Macro Quoting

Toby Dunn, AMEDDC&S, Fort Sam Houston

Abstract

One of the more perplexing things for any SAS® programmer to learn is the effective use of macro quoting functions. This confusion is of little wonder, given the number of quoting functions, their similarity, and the subtle differences between when each may be applied. Choosing the optimal function or combination of functions to use is often a daunting task for even advanced programmers. The difficulties in debugging macro errors increase the complexity even more. To intelligently choose the optimal quoting function the programmer needs to know what symbols to mask and when they need masking, and how this depends on the compilation and execution of the macro and surrounding program. This paper surveys macro quoting, with an emphasis on which quoting function to use and when to use them. The paper assumes working knowledge of macro processing and the intended audience is journey-level SAS programmers and above.

Key Words: Macro Quoting, Advanced Macro, Macro Strings, Macro Masking, Macro Debugging, Macro Functions, %Str, %Nrstr, %Quote, %NrQuote, %BQuote, %NrBQuote, %SuperQ, %QScan, %QUpcase, %QSubstr

Introduction

Many people say that Macro Quoting is difficult and indeed it is one of if not the most complex topics in the SAS language for a programmer to grasp. Those people who think it is easy or simple do not have a full grasp of Macro Quoting. Yet even though Macro Quoting is a difficult subject to master, we will soon see that learning the basics is not as bad as some would have you to believe. This paper will look at Macro Quoting from its most fundamental level of what it is, why it is needed, the basic quoting functions and some examples of their use and lastly, it will give some useful hints as to which one to use and when. It is important to note that Macro Quoting will not and cannot save you from poor macro design, lack of macro language knowledge, or just out right poor programming. It will, however, provide you with a tool to overcome some obstacles with the Macro language.

The Basics Of Macro Quoting

The number one problem I see programmers make when using Macro Quoting is that they just start quoting everything without regards to what and why they are quoting these values in the first place. So before any serious discussion can take place we first need to define; what it is we are talking about, why we need it, and how is it performed. Finally,

I would like to add that we will do this by first looking at Macro Quoting from the philosophical side of the language as well as the practical side, both of which are necessary to fully understand Macro Quoting.

Macro Quoting, in essence, does nothing more than hide certain characters from the macro processor at certain times, and it marks the starting and ending delimiters in certain macro functions. SAS is just a bunch of different languages (IE. Data Step, Procedures, Libname, etc…) that are held together by Data Sets. The Macro language is some what different from these in that its sole purpose is to create SAS code. To put it another way, Data Step and Procedure code use data sets as data while Macros use SAS code as data.

Since Macros use Data Step and Procedure code as data there needs to be some way in which to distinguish between the Macro code and the SAS code that the macro is to create. To find the answer to this we need look no further than the philosophical differences between the two languages. In the non-Macro world words have meaning Data, Run, Set, Merge Proc, Quit, etc…. and it is up to the programmer to distinguish between what SAS should interpret as having meaning and what it should just consider as plain text. To do this the programmer places quotes around what they want SAS not to interpret as having some special meaning. In the Macro language everything, by default, is excluded from being interpreted as having meaning. It is the programmer's job to tell SAS what should be interpreted as having meaning by prefixing those words with either a % or &.

We know that in the Macro language % and & prefix words that SAS should interpret as having some special meaning. However, it would be pretty darn inconvient for a person to place these everywhere such as in front of every parenthesis, equal sign, comma, or semi colon in, say, a Macro function. Therefore, once a Macro instruction is begun, those pieces of code that are required for the instruction need not be distinguished. This leads us to a problem in that how does SAS distinguish who owns a parenthesis, semicolon, comma, quote marks, or even an OR, AND, LE, EQ, etc… . Quote marks cannot be used because they already have a meaning in the Macro language, so SAS decided to use functions which will, in effect, mask the meaning of these character(s) to the Macro Facility. Incidentally, ever wonder where Macro Quoting received its name from? Remember that in order to tell SAS not to interpret a word as having meaning, quotes are placed around text in non Macro languages. Well, since Macro Quoting functions have pretty much the same effect in the Macro language someone along the way put two and two together and coined the phrase "Macro Quoting". The best way to solidify all of this is to look at an example.(you started 3 of 4 sentences with the word "well" so I switched it up)

Consider the following:

```
%Let ABC = 123 ;
```

When SAS encounters the % sign it invokes the Macro facility provided it is turned on. The Macro Facility sees the % sign and immediately looks for some Keyword. In the case of our example it is Let. The %Let is a macro instruction and the macro facility then knows that it needs to create a Macro variable and to expect certain syntax. It will then take everything between the word Let and the equal sign, drop the leading and trailing spaces and, if that word is a valid macro variable name, use it. If not, then it will report an error in the log. Then it takes everything from the right side of the equal sign up to the semicolon and assigns that as the value of the macro variable ABC. Now imagine if SAS required us to prefix every bit of Macro instruction with, say, a % sign. Our example above would look like this:

```
%Let %ABC %= 123 %;
```

Think of the absurdity of this style. Imagine how hard it would be to write, read, maintain and debug. Now lets take this example a bit further and suppose we also wanted a semicolon as part of the macro variable ABC's value.

```
%Let ABC = 123; ;
```

If we were to run the code above, how would SAS know that the first semi colon is to be included in the value of ABC and not as part of the Macro code that ends the %Let statement. This is why we need Macro Quoting. Since quote marks have meaning in the Macro language we cant very well use those so we have a set of functions.

```
%Let ABC = %Str(123;) ;
```

In the above example everything happens just as before except when the value of the Macro variable ABC gets assigned. In this case the %Str gets interpreted and the macro facility starts looking for characters to hide. The only character that can be hidden by the %STR function is the semicolon. Now the Macro Facility knows which semicolon is part of the Macro instruction and which is part of the Macro variable's value.

Macro Quoting Mechanics

Now that we know what Macro Quoting is and why we need it, it is time to look at how SAS actually performs the act of masking a value. The mechanics of Macro Quoting are pretty straight forward. Whenever a Macro Quoting function is used it simply attaches an unprintable ASCII or EBCDIC character to the beginning and ending of the specified string as well as exchanges the character(s) that need to be hidden for a delta character. Then, when these delta characters need to be switched back to printable characters, it reverses the process. Not only does SAS keep track of what characters it switched with what, the leading delta characters keep track of what type of quoting was used -- more about this in the next section.

If you are (or you're, but not supposed to use contractions) wondering how SAS knows what delta character to use for what special character, it is simple: somewhere down deep in the Macro Facility SAS has two tables; one is EBCDIC which stores the translations and the other is ASCII . The special characters are: blank ; % & ‘ “ ( ) + - = * / < > ^ | , ~ -- # GE LE EQ OR AND GT LT NE IN. Then, depending on the operating system, you are running SAS will choose which of the two tables to use. For the most part the programmer will never see these delta characters and even when one does see them on the computer screen, they look like junk characters. Let us use the ■ symbol to denote a delta character and look at what quoting actually did to our previous example:

%Let ABC = %STR(123;);

The value stored by SAS is:
■123■■

If we added leading and trailing spaces to the value:
%Let ABC = %STR(123;);

The value stored by SAS is:
■■123■■■

Lucky for us, SAS keeps all of the information straight as to what delta character was switched with which special character and it does the switching for us. As I mentioned earlier, normally you do not need to worry yourself with the details however, every once in awhile when you debug a macro it is helpful to see what is getting hidden. In these cases you can simply submit %Put <_User_, _Local_, _Global_) and, depending on your editor, will be able to see the junk characters in your macro variable values. It is important to note that %Put and %Symbolgen will convert the delta characters back to their original values when printing things to the log.

There will be those who think that this section is an overkill of sorts, but experience has taught (me) it is not. An example of how this type of knowledge came in handy is when I was working with PRX functions in the Macro Facility. After many attempts and much frustration I could not figure out why the position at which my PRXMatch function returned was wrong. After some careful digging and a %Put I realized that the Macro Facility did not remove the delta characters before handing off the value to the Perl engine. This lead me to contact a friend who works at SAS who specifically deals with the Perl functions and in V9.2 of SAS the problem is (was?) fixed. Had I not known about the mechanics of how Macro Quoting works I would not have solved the problem.

Macro Quoting Functions

In all of the previous examples the Macro Quoting function %STR has been used. This was partly because it is the most commonly used quoting function and partly because it fit the problem trying be solved by the examples. However, there are in fact 16 Macro Quoting functions in all. This section will first provide a brief description of Macro Quoting functions' evolution and then proceed to explain each one separately.

The very first quoting function was %STR and it masked certain characters at macro compile time thus necessitating another function %Quote that would work at macro execution time. Given there are functions that masked things, there also needed to be a way to unmask these characters hence, the %UnQuote function. Soon after %STR and %Quote came out it became apparent that these functions could not handle values that should come in pairs and have some special meaning to the Macro Facility. These values were unmatched single and double quotes, opening and closing parentheses, and interestingly enough they found that they needed the ability to mask the % sign. So SAS augmented these functions so that by prefixing these unmatched pairs of symbols or % sign with, of all things, a % sign (is that what you mean? I didn't understand that sentence if I didn't punctuate that inner clause). However, this posed a problem with %Quote because it works at execution time and the unmatched or % sign, more often than not, was the result of resolving a macro variable. Thus, a new quoting function was introduced, %BQuote, which is often called the Blind Quoting function. %BQuote not only masked unmatched pairs in the resolve, but it also allowed the macro facility to continue to try to resolve values as long as it could. To allow the case where the programmer wanted to only have the first resolution of a macro variable occur and stop, all subsequent macro resolutions in the value %SuperQ were (plural verb for plural resolutions) created.

At some point SAS realized that there were instances whe people needed to be able to easily mask the % and/or & signs for %STR, %Quote, and %BQuote so SAS introduced the NR versions of these functions. The NR stands for No Rescan which refers to the fact that the macro facility will not try to resolve any %Macro or Macro variables inside of these functions.

Macro quoting functions come in two varieties; compile time functions and execution time functions (when introducing new things it can be good to be repetitive). Compile time functions only work on values at compile time and these functions are %Str and %NRStr. All the other functions are execution time functions and work during macro execution time. As a general rule of thumb, if you want the value from resolving a macro variable or %Macro to be quoted then use an execution time function, otherwise use a compile time function.

Compile Time Quoting Functions

%STR and %NRSTR mask special characters at compile time of a %macro or macro variable and will remain quoted until it is explicitly removed. They both mask the following characters: ‘ “ ( ) + - = * / < > ^ | , ~ -- # GE LE EQ OR AND GT LT NE IN

while %NRSTR also masks the & and % sign. These quoting functions only mask special characters at compile time. This means that they will only mask the characters that are given to the argument and not any resolved macro values. To help make this clearer let us look at an example:

```
38   %Let A = X,X ;
39   %Let B = Y%Str(&A)Y ;
40   %Let C = Y%Str(X,X)Y ;
41   %Put %Substr( &B , 4 , 1 ) ;
ERROR: Macro function %SUBSTR has too many arguments.  The excess arguments
will be ignored.
ERROR: A character operand was found in the %EVAL function or %IF condition where
a numeric operand is required. The condition was: XY
ERROR: Argument 2 to macro function %SUBSTR is not a number.

42   %Put %Substr( &C , 4 , 1 ) ;
X
```

Why did the first %Substr fail while the second did not? To answer this question let us look at the values given to the %Str function in the compile time of the macro variable B. Its values are Y%Str(&A)Y. The leading and trailing Ys are fine, they are just text. The problem lies in the fact that what was given to the %Str at compile time was &A which is a macro variable reference but at compile time this reference is not resolved so &A was fed to the %STR function. Since there are no special characters for it to match it has nothing to change into delta characters. Thus, when the macro variable B is resolved in the %Substr function, the comma between the Xs is not hidden and we get the "too many arguments to the %Substr function" error. The second example works because the %STR was give X,X and at compile time the function sees the comma and converts it to a delta character and therefore no problems occur with the %Substr function call.

Had we wanted the & in the macro variable B to be just an & and not have special meaning to the macro facility then we should have used %NRSTR and the whole thing would have worked out.

```
 43   %Let A = X,X ;
44   %Let B = Y%NRStr(&A)Y ;
45   %Put %Substr( &B , 4 , 1 ) ;
Y
```

Execution Time Quoting Functions

The macro quoting functions %QUOTE and %UNQUOTE were the first macro execution time macro quoting functions. They mask the exact same special characters as %STR and %NRSTR at macro execution time. These functions were limited by their

inability to handle unpaired parentheses, single or double quotes, and percent signs. So %BQUOTE and %NRBQUOTE were created and they superseded these, thus negating originals' usefulness. Since these functions have been superseded we will not go any further with this explanation.

%BQUOTE and %NRBQUOTE or 'Blind quoting' mask special characters at macro execution, meaning that they mask the resolved value of either a macro variable or a %macro. They mask the same characters as %STR and %NRSTR, respectively, as well as unmarked and unmatched percent signs, opening and closing parentheses, and single or double quotes. One of the interesting things is that %BQUOTE and %NRBQUOTE allow for the macro expressions to be resolved as far as possible before they quote the special characters. If a macro call cannot be resolved it issues a warning and quotes the resolved value. %NRBQUOTE will mask all of the same characters as %BQUOTE as well as mask the & and % in the final value. An example will help us to see this a little better.

```
Data _Null_ ;
Call SymputX( 'A' , 'Ben&Jerry' ) ;
Run ;

108  %Let B = %NRBQUOTE( &A ) ;
WARNING: Apparent symbolic reference JERRY not resolved.
109  %Let C = %BQUOTE( &A ) ;
WARNING: Apparent symbolic reference JERRY not resolved.
WARNING: Apparent symbolic reference JERRY not resolved.
110
111  %Put B = &B ;
B =  Ben&Jerry
112  %Put C = &C ;
WARNING: Apparent symbolic reference JERRY not resolved.
C =  Ben&Jerry
```

In the example above you will see that both function calls generate warnings. Since each of these functions tries to resolve the expression as far as possible this is inevitable. At some point, if there is a & or % that it cannot resolve, the macro facility will issue the warning. The %BQUOTE example has 2 warning messages; the first warning message is the first attempt at resolving the value. When it cannot, it tries again. The %NRBQUOTE only has one warning message because after the first failed attempt at resolving the macro value it masks the &. However, when we look at the %Put statements we see that value was generated with %BQUOTE issue a warning the value for the %NRBQUOTE value does not (I can't even guess at how to change this prior sentence so that it makes sense). This is what is meant when we say that %NRBQUOTE will mask & and % in the final value.

The %SuperQ function is the most stringent execution time quoting function. It requires the macro variable name as its argument and masks all special characters in that value. It differs form %NRBQUOTE in that it will not try to further resolve the macro expression if either a & or % sign are present in the macro expression after the first resolution.

```
Data _Null_ ;
Call SymputX( 'A' , 'Ben&Jerry' ) ;
Run ;

%Put %SuperQ( A ) ;
```

Log Shows:
Ben&Jerry.

If you need to quote a value in order to hide it from some part(s) of the macro processor it stands to reason that at some point you will need the ability to unquote these values. %UNQUOTE works at execution time and simply reverses the macro quoting.

```
%Let Name1 = %NrStr(Ben&Jerry) ;
%Let Name2 = %NrStr( &Name1 ) ;
%Let Name3 = %UnQuote( &Name2 ) ;
%Put Name1=&Name1          Name2=&Name2       Name3=&Name3 ;
```

On Log:
Name1=Ben&Jerry Name2= &Name1 Name3=Ben&Jerry

In this example we start by assigning a value to macro variable Name1 which has an & in it so we use the %NrStr function to hide the & from the macro processor at compile time. Macro variable Name2 uses the same function to stop the resolution of &Name1 in its assignment. If you were to run the code you would find that, at this point, there were no warnings or errors produced by having those &s in the values. This means we have used the correct quoting function for the job at hand. Proof that everything worked out as planned can be seen in the %Put statement. Lastly, the %Unquote function was used to remove the macro quoting from &Name2. Assigning its resolved value to macro variable Name3 and using the %Put statement, we clearly see that, yes indeed, everything worked as intended. Interestingly enough you will notice that even though we unquoted the value for Name2, SAS did not complain about the & in its value. The reason for this is that when &Name2 gets resolved it resolves to a quoted value of &Name1, which is all %Unquote sees. It then unquotes that and the macro facility resolves it to the quoted value of Ben&Jerry.

Quoting Text Functions

Given that the Macro language is basically a text manipulation language it contains a host of text manipulating functions. Anyone who has used the Macro language has used these functions. A few examples are %Length, %UpCase, %Sysfunc, etc…. . These functions always return an unquoted value regardless of whether the value was initially quoted or not. This poses a problem when you need to do something like convert everything into upper case characters for a comparison and there is an & in the middle of the value. To handle these situations SAS came out with a Q or quoted version of these functions and they can be distinguished by the first letter starting with a Q. The following is a list of these Quoting Text functions: %QCMPRES, %QLEFT, %QLOWCASE, %QUPCASE, %QSCAN, %QSUBSTR, %QSYSFUNC, and %QTRIM.

We will look at a few examples:

One of the great things about a %Macro is that the programmer can pass values via a parameter and have the %Macro make a decision based on this value to conditionally generate SAS code. It is good practice to ensure that the value you are passing will compare properly to the hardcoded value in the %If statement. To do this it is common for the passed value to have all of its values upcased or lowercased. Let us turn back to our Ben&Jerry example.

```
27   %Let Name = %NrStr(Ben&Jerry);
28   %Put Name = &Name ;
Name = Ben&Jerry
29
30   %Let Name = %NrStr(Ben&Jerry);
31   %Put Name = %UpCase(&Name) ;
WARNING: Apparent symbolic reference JERRY not resolved.
Name = BEN&JERRY
32
33   %Let Name = %NrStr(Ben&Jerry);
34   %Put Name = %QUpCase(&Name) ;
Name = BEN&JERRY
```

The first %Let and %Put statement pair the value for Name has to be quoted due to the & in the value (rework this prior sentence, it doesn't make English sense). Now if we were to compare this to a hardcoded value we would have to know that value had a capital B in Ben and J in Jerry. Since different users could pass different variations of this, the easiest way is to hard code all upper case or lower case values in the %Macro and then convert the parameter's value to match, thus always giving the %Macro a good comparison. In the second pair, %UpCase was used and, sure enough, it creates a warning message because %UpCase always returns an unquoted value. In the third pair of statements %QUpCase was used and, as expected, the value retained its quoted ampersand.

Our next example uses %Scan and %QScan. Macro array processing has, for the most part, been superseded by macro list processing methodology. As such, it becomes necessary to know how to grab certain members(?) (you use 'elements' twice here) from a list of elements. One of the easiest ways to accomplish this task is through the use of %Scan.

```
35   %Let NameList = %Str(O%'Conner O%'Tool O%'Donald);
36   %Put %QScan( &NameList , 1 , %Str( ) )
37           %QScan( &NameList , 2 , %Str( )  )
38           %QScan( &NameList , 3 , %Str( )  ) ;
O'Conner      O'Tool      O'Donald
```

The %QScan allows us scan, grab and print out each name even though the names have a single quote embedded in them. Had we used just the %Scan function the single quotes in the names would have caused SAS to think that there was a mismatched pair of quotes and the code would not have executed properly. Furthermore, since quotes have meaning, if %Scan had been used, SAS would still be looking for a closing quote mark causing further problems with any code submitted after it.

General Macro Quoting Tips

As mentioned through this paper Macro quoting is the most difficult topic that a SAS programmer will have to learn and use. It is bad enough that even advanced users get tripped up over it and those who know the most about it still, in many cases, really have to think through what is going on in the code. This section will provide a short list of simple rules which should help any user when working with macro quoting.

Rule #1.) The first rule to macro quoting is DON'T (I would leave the contraction in (instead of DO NOT as it is more dramatic or has more effect). If you think or find yourself needing to use macro quoting, stop and rethink your program or your macro design. More often than not programmers tend to use macro quoting to overcome a poor knowledge of SAS or of a poorly designed macro. Macro programming is difficult enough as it is Do not make it any harder than it has to be.

Rule #2.) For 99.99% of all the macro quoting problems you will ever face, %STR, %NRSTR, %SuperQ and %UnQuote will solve them. The other .01% of the time %BQuote will solve them. Many books and very smart people will tell you to use %BQuote and %NRBquote for most of your execution quoting functions. However, I disagree. If you have to have the macro variable continually resolving the macro expression then most likely you have some serious design issues that you need to fix. To put it another way, you did not follow rule #1. Secondly, if you use these two functions you have to remember two different functions and what each will mask and will not

mask. %SuperQ is simpler in that you will only have to remember one function and only one execution time function.

Now I did say that %BQuote is necessary for .01% of the time and here is an example of that .01% of the time, which I took from SAS-L. The question asked, "Is there a way to determine if a %Macro exists?" Ian Whitlock came up with the following solution (the code is copied verbatim from his reply):

```
%macro symmacroexists( macname ) ;
  %eval(%str(%%)&macname ^= %bquote(%&macname))
%mend  symmacroexists ;



/* test code */
%macro bigcall(mac) ;
 %if %symMacroExists(&mac) %then %do;
   %&mac
 %end; %else %do;
   %put WARNING: %superq(mac) not a macro call or it is not resolved.;
   /* nothing */
 %end;
%mend  bigcall ;

options nomerror mprint ;
%bigcall(x)

%macro q ;
  data w ; x = "abc' 1" ; run ;
%mend  q ;

%bigcall(q)
```

Ian further went on to say the following: "This code gives an example where such quoting is useful and could not be handled by %SUPERQ because it is the consequence of a macro invocation (or more generally any macro expression that is not a macro variable) that must be hidden.". This type of problem and the use of %BQuote is the exception rather than the rule.

Rule #3.) Macro quoting is nothing more than hiding one or more specific sets of characters from some part of the macro facility. Always know what you are hiding, who you want to hide it from, and when you are hiding it. If you can answer these questions then choosing which of the macro functions in Rule #2 becomes easy.

Rule #4.) Earlier in this paper I stated that a value will stay quoted until either the user specifically unquotes the value or SAS unquotes (the UNQUOTE is plural, not possessive) it, say when a %Put or %Upcase is used. If you are using a quoted macro value and you are not getting the right result and everything looks good, Unquote it. More often than not the problem is that the value is quoted and the quoting is causing problems.

Conclusions

Learning Macro Quoting is difficult and mastering it is even more so. However, learning the basics of it is not. I truly hope that this paper has given you insight to Macro Quoting as well as taught you the tools that you will need to use and learn more about Macro Quoting. Remember, follow the simple rules in this paper and you cannot go wrong.

Special Thanks To: Paul St. Louis and Dianne Piaskoski for their hard work in editing this paper. Also Ian Whitlock who's time, patience, and knowledge have helped me more than e could ever imagine.

Whitlock, Ian "**A Serious Look at Macro Quoting"**
www2.sas.com/proceedings/sugi28/011-28.pdf

O'Conner, Susan "Secrets of Macro Quoting Functions How and Why"
www.**ats.ucla.edu**/stat/**sas**/library/nesug99/bt185.pdf

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

Toby Dunn
AMEDDC&S
Fort Sam Houston. TX
E-mail: Toby.Dunn@amedd.army.mil

Join the SAS-L! @ listserv.uga.edu or Google Groups