Paper 045-2008

SAS® MACRO: Beyond the Basics

Stephen Philp, Pelican Programming, Redondo Beach, CA

ABSTRACT

The SAS® macro facility has been an essential part of the SAS programmer's toolkit for many years. Its continued use is a testament to its relevance and power. With each new release of SAS® software, macros continue to be arguably the easiest way to create data-driven code generation. The intended audience for this paper includes any SAS programmer who writes macros or works with macros that are written by others.

The following topics will be discussed:

- Review of macro timing
- the %SYSFUNC macro function (avoiding the DATA step)
- macro expressions
- global and local symbol tables
- debugging
- creating and using macro libraries

INTRODUCTION

At its heart, SAS macro programming is nothing more than conditional text replacement. If some condition is true then write this code. Otherwise write this code. That really is it in a nutshell. Understanding this fundamental concept helps prepare you for more advanced macro topics since everything builds upon it.
When we talk about advanced macro topics, the things we focus on are the details of what's happening and most importantly when it's happening.

MACRO TIMING

Before we get into advanced macro topics, we should first review what happens when SAS encounters macro in code. When SAS code is submitted, it needs to be compiled into machine readable code. Compiling involves several steps working together. SAS macro simply adds an additional step to the compile process. Below is just a review of the process and not an in-depth discussion. Some details are simplified for clarity.

After SAS code is submitted, it is first placed onto the input stack. The input stack is scanned and parsed for tokens (words) by the word scanner. If a token contains a macro character (a percent sign or an ampersand) that token is sent to the macro compiler. The macro compiler does its work and places tokens back onto the input stack. The token is then examined by the word scanner and the process repeats. When the word scanner detects a step boundary it triggers the data step compiler.

Macro Compiler:	compiles SAS code into machine readable code.
Word Scanner:	Examines tokens.
Macro Processor:	Processes macro logic and looks values up in symbol table.
Symbol Table:	Holds macro variables.
Input Stack:	The SAS code.

MULTIPLE AMPERSANDS

Since the macro processor sends its result back to the input stack, which then passes it to the word scanner which can possibly send it back into the macro processor for further processing; we are able to have the same tokens rescanned multiple times. This can be useful when you want to have multiple levels of indirection, or create compound macro variable names.

When you put a macro variable as the value to another macro variable, use multiple ampersands to force SAS to rescan the variable. Although multiple ampersands may look daunting, they can be quite easy to understand as long as you stick with two rules:

1) && resolves to &
2) Each token is handled independently

Consider this simple case:

```
%let A = C;
%let B = A;

%put &&B;
```

At first blush, you would think &&B would resolve to C. But if you follow our two rules, && is a token and is resolved to & which is placed back onto the input stack resulting in &B. Because of the two ampersands, the text is rescanned and resolves to A. In this case, the only difference between &&B and &B is an extra rescan.

Consider:

```
%put &&&B;
```

Here, && is a token and resolves to &. The second token is &B which resolves to A. So now we are left with &A which is rescanned and resolves to C.

If you follow the two rules, you can see there generally is no reason to go beyond three ampersands.

COMPOUND VARIABLE NAMES

Most of the time when multiple ampersands are used, it is to reference compound macro variable names. Generally there is a numeric ending such as &&VAR&I.

```
%let var1 = one;
%let var2 = two;
%let var3 = three;
%let total = 3;

%do I = 1 %to &total;
   %put The value of VAR&I is &&VAR&I;
%end;
```

To Resolve &&VAR&I using the rules above:

1) && is the first token and resolves to &
2) Var&I is the next token and resolves to var1, var2, etc
3) This leaves &var1, &var2, etc which is rescanned and resolved.

DELIMITING VARIABLE NAMES

There are several characters SAS uses to recognize the end of a macro variable name. Consider the example above where we used &&VAR&I. SAS knows &&VAR is one variable because it sees the next & character. What if we wanted to concatenate a macro variable name to some text? Suppose we want to use a macro variable &MODEL as a prefix to the name of a data set. We cannot simply write &MODELresults. The word scanner would see &MODELresults as one token instead of two. You have to use the stop character to delimit the name of the macro variable so it is handled as one token. The stop character is a period. Using the stop character, your data set name would look like &MODEL.results. The period is "gobbled up" by the word scanner and does not appear in the resulting text. Therefore, if you need a period in the resulting text you must use two. Consider a situation where your macro variable contains the name of your library &LIBRARY. You cannot simply write:

```
set &LIBRARY.dataSet;
```

The period in the above set statement would be seen by the word scanner as a stop character and gets gobbled up. You have to use two periods.

```
set &library..dataset;
```

PASS BY REFERENCE

There is a specific situation where using three ampersands can make your macro code more efficient. Suppose you have a very large macro variable that contains a lot of text and you use that macro variable as a parameter to a macro. Something like:

```
%let myVar = a whole lot of text a whole lot of text a whole lot of text, etc…;

%macro passByValue(var);
  %if &var ne %str() %then %put The parameter is &var;
%mend passByValue;

%passByValue(&myVar);
```

In that case, SAS macro will store two copies of the exact same text value-- one in the global symbol table as &myVar and one in the local symbol table as &var. You can avoid copying and storing the large text value by passing the name of the global variable and using multiple ampersands inside the macro to force SAS macro to rescan it. That way the large text value is only stored in one place. This could result in a substantial memory savings.

```
%let myVar = a whole lot of text a whole lot of text a whole lot of text, etc…;

%macro passByRef(var);
  %if &&&var ne %str() %then %put The parameter is &&&var;
%mend passByRef;

%passByRef(myVar);
```

MACRO QUOTING
Since the main function of macro code is to generate code, you can often run into a situation where the characters you are generating may be interpreted by the macro processor. Your macro may have generated a special character such as % or & or a mnemonic operator such as EQ or AND. You do not want the macro processor interpreting these as part of your macro so you have to hide them. This process of hiding these from the macro processor has the unfortunate name of quoting. Sometimes it is referred to as masking or escaping characters. Note that it has nothing to do with you putting quotes around anything!

This table shows the characters that might require masking.

Blank	)	=	LT
;	(	|	GE
^	+	AND	GT
¬	--	OR	IN
~	*	NOT	%
COMMA	/	EQ	&
APOSTROPHE	<	NE	#
“	>	LE	

You use macro quoting functions to hide special characters as text so the macro processor doesn’t see them as part of the macro language. This is a list of the most commonly used macro quoting functions.

- %STR()
- %NRSTR()
- %BQUOTE()
- %NRBQUOTE()
- %SUPERQ()

These functions quote or mask the text and the macro processor rescans the result. However, functions that begin with NR (no rescan) keep the macro processor from rescanning the result of the function. These are useful for situations where an & or % needs to be maintained.

COMPARING THE FUNCTIONS
The first quoting function to look at is %STR(). It is used to mask text while the macro is compiling. It masks the characters and mnemonic operators in the above table.

By Default it lets you mask these characters when they are balanced:
' “ ()

In addition, if the above characters are not balanced, you can use the percent sign to mask the single character. You simply put the percent sign in front of the character you would like to mask:

```
   %let name = %str(O%'Connell);
```

%NRSTR() is exactly the same as %STR() except it additionally masks % and &.

There are a few cases where %STR() can be omitted, but using it makes the code easier to read and understand. A good example is when you are checking a macro variable for a blank or null value. You can simply code:

```
   %if &myVar = %then %do;
```

The macro processor is smart enough to know you are checking for a blank, but it is not readily apparent to the next programmer reading your code. Did you intend to check for a blank or did you forget to put a value there? A clearer way to code it would be to specifically look for a blank using %STR( ).

```
   %if &myVar = %str( ) %then %do;
```

Use %STR() in these situations:
When you want a semicolon treated as text rather than as part of the macro language.
You want to maintain blanks as part of a macro value.
You have an unmatched quotation mark or parentheses.
When passing parameters that contain special characters and mnemonics.

Both %STR() and %NRSTR() are the only macro quoting functions that take affect during compile time. That is, they return their values while the macro is compiling. All the other macro functions do their work during macro execution.

The macro functions with B in their name are for quoting unmatched quotation marks and parentheses. Just like using %STR() with the percent sign above.

The macro functions %BQUOTE() and %NRBQUOTE() are the execution time equivalent of %STR() and %NRSTR().

TO SUMMARIZE

If the value may contain an & or % use one of the NR functions.

Use %STR() and %NRSTR() during compile time to mask character literals in your code, such as during an assignment statement:

```
   %let myVar = %nrstr(SANFORD&SON);
```

Use %BQUOTE() and %NRBQUOTE() to mask the resolved values of variables during macro execution time. Such as:

```
   %if %nrbquote(&myVar) = %nrstr(SANFORD&SON) %then %do;
```

In the above statement, the value of &myVar in the left operand is not known until execution time so we use one of the execution time functions. In addition, the value may contain an & that we do not want rescanned. However, for the right operand we want to quote SANFORD&SON during compile time since that value won't be changing.

GLOBAL AND LOCAL SYMBOL TABLES

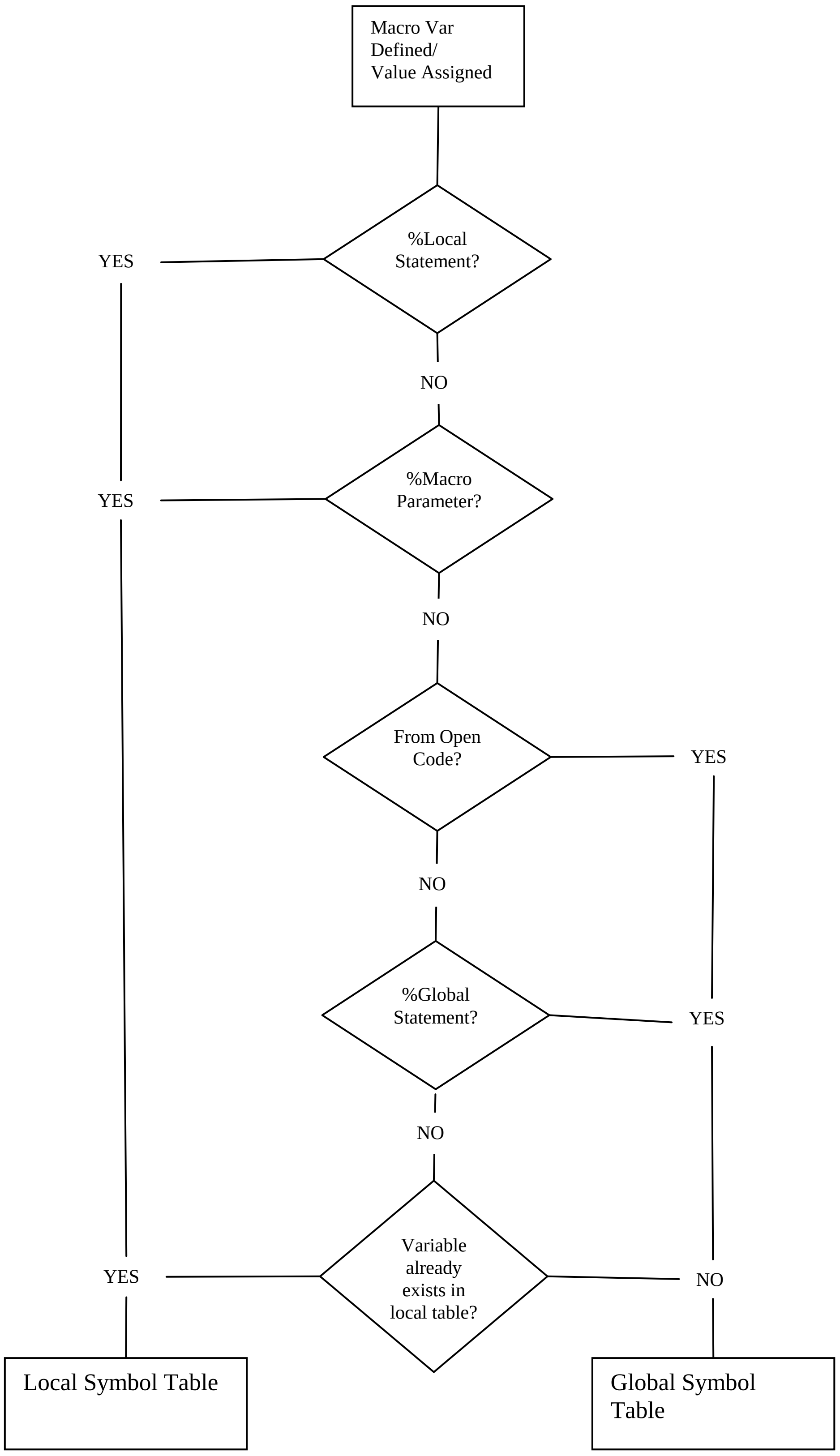
SAS Macro stores macro variables and their values in symbol tables. There are two types of symbol tables: the global symbol table and local symbol tables. The global symbol table is created at SAS startup and contains all the automatic macro variables and any user-created macro variables in open code. In this case, open code is any macro code outside of a macro definition.

The local symbol tables are unique to each macro definition. They store the values of the macro variables defined within that particular macro. There are as many local symbol tables as there are levels of macro nesting.

HOW MACRO VARIABLES ARE DEFINED

The following flow chart illustrates how SAS decides to create or use the variable in the local symbol table or the global symbol table.

Global Or Local Symbol Table?



The general rule for resolving/creating variables is to start at the most local level and keep looking up the levels until the variable is found and use/change its value there. If the global level is reached and the variable is not found, then go all the way down to the most local scope and create it there. However, as you can see in the chart above, there are two cases where a variable will be created in the local table no matter what (%LOCAL, %MACRO parameter). The same variable name can appear in both the local table and the global symbol table.

This first macro will change the value of &var in the global symbol table.

```
* declare it in open code and it will be created in global table;
%let var = global;

%macro checkGlobal();

%* this will change the value of the variable in the global table;
%let var = local;

%put var is &var;
%put _user_;
%mend checkGlobal;

%checkGlobal;

LOG:
var is local
GLOBAL VAR local
```

Here the macro will create two variables-- one in the global table and one in the local table.

```
%let var = global;

%macro checkGlobal();

%* this should force two instances of the variable local/global;
%local var;
%let var = local;

%put var is &var;

%put _user_;

%mend checkGlobal;

%checkGlobal;

LOG:
var is local
CHECKGLOBAL VAR local
GLOBAL VAR global
```

You may be wondering what happens if the same variable name appears in two different tables? Which value is used? SAS macro uses the most local scope. If there is a conflict, the local will mask the global.

Sometimes a problem can arise when a programmer assumes a variable is local when it is global. This can cause the value of the global variable to be changed unexpectedly. Consider the following code:

```
%let T = this is my title;

%macro loopThroughValues();
   %do t = 1 %to 10;
    %end;
%mend loopThroughValues;
```

Since T was created in the global symbol table, it is not recreated in the local symbol table. The global value is overwritten and the title then becomes '10'.

These situations can be avoided through name mangling. Name mangling is simply prefixing each variable name with its scope identifier. This helps avoid duplicating variable names and masking values. There is no single naming convention used, but sticking to something consistent can help when you are developing a large macro library. In this case I will use the prefix 'g' for global and a shortened version of the macro name 'ltv' for local.

```
%let gT = this is my title;

%macro loopThroughValues();
   %do ltvT = 1 %to 10;
    %end;
%mend loopThroughValues;
```

The above code could also be re-written using an explicit %local statement in the macro body. In that case the variable would exist in the local table and not affect the value in the global table.

What if you want to make sure you use the global variable and reach past the masking of a local variable? Consider a situation where you are working with a large macro library. There is a global macro variable called &setOptions which sets all the printer options. Unfortunately, when you use that variable in your macro you see that another programmer has used %local to create that variable in a local scope above yours. You are hesitant to change the other macro because of dependencies. You need a way to bypass the scope resolution and get the global value. I have written a utility macro called %rtnGlobal(). It will always return the value of a variable from the global table even if it also exists in a local table. It is included here for those odd situations where you might need it, and also to serve as a code example of some of the topics covered in this paper!

```
%macro rtnGlobal(rtngVAR=, rtngDEF=NULL);

  %* %rtnGlobal by Stephen Philp pelicanprogramming.com
                                datasteps.blogpspot.com
     09/18/2007;

  %* rtnGlobal is a function style macro that returns the value of a
  ** global macro var.  This can be useful because a local variable can
  ** "mask" a global value;
  %* Parameters:
  ** var= the global variable you would like to get the value from
  ** def= a default value returned if the variable does not exist
  ** If the variable does not exist, this macro will just return the
  ** default value.  It will not create the macro variable;


  %* check the parameters to make sure a value was passed in;
  %if &rtngVAR = %str()
   %then %do;
            &rtngDEF
                 %return;
            %end;

  %* open up the view of the macro symbol tables available from sashelp;

  %* make sure our var is upcased....;
  %let rtngVAR = %upcase(&rtngVAR);

  %let rtngDSID = %sysfunc( open(sashelp.vmacro,i) );  %* we cant use where=, cause
its a view;

  %* now that we have it open get the variable number for scope, name and value;
  %let rtngSCOPE = %sysfunc( varnum( &rtngDSID, SCOPE ) );
  %let rtngNAME = %sysfunc( varnum( &rtngDSID, NAME ) );
  %let rtngVALUE = %sysfunc( varnum( &rtngDSID, VALUE ) );

  %* cant get the number of observations cause its a view...;
  %* now loop through each observation of our view by fetching it
     into the ddv and looking at the values;


  %do %while( %sysfunc( fetch(&rtngDSID) ) = 0 );

    %if %sysfunc( getVarC( &rtngDSID, &rtngSCOPE ) ) = %str(GLOBAL)
    and %sysfunc( getVarC( &rtngDSID, &rtngNAME ) )  = %str(&rtngVAR)
     %then %do;
              %let rtng = %sysfunc( getVarC( &rtngDSID, &rtngVALUE ));
                   %str(&rtng)  %* return the value to the caller;

                   %let rtngRC = %sysfunc(close(&rtngDSID));
                   %return;
         %end;

  %end;

  %let rc = %sysfunc(close(&rtngDSID));

  %* otherwise, if we made it here then the variable was not found in global scope;
  %str(&rtngDEF)

%mend rtnGlobal;

* here is a little macro to test it out;

%let x = this is global;

%macro testit;
%local x;  %* now there are two instances of x, one in the global and one in the
local;
%let x = this is local;

  %let xG = %rtnGlobal(rtngVAR=x);
  %put xG = &xG;

  %let xL = &x;
  %put xL = &xL;

  %put more macro code;
%mend;
%testit;
```

%SYSFUNC()

The macro function %SYSFUNC() lets you use data step functions within macro code. This can be a powerful way to stay within the macro language without having to go to a data step to retrieve a value and then push it back to macro. Consider a situation where your macro needs to know how many observations there are in a data set. This can be accomplished with a data step such as:

```
data _null_;
  set myDataSet nobs=nobs;
  call symput('nobs', strip(nobs) );
stop;
run;
```

This code is analogous to the following code opening the data set explicitly using functions:

```
data _null_;
  dsid = open('myDataSet');
  nobs = attrn(dsid, 'nobs');
  rc = close(dsid);
  call symput('nobs', strip(nobs) );
run;
```

We can skip the data step and the CALL SYMPUT() altogether by using %SYSFUNC(). There are a few things to keep in mind when using %SYSFUNC().

1) You cannot nest functions within one %SYSFUNC() call. Each function needs its own %SYSFUNC(), but you can nest %SYSFUNC() functions.
2) All arguments to SAS language functions have to be separated by commas. You cannot use the OF shorthand such as SUM(OF q1-q10).
3) Because %SYSFUNC() is macro and all macro is text, you do not put quotes around character values.

Here is code to check the number of observations in a data set within a macro using %sysfunc():

```
%let dsid = %sysfunc( open(myDataSet) );
%let nobs = %sysfunc( attrn(&dsid,nobs) );
%let rc = %sysfunc( close(&dsid) );

%if &nobs gt 0 %then %do. . .
```

Notice we are using the same functions as the data _null_ step, but we did not use quotes for the parameters. Here is an example of nesting %SYSFUNC() calls:

```
%let previousMonth = %sysfunc( intnx( MONTH, %sysfunc( today() ), -1  ) );
```

In the above line of code we are using %SYSFUNC() to call the TODAY() function as the second argument to the INTNX() function which will back the date up to the previous month.

There is also the %QSYSFUNC() function which works exactly as %SYSFUNC(), except it provides automatic quoting/masking of special characters.

Most any SAS language function will work with %SYSFUNC() except these:

- dif()
- dim()
- hbound()
- input()
- lag()
- lbound()
- put()
- resolve()
- symget()

Instead of put() and input() you can use putC()/putN() and inputC()/inputN().

MACRO EXPRESSIONS

There are several ways of creating and comparing macro variables. Not only can you create macro variables in the macro language, you can also create them using the data step, proc Sql and SAS/CONNECT®. There are also some specific macro operators and functions for comparing values.

COMPARING NUMERICS

All macro code is text. There are no numeric values except those used in specific situations where SAS macro handles a value as an integer. There are no floating point values in SAS macro. All macro math is performed on integers. Generally SAS macro will treat a number as an integer in logical expressions and do loops. It will not treat a number as an integer in an assignment statement. To force SAS macro to evaluate numbers as integers instead of text, use the %EVAL() function. This tells macro to perform arithmetic evaluation on the expression. Consider the following:

```
%let A = 100+1;
%let B = %eval(100+1);
%put A is &A and B is &B;

LOG:
A is 100+1 and B is 101
```

The %EVAL() function only performs integer arithmetic. Consider:

```
%let C = %eval(6/5);
%put C is &C;
```

If you submit this you will see the value of C is 1. The %EVAL() function is performing division on integers and returning an integer value. Similarly, if you compare two numbers in macro code, the comparison will be treated as integers, unless there is a period. Then the comparison will be character, resulting in unintended consequences.

```
%macro compare(first, second);
  %if &first > &second %then %put &first is greater than &second;
  %else %if &first = &second %then %put &first is equal to &second;
  %else %put &first is less than &second;
%mend compare;

%compare(100, 200);
%compare(d, f);
%compare(10, 2.0);

LOG:
100 is less than 200
d is greater than f
10 is less than 2.0
```

The period in 2.0 causes SAS to compare as text.
To compare floating point values or to perform arithmetic on floating point values, you must use the %SYSEVALF() function.

```
%macro compare(first, second);
  %if %sysevalf(&first > &second) %then %put &first is greater than &second;
  %else %if %sysevalf(&first = &second) %then %put &first is equal to &second;
  %else %put &first is less than &second;
%mend compare;

%compare(10, 2.0);
```

```
Log:
10 is greater than 2.0
```

Actually, writing the equals comparison with %SYSEVALF() is not technically correct. Floating point values are approximate representations and should never be tested for equality.

CALL SYMPUT

CALL SYMPUT() is used to create a macro variable from a data step value. It takes two parameters. The first parameter is the name of the macro variable you are creating. It can be a character literal or an expression which resolves to a valid macro variable name. The second parameter is the value of the macro variable.

Here is a simple example creating a macro variable named var with the value from a data step variable x:

```
Data _null_;
  x = 'this is a data step variable';
  call symput('var', x);
run;
```

Here is another example using a data step variable to store the name of the macro variable:

```
data _null_;
  x = 'this is a data step variable';
  y = 'var';
  call symput(y, x);
run;
```

You can also create multiple macro variables using an expression:

```
data stuff;
  do x = 1 to 10;
    output;
   end;
run;
data _null_;
  set stuff end=eof;
  call symput( cats( ‘VAR’, _N_ ), strip(x) );
  if eof
   then call symput(‘total’, strip(_N_) );
run;
```

Here we are creating macro variables named var1, var2, etc containing the value of the variable x for each row of the table. We are also creating a final macro variable called total which holds the value of _N_. There are a few things to keep in mind when creating variables this way. First of all, if the value is numeric we have to remember SAS macro will transform it to character. This can happen automatically but the resulting character will be right-aligned and contain unwanted spaces preceding the value. I have used the STRIP() function to strip leading and trailing blanks. Additionally, I have used the CATS() function to strip blanks from _N_ and then concatenate the value to ‘VAR’.

Another thing to keep in mind when using CALL SYMPUT() is the macro variable will not be available until the data step is finished. This can cause a problem if you omit the run statement closing the data step before using the macro variable.

```
%**** BAD CODE USED AS ILLUSTRATION ONLY;
%macro test;
  data stuff;
    do i = 1 to 10;
      output;
    end;
  run;

  data moreStuff;
    set stuff end=eof;
    if eof then call symput('total', strip(_N_) );
    %if &total < 10
     %then %do;
              %put the data set has less than 10 obs;
             %end;

  proc sort data = moreStuff; by i; run;
%mend test;
%test;
```

SELECT INTO

You can create macro variables in SQL by using the INTO clause on the select statement. The INTO clause follows the same scoping rules as the %LET statement. You can also create multiple macro variables using a single INTO clause. The basic syntax is:

INTO :macro-variable-specification-1 < ..., : macro-variable-specification-n>

A simple example is:

```
Select count(*) into :total from work.attendees;
```

You can also specify multiple variables:

```
Select count(*), min(age) into :total, minAge from work.attendees;
```

In the above code, I am using the count() function and the min() function to get two values which will be put into the two macro variables &total and &minAge. Note, you do not need to use “roll-up” functions with the INTO clause. You can use any column in the table you are choosing from. However, I believe it illustrates clearly that one “set” of macro variables is created. Not a set from each row of the table you are reading from. If you would like to create a macro variable for each row of the table you are reading from you use the THROUGH (-) modifier:

```
data stuff;
  do x = 1 to 10;
    output;
  end;
run;

proc sql noprint;
  select x into :var1 - :var10 from stuff;
quit;
```

This code will create macro var1 through macro var10 with the values of x from each row of the table.

Another useful modifier is SEPARATED BY. This lets you specify one macro variable to hold all the values from a column.

```
Proc sql noprint;
  Select firstName into :allNames separated by “ “ from work.attendees;
Quit;
```

This will create one macro variable named &allNames which will contain all the first names separated by a space. You could choose any delimiter you like. This makes it easy to loop through all the values using an algorithm like the one below:

```
%* &names holds names separated by spaces like: STEPHEN MIKE TRACY YING KIM ED;
%* Loop through all the values in the &names variable;
%let i = 1;
%let name = NULL;
%do %while(&name ne %str( ) );
 %let name = %scan(&names, &i);
 %let i = %eval( &i + 1 );
  %if &name ne %str( )
    %then %do;
             %put name has the value &name;
          %end;

%end; %* end of do while;
```

The above code is using a do while loop and a sentinel variable to treat &names like an array and loop through all the values.

SYSLPUT

The %syslput statement is used to create or modify a macro variable on a remote server using SAS/CONNECT. It has the same syntax as a %let statement with an additional remote= parameter:

```
%syslput macrovariable = value / remote= myServer;
```

The %sysrput statement does nearly the same thing but in reverse. It creates or modifies a macro variable on the local host based on a macro variable on the remote host. It has the syntax:

```
%sysrput localMacroVariable = remoteMacroVariable;
```

The %sysrput statement has the same scoping rules as %let.

DEBUGGING

Due to the layered complexity of SAS Macro interacting with and producing regular SAS statements, SAS macros are notoriously difficult to debug. Indeed, the topic of debugging macros could fill its own paper. So I will just illustrate some specific debugging cases.

MACRO COMMENTS

For some reason the macro processor does not handle a macro comment with an unbalanced quote correctly. This has bitten me more times than I would like to admit.

```
%**** BAD CODE USED AS ILLUSTRATION ONLY;
%macro test;
  %* macro comments shouldn't contain unbalanced quotes;
  %put Unbalanced quotes cause a problem in macro comments.;
  %put Unbalanced quotes cause a problem in macro comments.;
  %put Unbalanced quotes cause a problem in macro comments.;
  %put Unbalanced quotes cause a problem in macro comments.;

%mend test;
%test;
```

FILENAME MPRINT

Sometimes you are put into the unfortunate position of debugging a large macro that you didn't write. This can be especially frustrating when the bug is in the generated code and not the macro logic. You can use the systems options MACROGEN and MPRINT to have the macro generated statements written to the log. But the output can be very messy and difficult to trace. Not to mention, in order to test any changes to the generated code, you have to re-run the whole macro. Using a FILENAME MPRINT with OPTIONS MFILE will tell SAS to write the macro generated code to a file.

```
Filename mprint “/tmp/code_to_debug.sas”;
Options mfile;
```

Then you can work with the generated code in its own file.

MACRO DESCRIPTIONS

When you are creating a large macro library, it can often get confusing as to where a macro definition is defined. Especially if the code generating the macro is in startup files, hidden files, files in users directories, etc. There is an option on the %macro statement called description. I have found it useful to put the name of the SAS file into the macro's description field. If you do this for all your macros then you will always know where to look when you want to inspect a macro definition.

```
%macro setUser(username=) / des = “/home/stephen/sasCode/setUser.sas”;
```

The following macro named getDescr takes the name of a macro as its parameter and returns its description.

```
%macro getDescr(macroName) ;
  %local r;
  %let macroName = %upcase(&macroName);
  proc sql noprint;
    select objdesc into :r
    from dictionary.catalogs
    where memname = 'SASMACR' and
          objName = "&macroName"
    ;
  quit;
  %put &macroName --> &r;
%mend getDescr;
```

PERSISTENT MACRO VARIABLES

If you are debugging code that contains macro variables (including macro code) you have to keep in mind that macro variables in the global symbol table will exist until the SAS session is shut down. Sometimes this can cause confusion. Especially when debugging macros interactively. To see all the variables you have created use %put.

```
%put _user_;
```

To delete a macro variable you can use the %symdel() function.

```
%symdel(myVar);
```

It can be a good idea to delete global macro variables when debugging interactively.

AUTOCALL LIBRARIES

If you are creating a lot of macros that will be used by many users, a good way to organize them is in an autocall library. An autocall library is simply a directory where the macro source code is stored. Each macro must have it's own source file and the file must be named the same as the macro. An example of an autocall library is:

```
Directory of C:\Program Files\SAS\SAS 9.1\core\sasmacro

02/12/2007  03:33 PM    <DIR>          .
02/12/2007  03:33 PM    <DIR>          ..
03/02/2004  07:28 AM             1,155 af.sas
03/02/2004  07:28 AM             1,941 angle.sas
03/02/2004  07:28 AM            36,995 annomac.sas
03/02/2004  07:28 AM             7,407 armconv.sas
03/02/2004  07:28 AM             6,506 armend.sas
03/02/2004  07:28 AM             3,909 armend2.sas
03/02/2004  07:28 AM            11,137 armgtd2.sas
03/02/2004  07:28 AM            14,507 armgtid.sas
03/02/2004  07:28 AM             4,289 armini2.sas
03/02/2004  07:28 AM            10,678 arminit.sas
03/02/2004  07:28 AM            19,719 armjoin.sas
03/02/2004  07:28 AM            14,439 armproc.sas
```

In fact, this is a partial listing from the SAS Institute supplied autocall library installed with SAS. As you can see, the name of the SAS program is the same as the name of the macro.

After you have put your source code in a library just follow these steps in SAS to use the macros:

1) Associate a fileref SASAUTOS with your directory. (filename sasautos "z:\shared macro directory\";)
2) Turn on the MAUTOSOURCE SAS system option.
3) Invoke the stored macro.

Invoking a macro that has not been defined in the current SAS session will cause SAS to search the autocall library for the source file named the same as the macro. If it does not find a match it will issue an error. SAS will automatically include the macro source, compile it and submit the macro if it's found.

CONCLUSION

The SAS macro facility is multilayered and complex. When considering advanced macro topics, some topics will invariably get left out or receive less attention than others. Some of the topics this paper did not cover:

Nested macro definitions.
Call execute().
Macro coding conventions.
Macro variables and definitions in SQL dictionary tables.
Compiled macros.

I tried to stick to topics I believe are the most useful and improve the average macro programmer's coding abilities. If you have any suggestions, comments or questions please feel free to contact me using the information below.

ACKNOWLEDGMENTS

Thank you to all the people who have written and shared their knowledge about SAS macro before me. No work stands alone, and certainly this paper was influenced by those that came before it. Thanks also to SAS Institute and everyone who takes the time to read this paper.

CONTACT INFORMATION

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

Stephen Philp
Pelican Programming
2806 Mc Bain Ave
Redondo Beach CA 90278
E-mail: Stephen@pelicanprogramming.com
Web: www.pelicanprogramming.com
http://datasteps.blogspot.com