is a simple file that contains 2 variables. PATH is a server path which is used on a LIBNAME statement, and DWH_NAME is the name of a data warehouse which is associated with each PATH.

This program reads the file, assigns the LIBNAME, does a PROC CONTENTS to get the details of the fields, and merges the new details to an existing file. Note that the base file, COLS, is initialize in a step which is not shown.

%MACRO PROCCONT;
LIBNAME DIRPATH "&PATH";
PROC CONTENTS DATA=DIRPATH._ALL_
  OUT=COLS1 NOPRINT POSITION;
RUN;

DATA COLS2 (KEEP=DWH_NAME MEMNAME NAME TYPE LENGTH VARNUM LABEL FORMAT);
LENGTH DWH_NAME $15. MEMNAME $20. TYPE LENGTH VARNUM 8. LABEL $50.
FORMAT $8.;
SET COLS1;
DWH_NAME="&DWH_NAME";
RUN;

PROC APPEND BASE=COLS DATA=COLS2 FORCE;
RUN;%MEND PROCCONT;

DATA _NULL_; SET DATADICT.PATHS;
CALL SYMPUT('DWH_NAME',TRIM(DWH_NAME));
CALL SYMPUT('PATH',TRIM(PATH));
CALL EXECUTE('%PROCCONT');
RUN;

Notice again that the macro is defined before it is called. In the main DATA step the input file is read and the loop of reading the records drives the multiple iterations of the macro. And in this case the macro contains more than one DATA and/or PROC steps.

EXAMPLE 5
This simple example uses the macro keyword parameter instead of CALL SYMPUT and shows the potential for the CALL EXECUTE statement:

DATA A;
PARTY="DEM"; VOTES=50; OUTPUT;
PARTY="REP"; VOTES=70; OUTPUT;
PARTY="IND"; VOTES=90; OUTPUT;
RUN;

%MACRO DEM(VOT=);
DATA _NULL_; PUT "DEMOCRATIC VOTES=&VOT";
RUN;%MEND DEM;

%MACRO REP(VOT=);
DATA _NULL_; PUT "REPUBLICAN VOTES=&VOT";
RUN;%MEND REP;
%MACRO IND (VOT=);
DATA _NULL_;  
PUT "INDEPENDENT VOTES=&VOT";
RUN;
%MEND IND;

DATA _NULL_;  
SET A;  
SELECT (PARTY);  
WHEN("DEM") DO;  
CALL EXECUTE('%DEM(VOT='||TRIM(PUT(VOTES,BEST.))||')');  
END;  
WHEN("REP") DO;  
CALL EXECUTE('%REP(VOT='||TRIM(PUT(VOTES,BEST.))||')');  
END;  
WHEN("IND") DO;  
CALL EXECUTE('%IND(VOT='||TRIM(PUT(VOTES,BEST.))||')');  
END;  
END;  
RUN;

Results in log:
DEMOCRATIC VOTES=50
REPUBLICAN VOTES=70
INDEPENDENT VOTES=90

The %DEM(VOT='||TRIM(PUT(VOTES,BEST.))||') portion
of the code resolves to %DEM(VOT=50). So it passes the
DATA step value of VOTES, through the macro variable
VOT to the macro %DEM. "VOT" is called a keyword
parameter.

Also notice that the macro has the keyword defined on its
%MACRO statement. Inside %DEM, "VOT" is used just
like any other macro variable (%VOT).

Not too impressive - until you realize that you can read in
a file and drive a wide variety of macros, with any
combination of DATA and PROC steps, based on the
input data.

I have an application where the user selects a group of
files to be imported into a SAS application. These files
have a variety of input formats in .csv, .txt and .xls
formats. Each input file has a File Format Code
associated with it. In a DATA step, like the one above, I
read through each file selected by the user and, based on
its File Format Code, I execute a macro. So each File
Format Code has a unique macro. There are only 3 File
Format Codes, but 30 files. Also each macro requires a
different combination of keyword parameters. The result
is that all the files are converted into one format. The user
can select any combination of files, and the data will be
converted into one format. That's dynamic and data

EXAMPLE 6
This example takes a different approach. In the DATA
step, records are selected from a file and 3 variables are
kept. This step creates an "array" of 3 columns by "CNT"
rows.

%LOOP then loops through the array much like a DATA
step would loop through a file.

DATA _NULL_;  
RETAIN COUNT 1;  
SET LIB1.FILE1(WHERE=(KEY="AA"))  
END = LAST;  
COUNT+1;
CALL SYMPUT('VARA'||TRIM(LEFT(PUT(COUNT,BEST.))),TRIM(LEFT(VARA)));  
CALL SYMPUT('VARB'||TRIM(LEFT(PUT(COUNT,BEST.))),TRIM(LEFT(VARB)));  
CALL SYMPUT('VARC'||TRIM(LEFT(PUT(COUNT,BEST.))),TRIM(LEFT(VARC)));  
IF LAST THEN CALL SYMPUT('CNT',PUT(COUNT,BEST.));  
RUN;

%PUT __ALL__;

%MACRO LOOP;  
%DO I = 1 %TO &CNT;  
%PUT &&VARA&I &&VARB&I &&VARC&I;  
%END;  
%MEND LOOP;

%LOOP;

The DATA step creates an implied array:
VARA1   VARB1   VARC1
VARA2   VARB2   VARC2
VARA3   VARB3   VARC3
VARA4   VARB4   VARC4
Etc.

The %PUT __ALL__ or %PUT __USER__ statements will
display all the macro variables in the SAS Log but they will
not as be nicely organized as the %PUT generated.

This example simply prints the values of the macro
variables using the %PUT statement. But the %PUT can
be replaced with any amount of code, including additonal
macros, which could process the "array" to meet your
needs. Note that CALL EXECUTE cannot be used within
%LOOP since it must be used inside a DATA step.
CONCLUSION
The SAS language, through the use of macro programming, provides an extensive set of tools to make developing repetitive sets of code reasonable easy to do.

One of the benefits to using SAS Macro programming is to write code that is dependent on the values in the input data. The examples in this paper were simple due to space. However, if the code contained logical and selection statements such as %IF-%ELSE, IF-ELSE and BY statements, and WHERE clauses, based on the input data, the code could have been more dynamic and responded more specifically to various input values.

Another benefit is that Macro programming can generate a volume of coding. If a program has about 20 DATA and PROC statements, and if the input file had 50 records, the macro would generate 1000 lines of SAS code to be executed. Maybe that’s still not too much to be entered manually. But even with just 20 DATA and PROC statements, if there was a larger input file, it could potentially cause the macro program to generate as much as 5,000-10,000 lines of code. I doubt anyone would want to sit through that.

However, macro programming is not appropriate for every SAS program written. In each situation the development and testing time for adding macro code has to be weighed against the benefits.

REFERENCES: