Paper 078-2018

This Too Shall Pass:

Passing Simple and Complex Parameters In and Out of Macros

Ted D. Williams, PharmD, BCPS, Magellan Method

ABSTRACT

Even a rudimentary knowledge of SAS® macros will highlight one glaring limitation, macros only accept text input parameters and have no output parameters. How can a SAS programmer develop robust and reusable programs with strong encapsulation and abstraction without the ability to return complex data types (i.e. datasets)? Too often the solution involves compromising encapsulation (i.e. referencing global variables) which severely limits reusability by invariably producing runtime errors and the need for modification. But there is a better way. Two programming techniques, data validation and passing parameters by reference, enable the SAS programmer to create robust and reusable macros with strong encapsulation. This study will demonstrate how these techniques, with minimal additional code, can dramatically increase the flexibility and usability of macros.

INTRODUCTION

Encapsulation is one of the cornerstone principles of model programming.[1] Encapsulation, in part, means creating an interface which is explicit and exclusive through which data is passed in and out of procedures. Encapsulation make code reuse and sharing dramatically more practical and reduces overall development and debugging time. The architecture of modern programming languages (e.g. C#) take concepts like encapsulation for granted. SAS, having been originally developed in 1966, has no such intrinsic assumption. But that does not mean that SAS does not support encapsulation. SAS does support strong encapsulation with the use of macros, albeit with some additional effort. This paper will discuss how to pass simple (text, numbers, etc.) parameters as well as complex parameters (datasets) in and out of macros, rather than rely on global macro variables and datasets. The initial time investment pay off in reduced development time through code reuse without cutting and pasting or making the inevitable coding modifications. The techniques discussed in this paper are intended for all levels of SAS programmers. Novice SAS developers will learn how to write simple macros and pass parameters in and out those macros. Experienced SAS developers will also learn how parameter validation can increase code reuse, especially among developers with different coding styles and naming conventions.

PASSING BY REFERENCE AND PASSING BY VALUE

Any discussion of parameters is predicated on a sound understanding of the difference between passing parameters by reference and passing parameters by value. The difference between passing by reference and passing by value can be illustrated using the real life example of a friend sending a funny picture of a fat cat to you on your smart phone or computer. Some friends will send you the actual picture, which takes up space on your devices memory and will immediatey appear. This is an example of passing by “value.” You received the actual picture of the fat cat. Alternatively, your friend could send you an internet link and described it as “a picture you have to see to believe.” This is an example of passing by “reference.” They didn’t send you the actual picture, just a reference to the picture which you can use to access the picture. This illustration also point to a very important difference between the handling of parameters passed by value or by reference. When values are passed (the actual picture of the fat cat), it is very easy to evaluate if the picture is there and is appropriate. If your friend sends the message, stating it is the fattest cat they have ever seen, but the picture is of a car, you know immediately there is a problem. On the otherhand, when a friend sends a link, we are all (or should be) skeptical of the link. You might check to makes sure the friend actually sent the link and it wasn’t some sort of phishing attack. You might also inspect the link address to make sure it isn’t to an inappropriate website. Likewise, programs dealing with parameters passed by reference also need to exercise additional care to correctly use the parameters. This additional care can be described as parameter validation. Armed with the illustration of the fat cat, these principles can be applied to macros.

PASSING INPUT PARAMETERS BY VALUE

Macro without parameters can be helpful in performing simple, repetitive tasks, such as creating an execution log.

```
%MACRO CreateLog();
    DATA ExecutionLog;
          ATTRIB CalledBy LENGTH = $ 100;
          ATTRIB Message LENGTH = $ 1000;
    RUN;
%MEND;
%CreateLog();
```

Although this macro is sufficient for this simple task, it is quite rigid. Changing the name of the dataset from ExecutionLog or changing the library requires a programming change. Adding a parameter eliminates the need to change the macro every time. In the example below, a parameter (the name of the dataset) will be passed "by value" to the macro.

```
%MACRO CreateLogWithPositionalParameter(DatasetName);

    DATA &DatasetName;
          ATTRIB CalledBy LENGTH = $ 100;
          ATTRIB Message LENGTH = $ 1000;
    RUN;

%MEND;
%CreateLogWithPositionalParameter (MyDS);
%CreateLogWithPositionalParameter (YourDS);
%CreateLogWithPositionalParameter (OurDS);
```

By adding the parameter DatasetName to the %MACRO statement, the same macro can be called 3 times creating three different datasets by simply changing the macro call. The parameter DatasetName now acts as a local variable within CreateLogWithParameter and replaces the hard-coded dataset name in the DATA statement.

DatasetName is defined as a positional parameter.[2] This means that when calling CreateLogWithParameter, there is no need to explicitly state the name of the parameter, the value within the parentheses in the macro call will be assigned to the DatasetName parameter. This is fine for simple macros, but when there are multiple parameters, keyword parameters can be helpful. Keyword parameters require the macro call to explicitly assign the value to the value to the parameter name. Below is the revised macro using a keyword parameter, as well as the revised calls to the macro.

```
%MACRO CreateLogWithKeywordParameter(DatasetName=);

    DATA &DatasetName;
          ATTRIB CalledBy LENGTH = $ 100;
          ATTRIB Message LENGTH = $ 1000;
    RUN;

%MEND;

%CreateLogWithKeywordParameter(DatasetName=HersDS);
%CreateLogWithKeywordParameter(DatasetName=HisDS);
```

The only change to the %MACRO statement was the addition of the equal sign after the parameter name. But note that the calls to the macro now must list the parameter name, the equal sign, then the value. This may seem cumbersome for a simple macro with just one parameter, but as will be demonstrated later, it is invaluable with mulitple parameters, especially when code is being re-used across projects, over time, and across development teams.

VALIDATING PARAMETERS

The macro CreateLogWithKeywordParameter works correctly as long as a valid SAS dataset name is provided for the DatasetName parameter. What if an invalid dataset name is passed into the macro, such as blank value? To prevent a runtime error, the parameter can be validated with some simple code. CreateLogWithValidation illustrates the technique by checking the parameter value. If no dataset name is provide, the text DefaultLogName is used.

```
%MACRO CreateLogWithValidation(DatasetName=);

    %IF %Length(&DatasetName)=0 %THEN
          %LET DatasetName = DefaultLogName;

    DATA &DatasetName;
          ATTRIB CalledBy LENGTH = $ 100;
          ATTRIB Message LENGTH = $ 1000;
    RUN;

%MEND;

%CreateLogWithValidation(DatasetName=Who);
%CreateLogWithValidation(DatasetName=What);
%CreateLogWithValidation(DatasetName=);
```

PASSING OUTPUT PARAMETERS BY VALUE

In certain situations parameters can be returned from SAS macros.[3] There are significant restrictions which limit the statements that can be used.[4] In general, the macro cannot execute PROCs or use comments other than the /**/ type. Therefore, macros best suited for passing values out by value are simple. The example below checks if a dataset parameter was provided, and returns either a 1 or a 0.

```
%MACRO DatasetNameValid(Dataset);
    %LOCAL Result;
    %LET Result=0;
    %IF %LENGTH(&Dataset)>0 %THEN
          %LET Result=1;
&Result %MEND;

%PUT A Blank Dataset name Returns *%DatasetNameValid()*;
%PUT HELLO Dataset name Returns *%DatasetNameValid(HELLO)*;
```

The %MACRO statement has the same structure as previous examples with one positional input parameter. The macro then proceeds to determine if a dataset name is a string with at least one character. Note how the %MEND statement is different than in previous versions. The %MEND statement is proceeded by the local variable &Result. By preceding the %MEND with the variable, the value of the variable is returned as if the macro was a built-in SAS function, as in the example below.

```
%MACRO CheckAndBuild(DatasetName=);

    %IF %DatasetNameValid(&DatasetName)=1 %THEN
    %do;
          DATA &DatasetName;
                ATTRIB CalledBy LENGTH = $ 100;
                ATTRIB Message LENGTH = $ 1000;
          RUN;
    %end;
    %else
    %do;
          %put Cannot create a dataset without a name;
    %end;

%MEND;
%CheckAndBuild(DatasetName=Checkered);
%CheckAndBuild();
```

The first time CheckAndBuild is executed the macro DatasetNameValid returns 1, and the DATA statement creates the dataset Checkered. The second time CheckAndBuild is executed, the DatasetExist macro returns 0, resulting in the %PUT statement being executed, rather that the DATA statement. Macros returning parameters by value have tremendous utility for validating macro parameters as demonstrated in the following sections.

PASSING COMPLEX DATA TYPES INTO MACROS BY REFERENCE

Although checking for missing parameters is helpful, real world programs require more sophisticated validations. For programs to be shared widely across SAS programmers and over time, the macros must be able to accept complex data types. In SAS complex data types are typically datasets. The syntax of passing by reference is the same as passing by value. Only the interpretation of the data changes. In the example above, DatasetNameValid treated the Dataset parameter as a simple string value, making no other assumptions. A more useful function would be to check for the existence of a dataset. The macro ValidateDataset make the assumption that the string value is not simply a text value, but is referring to a complex data type, a dataset.

```
%MACRO DatasetExist(Dataset);
    %LOCAL Result;
    %LET Result=0;
    %IF %LENGTH(&Dataset)=0 %THEN
          %LET Result=0;
    %ELSE
          %LET Result = %SYSFUNC(EXIST(&Dataset));
&Result %MEND;

%MACRO ValidateAndBuild(DatasetName=);

    %IF %DatasetNameValid(&DatasetName)=1 %THEN
    %do;
          %IF %DatasetExist(&DatasetName)=0 %THEN
          %do;
                DATA &DatasetName;
                      ATTRIB CalledBy LENGTH = $ 100;
                      ATTRIB Message LENGTH = $ 1000;
                      STOP;
                RUN;
          %end;
          %else
          %do;
                %put Dataset &DatasetName already exists;
          %end;
    %end;
    %else
    %do;
          %put Cannot create a dataset without a name;
    %end;


%MEND;
%ValidateAndBuild();
%ValidateAndBuild(DatasetName=Validated);
%ValidateAndBuild(DatasetName=Validated);
```

The macro DatasetExist first verifies that a parameter was provided, then treats that parameter as a reference to a dataset, verifying its existence. The first call to ValidateAndBuild does not create the dataset because the DatasetName parameter was not provided. The second call successfully creates the dataset. The third call does not execute the DATA statement, because the dataset already exists. This overly simple DatasetExist macro illustrates the principle of making assumptions about a parameter beyond the intrinsic text value.

The more practical example below shows how this principle can be used to prevent runtime errors by verifying not only the existence of a required dataset, but validating the structure, ensuring all of the expected variables are present.

```
%MACRO VariableExist(Dataset,Variable);
     %LOCAL dsid ;
    %LOCAL result;
    %LOCAL Dummy;

    %IF %length(&Variable)=0 %Then
          %LET Result = 1;
    %ELSE
    %DO;
        %LET dsid = %SYSFUNC(open(&Dataset));
        %IF %SYSFUNC(varnum(&dsid, &Variable)) > 0 %then
            %LET result = 1;
        %ELSE
            %LET result = 0;
        %LET Dummy = %SYSFUNC(close(&dsid));
    %end;
&result %MEND ;


%MACRO AddToLog(DatasetName=,CalledBy=,Message=);

    %IF %DatasetExist(&DatasetName)=1 %THEN
    %do;
                %IF
                      %VariableExist(&DatasetName,CalledBy)=1 AND
                      %VariableExist(&DatasetName,Message)=1
                %THEN
```

```
                  %do;
                        PROC SQL;
                              INSERT INTO &DatasetName
                              VALUES("&CalledBy","&Message")
                              ;
                        QUIT;
                  %end;
                  %ELSE
                  %do;
                        %PUT Dataset &DatasetName has an unexpected format;
                  %end;
      %end;
      %else
      %do;
            %put Dataset &DatasetName does not exists;
      %end;

   %MEND;

   %MACRO LogSome(DatasetName);
      %Local CalledBy;
      %let CalledBy = &SYSMACRONAME;
      %AddToLog(DatasetName=&DatasetName,CalledBy=&CalledBy,Message=HELLO);
      %AddToLog(DatasetName=&DatasetName,CalledBy=&CalledBy,Message=World);
      %AddToLog(DatasetName=&DatasetName,CalledBy=&CalledBy,Message=Here);
      %AddToLog(DatasetName=&DatasetName,CalledBy=&CalledBy,Message=I);
      %AddToLog(DatasetName=&DatasetName,CalledBy=&CalledBy,Message=am);
   %MEND;

   %MACRO LogSomeMore(DatasetName);
      %Local CalledBy;
      %let CalledBy = &SYSMACRONAME;
      %AddToLog(DatasetName=&DatasetName,CalledBy=&CalledBy,Message=Other);
      %AddToLog(DatasetName=&DatasetName,CalledBy=&CalledBy,Message=Macro);
   %MEND;
   %ValidateAndBuild(DatasetName=ThisLog);
   %LogSome(ThisLog);
   %LogSomeMore(ThisLog);
```

In this example, the macro AddToLog has 2 parameters passed by value (Message and CalledBy) as one parameter passed by reference (DatasetName). After validating the value parameters, the Macro AddToLog verifies that the LogDataset exists and is in the expected format. Only then does the macro attempt to add an observation. The macros LogSome and LogSomeMore add a few observations, demonstrating how the macro can be reused for multiple messages with a single line of code.

RETURNING COMPLEX DATA TYPES FROM MACROS

Returning complex datatypes is possible, but requires a non-intuitive approach. Returning simple values from a macro is achieved by including a variable reference before the %MEND statement. As discussed previously, these macros are limited in the types of operations allowed. Therefore, the opportunities to create a macro which would need to return a complex data type which is also capable of creating a complex data type are limited. To work around this limitation, macro reference parameters can be passed into the macro, with the expectation that after the macro completes, they will reference a dataset.

```
%MACRO SummarizeLog(LogName=,SummaryName=);
    %IF %DatasetExist(&LogName)=1 %THEN
    %do;
          %IF %DatasetNameValid(&SummaryName) =1 %THEN
          %do;
                %IF %VariableExist(&LogName,CalledBy) =1 %THEN
                %do;
                      PROC FREQ DATA=&LogName;
                            TABLES CalledBy / NOPRINT OUT =&SummaryName;
                      RUN;
                %end;
          %end;
          %else
          %do;
                %put Parameter SummaryName was not a valid dataset name;
          %end;
    %end;
    %else
    %do;
          %put Log &LogName cannot be found;
    %end;
%MEND;

%Macro FillTheLog(DatasetName);
    %ValidateAndBuild(DatasetName=&DatasetName);
    %LogSome(&DatasetName);
    %LogSomeMore(&DatasetName);
    %SummarizeLog(LogName=&DatasetName,SummaryName=MySummary);
    %IF %DatasetExist(MySummary)=1 %THEN
    %do;
          PROC PRINT DATA=MySummary;
          RUN;
    %end;
    %else
    %do;
          %PUT MySummary was not created correctly;
    %end;
%mend;

%FillTheLog(ThisHereLog);
```

The macro SummarizeLog has two parameters LogName (input parameter) and SummaryName (output parameter) both referencing datasets. The LogName is a data which exists prior to execution and SummaryName is created during execution. Note how the macro FillTheLog validates the returned by reference parameter MySummary prior to executing PROC PRINT.

REAPING THE REWARDS

Writing these helper macros and checking parameters is an investment which pays off in both development time and runtime. The example below uses parameters and validation to verify the state of the SAS system prior to execution. The macro FearTheReaper has no parameter validation and would execute multiple long running macros (LRM and VLRM) before failing due to a runtime error. DontFearTheReaper has parameter validation causing the macro exits immediately, allowing for human intervention and correction without waiting for hours to discover the problem.

```
%Macro LRM(DatasetName);
    %PUT I am a long running macro LRM;
    PROC PRINT DATA = &DatasetName;
    RUN;
%mend;

%Macro VLRM(DatasetName);
    %PUT I am a very long running macro VLRM;
    PROC PRINT DATA = &DatasetName;
    RUN;
%mend;

%Macro FearTheReaper (Dataset1, Dataset2, Dataset3,Dataset4);

    %LRM(&Dataset1);
    %VLRM(&Dataset2);
    %VLRM(&Dataset3);
    %LRM(&Dataset4);

%MEND;
%Macro DontFearTheReaper (Dataset1, Dataset2, Dataset3,Dataset4);
    %IF
          %DatasetExist(&Dataset1)=1 AND
          %DatasetExist(&Dataset2)=1 AND
          %DatasetExist(&Dataset3)=1 AND
          %DatasetExist(&Dataset4)=1
    %THEN
    %DO;
          %LRM(&Dataset1);
          %VLRM(&Dataset2);
          %VLRM(&Dataset3);
          %LRM(&Dataset4);
    %END;
    %ELSE
    %DO;
          %PUT A required dataset was missing, so I stopped before I
started;
    %END;
%MEND;

%FearTheReaper(ThisLog,MyDS,YourDS,TypoDataset);
%DontFearTheReaper(ThisLog,MyDS,YourDS,TypoDataset);
```

CONCLUSION

Passing parameters into macros requires investing a few dozen keystrokes to save hours of runtime, hours of debugging time, and hours of code modification. Although passing parameters into macros is not fundamental to SAS programing, macros do support passing parameters by reference or by value. Code reuse can be greatly increased by replacing hard coded names of datasets inside macros with parameter variables representing the dataset name (passing by reference). Validation of parameters can be encapsulated into reusable macros which return values, much like built-in SAS functions. These macros make validation of parameters at the beginning of the macro a one line operation. These small investments and modifications to the style of programming greatly increase code reuse, reducing develop and debugging time. Who doesn't need more time?

REFERENCES

1. Nakov S. Introduction to Programming with C# / Java Books » Chapter 20. Object-Oriented Programming Principles (OOP). http://www.introprogramming.info/, http://www.introprogramming.info/english-intro-csharp-book/read-online/chapter-20-object-oriented-programming-principles/. Accessed July 11, 2018.

2. Macro Statements: %MACRO Statement. http://support.sas.com/documentation/cdl/en/mcrolref/61885/HTML/default/viewer.htm#macro-stmt.htm. Accessed July 11, 2018.

3. Pine D. Using SAS® Macro Language to Develop User-Written Functions. In: *SESUG 2003*. ; 2003. https://analytics.ncsu.edu/sesug/2003/CC13-Pine.pdf. Accessed July 11, 2018.

4. Foster E. Create your own Functions using SAS/MACRO and SCL. In: *Pharmaceutical Users Software Exchange*. ; 2006. https://www.lexjansen.com/phuse/2006/cs/CS06.pdf. Accessed July 11, 2018.

ACKNOWLEDGMENTS

I would like to thank Michael Polson of Magellan Method for perennially supporting me optimize my SAS programs. Anne Kangethe, Donna Moore and Servi Barrientos provided invaluable assistance in editing and vetting this paper. Thanks to Art Carpenter for his mentoring and sage advice in preparing this paper. Finally, I would like to thank the volunteers of Western Users of SAS Software (WUSS) for the opportunity to present this paper.

RECOMMENDED READING

CONTACT INFORMATION

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

Ted D. Williams
Magellan Method
88 Silva Lane
Tech 4
Middletown, RI 02842
314-387-4305
tdwilliams1@magellanhealth.com