

Paper CC-039

Parameter Passing by Name

Ian Whitlock, Independent SAS Consultant, Kennett Square, PA

Abstract

SAS® Stored Processes are used in SAS® Enterprise Guide® and many other SAS® software products.¹ Communication of the user's requests to the stored process, i.e. program or macro, is via global macro variables. The user's multiple choice selections are communicated to the SAS program or macro by three different interfaces depending on the number of selections. Consequently each program has to go through the logic of deciding which interface is appropriate whenever a selection of multiple choices is involved. A macro is presented to handle this logic in general so that the program can simply call one macro and expect a single common interface for all user choices.

The method hinges on passing parameter names instead of values. Consequently passing parameters by name instead of value will be discussed. The techniques involved will be explained, and then applied to the problem stated above. Although this problem is of interest to those faced with it, the principle is important, and applicable in many other situations.

All SAS platforms and products are appropriate to the talk. The reader should have some knowledge of macro to understand the code, but the principles can be understood by anyone with any sort of programming background.

Introduction

Using Cynthia L. Zender's example², The REGION variable from SASHELP.SHOES one might want to run a list of reports where the user can choose any number of different regions. So you might have the macro code:

```
%do i = 1 %to &reg0 ;  
    /* some code for &&reg&i */  
%end ;
```

However, REG0 isn't always assigned. The server has one variable, REG, and then makes other variables as needed. The rules appear to be:

1. 0 choices → REG is not defined (possibly defined but empty³) and REG0 is not defined
2. 1 choice → REG holds the choice value and REG0 is not defined
3. 2 or more choices → REG holds first choice, REG0 holds number of choices, and REG2 - REGn hold the choices⁴

¹ See http://support.sas.com/rnd/itech/doc9/dev_guide/stprocess/using.html for a complete list.

² Cynthia L. Zender, "Adapting Your Programs to the SAS®9 Paradigm", <http://www2.sas.com/proceedings/forum2008/024-2008.pdf>, page14.

³ It is not absolutely clear whether REG is empty due to recommendation that one use a %GLOBAL statement, or whether some of the server systems really don't create the variable.

⁴ http://support.sas.com/rnd/itech/doc9/dev_guide/stprocess/input.html gives the one HTML example:

```
CBOX = optOne  
CBOX0 = 4  
CBOX1 = optOne  
CBOX2 = optTwo  
CBOX3 = optThree  
CBOX4 = optFour
```

without specifying the rules.

It is not clear to me whether the difference indicated in footnote 3 is real and between systems, or the difference represents a change in time, or is just a mistake. Since I do not have access to any of the systems, I cannot confirm which case is true and so choose the most conservative one - REG1 may not be defined when there is more than one choice, but that choice is reliably held in REG. My point of view is that this paper is about the macro and not a lesson in working with stored processes, so I have pursued the matter.

For the ease of programming one would like to have REG0 always defined and to have REG1 to REGn always hold the selected values. My understanding is that most programmers resolve the situation by special code each time a selection of multiple choices is required from the user. It would be better to handle this by a general macro. When I have made the suggestion, I have met with either, "The system, not macro, is the place to do this.", or "The matter is too simple to bother with." It is simple, but I don't like writing the same code for each separate situation. The macro is also simple, but it requires understanding how to pass parameters to a macro that is a level higher than many macro programmers meet.

Macro Parameters

There are two types of parameters, positional and keyword. Although the positional parameters are more common, most of the time keyword parameters are better because they

1. make the using code clearer,
2. make the using code more maintainable,
3. are easier to extend, when new functionality must be added, because the macro author can assign default values, and
4. are more flexible because the user decides the order of values, and need not specify parameters with acceptable default values.

Consider the call to a macro, MAC, in code that you must maintain.

```
%mac( w, x, 1 )
```

versus

```
%mac( data = w, var = x, cutpoint = 1 )
```

In the first case, what is W? What is X? What is the significance of the value 1? In the second case we certainly know that W is an input work data set, since SAS almost always uses the parameter name DATA for an input SAS data set. In the second we know X is the name of a variable, probably in the supplied data set W, since no output data set is specified. Third the name CUTPOINT suggests the variable X is numeric and we want to divide the values into those smaller than one and those larger than one. Moreover, all of this is without any help from the name of the macro.

Now consider adding a parameter OUT, to name the output data set. In the first case: How does the macro author decide where to put the parameter? The calling code must change even if the new feature is not used. In the second case, it is no problem for the macro author, and the calling code need change only when the new functionality is required. Moreover, the macro author can assign by default the name if an output data was previously made. Now the user has the ability to change the name, but if he doesn't want this ability, he simply omits the parameter in the call, i.e. the user's code doesn't even have to change if he doesn't want the new functionality.

One of the more undocumented features of macro is that even when the macro is written with positional parameters, the caller can use the keyword style parameters and put them in any order. In this case, any unmentioned parameter has the default value null or empty.

The Rules for Resolving or Evaluating Macro Expressions

You don't need to hard code the name of a macro variable to assign it. Consider:

```
%let var = v ;
```

```
%global &var ;  %* declare V global *;
%let &var = value of v ;  %* assign V *;
```

Now to refer to the variable V in terms of VAR we need multiple ampersands. Now the line

```
%put &var = &&&var ;
```

writes

```
v = value of v
```

on the log.

In general, to create and reference macro variables in terms of other variables the programmer needs to know 1) that a dot is needed to end a macro variable reference, whenever a separator is not at the end of the reference, 2) if a dot ends a reference then it is part of the reference and eaten along with the rest of the reference by the macro facility in the process of resolving or evaluating the reference, and 3) how multiple ampersands are resolved.

To assign a variable V1 in terms of VAR, a dot is required because there must be a separator, and it must be removed, in the evaluation process between the reference &VAR and the number 1. Otherwise the subsequent scan of the expression either contains an illegal or wrong reference. Consequently

```
%let &var.1 = first choice ;
```

creates a variable V1 because the reference, &VAR., resolves to V and the 1 immediately follows, since the ending dot is part of the reference and not part of the value.⁵

The rules for resolving macro expressions consisting of text and ampersands are:

1. The expression is scanned left to right creating a new expression.
2. Two consecutive ampersands are resolved to one.
3. A single ampersand followed by text is a macro variable reference ending at a separator. If the separator is a dot, then it is part of the reference; otherwise it is not. The reference is resolved to the value indicated by that reference.
4. At the end of the initial expression the resulting expression is rescanned as a new initial expression until there are no unresolved references, i.e. the final expression is constant text.
5. Ampersands that cannot indicate a reference are treated as constant text and a warning is written to the log.

In the following examples an → is used to indicate subsequent expression as a result of scanning. From the above assignments we have:

```
&var → v
&&&var → &v → value of v
&&var.1 → &v1 → first choice
```

Given that the variable, I, has the value 1

```
&&var&i → &v1 → first choice
```

The pattern &&&VAR is extremely important because it allows one to reference one variable in terms of another variable. In other words, the name of a macro variable need not be written into the code. It can be made known via a macro parameter or variable. Consequently the resulting code can be more flexible. It is precisely this flexibility that is required to solve the posed problem about stored processes.

⁵ Some programmers suggest always using a dot to end a macro variable reference, but this policy merely proliferates the number of dots in the code and hides which dots are serving an essential purpose.

The pattern &&VAR&I is even more ubiquitous because it allows for the construction of a list of expressions that may be treated as an array of macro variables.⁶

You can build even more complex expressions with more ampersands, dots, and other letters, numbers and underscores. However, in general, the code is clearer if you create intermediate temporary (local) variables instead of trying to directly build and interpret these more complex expressions because they are hard to work with.⁷ The basic principle is construct the name you want to evaluate and then use a triple ampersand to reference the value of this name. See the Appendix to explore this idea further. However, it is usually best to avoid the need for more complex constructions wherever possible.

A Matter of Philosophy?

In the introduction, the rule, for when no choice is made, states that the parameter REG is not defined, but it may be that REG is defined and empty, since the recommendation, to use a %GLOBAL statement, forces the existence of REG. In practice, one probably rarely has a need to distinguish the two cases. However, I prefer not to rule out an empty choice, since the test for existence is easily provided by the macro function %SYMGLOBL, which returns 0 when the variable does not exist in the global table and 1 when it does. If you disagree with this policy or choose to force the issue with a %GLOBAL statement, then simply change the test for existence from

```
%synglobl(&parm) = 0
```

to⁸

```
%length(&parm) = 0
```

or possibly

```
%superq(parm) = %str()
```

In general I prefer the %LENGTH test for an empty parameter on the assumption that anyone who wants to fool the test by quoting blanks deserves what comes of it.

The Macro

We now have all the tools to write the macro. What should we call it? I chose STDPARM to suggest that the function of the macro is to present a parameter in a standard manner, i.e. one which is fixed and consistent. What does the macro need to do this job? The name of the parameter is required, not its value, because it is the name that must be manipulated to create new variables. However, the value must also be available, but in the section on evaluating expressions we have seen how this can be done even when the name is held in another variable. What should we call the macro's parameter? I chose PARM to suggest that it is the name of a parameter. Good alternatives might be NAME or PARM_NAME. So now we can write the header.

```
%macro StdParm ( parm = /* name of parameter requiring standardization */  
                ) ;  
  /* Present parm=abc as global variables  
     abc0 - number of selections (legal values 0, 1, 2, etc.)  
     abc1 - first choice for parameter abc
```

⁶ Often it is simpler, and easier to read and write code based on a space (or another symbol) separated list of values instead of an array of variables.

⁷ Any legitimate macro expression can be broken down into a sequence of variable assignments and evaluations which require no more than three consecutive ampersands.

⁸ The %LENGTH test passes if and only if PARM is empty, and the last comparison is passed whenever PARM has just 0 or more blanks.

```

        abc2 - second choice for parameter abc
        etc.
Usage:
    %stdparm(parm=abc)
*;

```

The remainder of the macro is

```

%if %length ( &parm ) = 0 %then
%do ;
    %put ERROR: (STDPARM) Illegal name of parameter ;
    %return ;
%end ;

%global &parm.0 ;
%if %synglobl(&parm) = 0 and %length(&&&parm.0) = 0 %then
    /* &parm is not defined and &&&parm.0 is empty,
       i.e. no choice was made
    */

    %let &parm.0 = 0 ;
%else
%do ;
    %if %length(&&&parm.0) = 0 %then
        /* exactly one choice was made */
        %let &parm.0 = 1 ;
    %global &parm.1 ;
    %let &parm.1 = &&&parm ;
    %symdel &parm ;
%end ;

%mend stdparm ;

```

After a program executes the macro with say, PARM=REG, it is guaranteed to have a global macro variable REG0 giving the number of choices, and ®0 global variables of the form REG1, REG2, etc. up to ®0. Consequently the macro loop

```

%do i = 1 %to &reg0 ;
    /* process &&reg&i */
%end ;

```

is always valid. If an error message is needed when the loop does not execute, then the program can test whether the value of REG0 is equal to 0.

Proof of Concept

To test the macro we use a simple macro REPORT that just loops through the respective choices report them to the log or issues a message that the parameter does not correspond to a global variable. Whether one wants this message or not really, depends on the application. Here is the code⁹.

```

%macro report ( parm ) ;

    %local i ;
    %do i = 1 %to &&&parm.0 ;
        %put &parm&i = >>>&&&parm&i<<< ;
    %end ;
%macroend ;

```

⁹ A positional parameter is used here because the purpose of the macro is simple testing, i.e. it is not part of system, but rather part of the scaffolding.

```

%end ;
%if &&&parm.0 = 0 %then
    %put No global variable for REPORT parameter: &parm ;
%mend report ;

```

Here is the relevant log testing each case for up to the selection of two choices.

```

43          /* parameter does not exist */
44          %stdparm(parm=w) ;
45          %report (w)
No global variable for REPORT parameter: w
46
47          /* parameter exists and is empty */
48          %global x ;
49          %stdparm(parm=x) ;
50          %report (x)
x1 = >>><<<
51
52          /* parameter exists & isn't empty, i.e. single choice selected */
53          %let y = 7 ;
54          %stdparm(parm=y)
55          %report (y)
y1 = >>>7<<<
56
57          /* multiple choices selected */
58          %let z0 = 2 ; %let z = 99 ; %let z1 = 99 ; %let z2 = abc ;
59          %stdparm(parm=z) ;
60          %report (z)
z1 = >>>99<<<
z2 = >>>abc<<<
61
62          %put _user_ ;
GLOBAL X0 1
GLOBAL X1
GLOBAL Y0 1
GLOBAL Y1 7
GLOBAL Z0 2
GLOBAL Z1 99
GLOBAL Z2 abc
GLOBAL W0 0

```

Conclusion

We have illustrated how to work with the name of a macro variable rather than hard coding the variable itself. The technique has many different applications in providing flexibility. The discussion of how to work with macro expressions is detailed enough to allow the technique to become part of any macro programmer's standard techniques.

The example used to illustrate the techniques came from a problem in the way SAS stored processes receive parameters. Consequently the techniques were illustrated with global variables. However, nothing prevents application of the techniques to local variables in the calling environment. In this case, of course, one cannot create new local to the outer environment variables because the macro facility does not provide the tools to do this. In other words, with the exception of the global environment a macro cannot ask another macro to create a local variable in the calling environment.

Contact Information

The author may be contacted via mail using the address

Ian Whitlock
Independent SAS Consultant
29 Lonsdale Lane
Kennett Square, PA

or perhaps better via e-mail at

ian.whitlock@comcast.net

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration

Appendix

In general, it is best to keep the complexity of macro expressions limited to three consecutive ampersands since longer sequences of ampersands are rarely needed, hard to work with and can be eliminated. On the other hand they provide a good exercise for understanding how expressions are resolved. The following example is challenging, but not obviously useful.

Suppose we need to refer to a variable X0Y, not by name, but in terms of other entities, say an array V1, V2 to hold the letters and variables I, J, and K to hold the numbers 0, 1, and 2 respectively. The problem is to refer to the value of X0Y in terms of the root V and the variables I, J, and K.

The problem can be set up with

```
%let v1 = x ;  
%let v2 = y ;  
%let x0y = 99 ;
```

and the problem would then be

```
/* reference x0y in terms of constant text V  
   and variables I=0, J=1, K=2  
*/  
%let i = 0 ;  
%let j = 1 ;  
%let k = 2 ;
```

We could directly solve the problem with

```
%put desired value is &&&&&v&j..&i&&v&k ;
```

However this presents two problems, how did we get the expression in the first place, and what are the chances that we can read and understand the code next week? For practice apply the rules of expression resolution given in the section on how to resolve expressions to see why the above works.

It is somewhat better to introduce a variable NAME to hold the name of the variable we want. Here is code building NAME one part at a time.

```
%* start with the x *;           %let name = &&v&j ;  
%* add the 0 *;                 %let name = &name&i ;  
%* then the y so name is x0y *; %let name = &name&&v&k ;  
%* write the name x0y to the log *; %put &name ;
```

Now the solution is

```
%put name is &name and value is &&&name ;
```

Here the solution is clearer because the steps in the thought process are shown in the code, i.e. the Hansel and Gretel code explains what is happening. Or you might find introducing more variables simpler.

```
%let p1 = &&v&j ;  
%let p2 = &i ;  
%let p3 = &&v&k ;  
%let name = &p1&p2&p3 ;
```

and again

```
%put name is &name and value is &&&name ;
```

Here the strategy is to reduce all parts of the name to same level so that NAME has one simple assignment. Again it has the advantage of leaving a clear trail of what and how the objective was achieved.

So how did I arrive at the expression without introducing any variables? First is looked at

```
%put &&v&j &i &&v&k ;
```

to show that I had the components of the name. Now the second space can be eliminated because the ampersand is a separator ending the reference to I. I cannot simply eliminate the first space because the ampersand in &I will be eaten before the first part is ready for the last scan, so a dot is needed. But that dot will be eaten by the evaluation of &J. Thus two dots are needed - the first to be eaten by &J and the second to end the final scanning of &&V&J → &V1. So now we could have

```
%let name = &&v&j..&i&&v&k ;  
%put name is &name and value is &&&name ;
```

Finally the extra variable NAME can be eliminated by delaying until the expression for NAME is resolved. But that expression takes two scans - one to evaluate &J, &I, and &K, and a second to resolve the resulting &V1 and &V2. So we need to delay twice. That means we need four more ampersands (&&&& → && → &) that will reduce to one for the final evaluation of the expression that represents the name of the variable to be evaluated.

Was it worth it? Probably yes because, if you got this far, you should have a better understanding of complex expressions, not because the problem itself is significant. Remember that it is always better to have more technique available than is usually needed.

In the above work V was hard coded constant text. Suppose everything is held in variables, say

```
%let root = v ;
```

Now for the above problem how do you evaluate XOY in terms of just variables - ROOT, I, J, and K?

Don't look, try to solve it first!

If you got this one, declare yourself an expert and remember to use no more than three consecutive ampersands by introducing extra variables.

Hint:

```
%put &&&&&&&&&&root&j..&i&&&root&k ;
```