PharmaSUG 2014 - Paper BB12

BASICS OF MACRO PROCESSING - 'Q' WAY

Usha Kumar, inVentiv Health Clinical, Pune, Maharashtra, INDIA

ABSTRACT

Macros have been one of the challenging areas of SAS® programming. This paper is an attempt to make the fundamentals of macro processing easy through the 'Q' way. It is so named because we will look at the Macro Quoting functions to understand how the Macro works. There are numerous articles on the Macro Quoting functions. To understand the stages of Macro processing is what is differentiating this paper with other papers/article on the subject.

INTRODUCTION

Macro processing occurs in two stages - compilation and then execution. Compilation refers to one pass through the code that performs syntactical checks and translates the original code to executable form. Execution refers to the second/final pass through the compiled code and executes it. It is observed that many of the errors in macros occur because our lack of clear understanding of these two stages.

In this paper, we will study through examples, the difference between compilation and execution using the compile time quoting functions %STR, %NRSTR and execution time quoting functions %BQUOTE and %NRQUOTE. We will also observe that even at the time of execution, macro statements execute prior to DATA step statements.

MACRO PROCESSING

Macro is all about replacing any large amount of text with just a named reference to it. This helps in bringing about lot of efficiency in programming by reducing the amount of coding that needs to be written, thereby helping re-usability and portability. The key tool in writing macros is the macro variable. These are variables which when referenced will be replaced with the text stored against that variable in the symbol table. Macro variables are created and referenced using special characters "%" and "&" respectively. These are also called as Macro triggers as they tell SAS to invoke the Macro processor as per the syntax of Macro language.

There are many ways to create macro variables; Let us look at one of the way, using %LET statement.

A simple assignment statement that assigns the text "7o clock" to a macro variable named *time*

```
%LET time=7o clock;
```

The created macro variable can be referenced using %PUT statement by prefixing an ampersand sign besides the macro variable name.

```
%PUT &time;
```

The above line of statements runs without any error printing the text *7o clock* in the Log.

WHAT REALLY HAPPENS?

The code when submitted undergoes above mentioned stages of processing viz., compilation and then execution. At the compilation stage, syntactical checks are performed as per the syntax rule of the macro language and all macro code is resolved into compiled code. At the execution stage the compiled code is then executed. Compilation precedes execution. We consider an example to see this happening.

Code Submitted

```
%LET time=7o clock;
%PUT &time;
```

Compilation

When the first line of macro statement %LET is compiled, macro variable *time* is created in the symbol table with the value of "7o clock" stored against it. At compile time of %PUT, the reference to *time* is searched for, in the symbol table and is replaced by the stored text value.

Compiled code looks like –

```
%PUT 7o clock.
```

Execution

Now the compilation is complete and the next stage of processing i.e. execution stage begins. At the time of execution what we now have is to execute just the compiled code i.e. %PUT statement. This will print the already resolved macro variable from compiled code, as expected, in the log.

Display from Log:

```
7o clock
```

That was simple. Isn't it?? Now that we have understood the basics of compilation and execution, let us look at some more involved examples using macro quoting functions.

EXAMPLE 1:

Now, suppose we want a slightly different text to be assigned to the macro variable *time*, say, "7'o clock". Here, we are just introducing an extra single quote sign. Along with it, we are also defining another macro variable *time1.*

Code Submitted

```
%LET time=7'o clock;
%LET time1=in the evening';
%PUT &time;
%PUT &time1;
```

We might expect the above to work without any problems, as in the earlier case and print the text "7'o clock" and "in the evening' "to the log on 2 different lines. However, that does not really happen and we see the text "7'o clock;%LET time1=in the evening'" printed to the log in one line along with a warning message as below

Display from Log:

```
7'o clock;%LET time1=in the evening'
WARNING: Apparent symbolic reference time1 not resolved.
```

We will see in detail what has gone wrong.

Compilation

At the time of compilation, we would expect the macro variable *time* and *time1* to be created in the symbol table with values "7'o clock" and "in the evening'" respectively and the %PUT statement to resolve as below for the macro variable *time*

```
%PUT 7'o clock;
```

However, what happens actually, when the compiler compiles the first assignment step, it finds a single quote in the value and hence looks for matching closing single quote as it would treat everything between the quotes as literal constant as per the syntax rule of the macro language. It continues to read until we see the closing quote which is in the second assignment statement. So, the whole text *'o clock….evening'* now becomes part of the assignment to the macro variable *time* resulting in a value that is not intended.

Compiled code looks like –

```
%PUT 7 'o clock;%LET time1=in the evening '
WARNING: Apparent symbolic reference time1 not resolved.
```

Execution

When the compiled code is executed, we will see unintended text displayed for *time* macro variable and a warning message indicating that the macro variable *time1* cannot be resolved. This is because due to unbalanced quotes, the second macro statement was treated just as a plain text by the compiler and not as a macro assignment statement.

Display from Log:

```
7'o clock;%LET time1=in the evening'
WARNING: Apparent symbolic reference time1 not resolved.
```

How do we fix this problem?

Quoting functions come to our rescue. These functions mask or quote the real meaning of a character.

We will use the compile-time quoting function %STR to mask the quote sign. In order to mask these special characters, apart from using the %STR function, we also need to precede the single quote character with a percent sign.

Code Submitted

```
%LET time=%STR(7%'oclock);
%LET time1=%STR(in the evening%');
%PUT &time;
%PUT &time1;
```

Compilation

There is no syntax error detected this time at compile time as the single quotes are masked by the compile time function %STR. The text "7'o clock" and "in the evening'" gets stored in the symbol table for the macro variables *time* and *time1* respectively.

Compiled code looks like

```
%PUT 7'o clock;
```

```
%PUT in the evening';
```

Execution

At execution, %PUT prints whatever follows it in the line, up until it encounters the semicolon sign.

Display from Log:

```
7'o clock
IN THE EVENING'
```

This time, the value of both the macro variables *time* and *time1* are printed correctly, as expected, in the log.

So, we see that the use of the quoting function %STR has helped us understand how the code undergoes the stages of compilation and execution.

EXAMPLE 2:

We shall now create a macro variable with the value that has the character "&". Note that this character "&" being a macro trigger character, SAS looks for a macro variable to follow it, as per the macro syntax. Let us see what happens if we use %STR to mask this special character.

Code Submitted

```
%LET rhyme=%STR(Jack&Jill);
%PUT &rhyme;
```

Display from Log:

```
WARNING: Apparent symbolic reference JILL not resolved.
WARNING: Apparent symbolic reference JILL not resolved.
Jack&Jill
```

Compilation

When the above step is submitted, SAS will compile the code first by trying to resolve the macro reference because "&" is encountered in the text value assigned and throws a message in the log indicating that no macro variable of the name Jill is found following "&" sign.

Compiled code looks like

```
WARNING: Apparent symbolic reference JILL not resolved.
%PUT Jack&Jill;
```

Execution

So, the compiled code, on execution, displays the same error message in the log as in the compilation stage though the value gets printed.

Display from Log:

```
WARNING: Apparent symbolic reference JILL not resolved.
Jack&Jill
```

So, we see above that the same warning message appears twice in the log since an attempt to resolve the macro reference is made at first, at the compilation stage itself as part of syntax check and then at the execution stage.

Let's assure our understanding of the difference between compilation and execution using another macro quoting character function %NRSTR. NR stands for No rescan. This behaves exactly like %STR and masks all the special characters including the macro trigger characters "%" and "&" at compile time.

We repeat the same example as above this time, using the macro quoting function NRSTR.

Code Submitted

```
%LET rhyme=%NRSTR(Jack&Jill);
%PUT &rhyme;
```

Compilation and Execution

When we submit the above statement, there is no error message in the log as compilation is successful since the special character "&" is masked by the macro function NRSTR. The compiled code will have the text "Jack&Jill" assigned to the macro variable *rhyme* as is, without interpreting the "&" sign, due to it being masked, treating it as just a part of plain text. At the execution stage, since the macro variable is already resolved at the compilation step, the text "Jack&Jill" is printed in the log.

Display from Log:

```
Jack&Jill
```

From the basic Illustrations, we have so far seen how the macro is first compiled and then executed.

DETAILED VIEW

Let's see few more examples that would make us more confident of this concept of stage-wise macro processing.

Example 3:

Code Submitted

```
%LET param=%STR(a,b,);
%LET param1=%STR(c,d   );
%PUT %SYSFUNC(TRIM(%STR((&param &param1))));
```

Note that SYSFUNC function is a macro function that enables us to invoke any DATA step function, in this case the TRIM function here.

Compilation and Execution

At compile time, the above statements compiles without any error or warning since the comma character is masked by the compile time function %STR and the text available to TRIM function at the time of execution is "a,b" "c,d" seen as one complete string. So the output in the log has the values as "a,b,c,d".

Display from Log:

```
a,b,c,d
```

Let us now assign values to the macro variable *param1* little differently using CALL SYMPUT routine. CALL SYMPUT enables us to create a macro variable from within DATA step.

Code Submitted

```
%LET param=%STR(a,b,);
DATA _null_;
      CALL SYMPUT('Param1',"c,d    ");
RUN;

%PUT %SYSFUNC(TRIM(%STR(&param &param1)));
```

The above, though appears the same, acts differently when it gets processed. Log prints the below message

```
ERROR: The function TRIM referenced by the %SYSFUNC or %QSYSFUNC macro function has
too many arguments.
```

Compilation and Execution

At the compile time, the macro variable *param1* will not get resolved as in the previous case since the values are still to get assigned. The assignment will happen only at the execution stage when the DATA step executes. Since the above submitted code is syntactically correct, there is no issue at compile-time. However, when the results are available from *param1* at the execution time, the comma is not masked as %STR is a compile time function whereas the value is made available only at the execution stage. So, for the TRIM function, comma from the text "c,d" is treated as a separator between the two arguments "a,b,c," and "d". As TRIM function does not expect more than one single argument, an error is thrown at the execution stage. Basically, the comma in *param* has not caused any issue as they were masked by %STR but the values from *param1* were responsible for the log message as they became available only at the execution stage.

To conclude, %STR being a compile time function does not throw an error as it does not know the value of the macro variable *param1* until execution. However, since it does not act at the time of execution, error is thrown thinking of comma to be a separator between arguments on resolution only later.

Solution to this is the execution time masking function %BQUOTE.

Code Submitted

```
%LET param=%STR(a,b,);
DATA _null_;
      CALL SYMPUT('Param1',"c,d    ");
RUN;
%PUT %SYSFUNC(TRIM(%BQUOTE(&param &param1)));
```

Compilation and Execution

The above statement compiles and executes all fine as BQUOTE masks the characters at the execution stage and thereby takes care of the error message.

We also have %NRBQUOTE which is again an execution time quoting function that works the same way as %BQUOTE but masks even the macro trigger characters % and & at execution time.

We are now clear how the macro statements are processed in 2 stages, compilation and then execution.

Extending the concept a little further to understand how the DATA step executes if it has macro statements within. We will see an example.

Example 4:

Code submitted

```
DATA test;
      Student=1;
      mark1=60;
       mark2=40;
      mark3=50;
      Average =AVG(mark1,mark2,mark3);
      IF Average>=40 THEN DO;
            %LET status=Pass;
      END;
      ELSE if Average < 40 THEN DO;
            %LET status=Fail;
      END;
RUN;
```

The average mark is clearly above 40. So, we would expect the *status* macro variable to have the value Pass. When we use %PUT to display the value of macro variable *status*, we see that the value is Fail.

```
%PUT &status;
```

Display from Log: `Fail`

Confusing?? .. Not really, the answer lies in the concept that we discussed above.

Compilation

When the submitted step are compiled, all macro variables are created first in the symbol table. So, the first LET statement creates a macro variable named status with the value Pass. The condition inside the DATA step is not looked at, as there is only syntax check at this stage. The second LET statement updates the value of the macro variable named *status* with the value Fail. So, at the end of compilation phase, the symbol table would have the value of Fail in the macro variable *status*.

Execution

When the DATA step executes, the statements inside the DATA step creates the variable *average* with the value of 50. The conditional statement is checked and nothing happens as there are not really any statements to execute from the compiled code. In the compiled code, the LET statement resolves to just creating and storing the macro variable in the memory. To get rid of the above problem, we need to create the macro variable at the execution stage and not at the compilation stage. This is achieved through CALL SYMPUT routine as illustrated below.

```
DATA test;
       Student=1;
       mark1=60;
        mark2=40;
       mark3=50;
       Average =AVG(mark1,mark2,mark3);
       IF Average>40 THEN DO;
              CALL SYMPUT('status','Pass');
       END;
       ELSE IF Average < 40 then do;
              CALL SYMPUT('status','Fail');
       END;
RUN;

%PUT &status;
```

Display from Log: `Pass`

CONCLUSION

Macro facility in SAS is a language in itself. This is quite an effective tool that helps in efficiency and reusability of code. The understanding of the 2 stages of processing - Compilation and Execution, plays a key role in getting the basics right before we leap into learning anything complex on Macro and its functionality. Complex Macro programming and error debugging and fixing can get easier with this basic understanding of the working of the macro processing.

REFERENCES

Philp, Stephen. 2008. "SAS® MACRO: Beyond the Basics"

Proceedings of the 2008 SAS GLOBAL FORUM. Redondo Beach, CA: Pelican Programming.

Rashleigh-Berry, Roland. "Macro Quoting Functions". 2012.

Available at http://www.datasavantconsulting.com/roland/mquoting.html

More details with examples on basics of macro, available at

http://support.sas.com/documentation/cdl/en/mcrolref/61885/HTML/default/viewer.htm#a001302436.htm

ACKNOWLEDGMENTS

I thank my colleagues for all their support and suggestions. Special thanks to Dr. Prashant Kirkire, Senior director, strategic resourcing group, India at inVentiv Health Clinical for his valuable guidance and inputs.

CONTACT INFORMATION

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

Usha Kumar
inVentiv Health Clinical
6th Floor, Building No.4, Commerzone, Survey No. 144/ 145, Airport Road, Yerwada
Pune – 411006, INDIA
+91 20-30569112
usha_cool@hotmail.com
www.inventivhealth.com