

Call Execute: Let Your Program Run Your Macro

Artur Usov, OCS Consulting BV, 's-Hertogenbosch, Netherlands

ABSTRACT

The SAS Macro language is an extremely flexible tool which facilitates SAS programmers with the ability to re-run the same code for a set of different parameters. As the number of required re-runs increases one could use a do-loop to rerun the same macro, passing macro parameters from a list of parameter values stored in a global macro variable. That can, however, become cumbersome as the list of parameters and macro calls becomes large. Instead, Call Execute allows storing all macro parameter values in a SAS dataset and using those values in a data step to execute the macro. Or, the other way around, it allows using the values in a SAS dataset as parameters to a macro. It will also allow you to create programs that are completely controlled by the data that it processes. This paper will demonstrate how the SAS Macro language combined with Call Execute statements in a data step can help the execution of macros with macro parameter values obtained from a SAS dataset.

INTRODUCTION

The SAS Macro language is a very flexible extension of SAS Base; it allows creation of generic code in a macro which can be re-executed for different scenarios by passing different parameters. Usage of SAS Macro facility is very efficient and can save a lot of code space, reduce the number of errors, and simply make the code more elegant.

As the number of macro parameters and macro executions increase, simple macro calls in the program can become inefficient, especially if parameter values are subject to change. One way to go around it is to use a do-loop to rerun the same macro, passing macro parameters from a list of parameter values stored in a global macro variable. That can, however, become cumbersome and prone to errors as the list of parameters and macro calls becomes large.

Call Execute routine allows an elegant solution by storing all macro parameter values in a SAS dataset and using those values in a data step to execute the macro. It will also allow you to create programs that are completely controlled by the data that it processes.

This paper will familiarise the reader with Call Execute by demonstrating the syntax with application of simple examples. Further it will demonstrate how the SAS Macro language combined with Call Execute statements in a data step can help the execution of macros with macro parameter values obtained from a SAS dataset or user input.

SYNTAX

Call Execute is a facility of the DATA step which allows executing SAS code generated by the DATA step. Also, the data from the DATA step can be used as part of the executable code in the Call Execute.

The syntax of the Call Execute routine is rather simple:

```
call execute('argument');
```

Here, *argument* can be any SAS code, enclosed in single quotes, which has to be executed. It can also contain variables from the DATA step in which the Call Execute code is invoked. All arguments passed to the Call Execute statement are executed as regular SAS code directly after it finishes interpreting the Call Execute statement.

The simple example below executes a %PUT statement and writes "Hello World" to the log. Since this DATA step has no input dataset (there is no SET statement) this Call Execute code will be executed only once.

Submitted code:

```
data _null_;  
  call execute('%put Hello world;');  
run;
```

Code generated and executed by Call Execute:

```
%put Hello world;
```

Output in log:

```
Hello world
```

INCLUDING VALUES FROM A DATASET

Call Execute becomes especially powerful when you start including values from a dataset in the arguments. The following example will put a list of names of student from the SASHELP.CLASS in the log:

Submitted code:

```
data _null_;  
  set sashelp.class;  
  call execute('%put Students name is '||strip(name)||'; ');  
run;
```

Code generated and executed by Call Execute:

```
%put Students name is Alfred;  
%put Students name is Alice;  
%put Students name is Barbara;  
<...et cetera>
```

Output in log:

```
Students name is Alfred  
Students name is Alice  
Students name is Barbara
```

Here, the dataset SASHELP.CLASS is read with a SET statement. Within this DATA step a Call Execute will execute the PUT statement for every observation read from the SASHELP.CLASS dataset. The code in the Call Execute which is not within the quotation marks (i.e. the strip(name)) is going to be executed first and concatenated with the rest of the Call Execute code. In this example, it will obtain names from the NAME variable of the SASHELP.CLASS dataset, apply a strip function and concatenate the NAME variable values with the %PUT "Students name is" code. You use the double pipe for concatenating data from the data step with the string which will be executed, or use one of the string concatenation functions such as CAT or CATS. This process will be repeated for each observation of the SASHELP.CLASS dataset.

It is also possible to generate and execute a full data step within a data step using Call Execute. The code below will create one data set for each student name in the SASHELP.CLASS. First, the SET statement reads in the observations from the dataset SASHELP.CLASS. Once SAS reaches the Call Execute statement it would first resolve the code which is not within quotation marks, namely execute the strip functions while obtaining NAME value from the name variable, and then concatenate it to rest of the rest of the Call Execute code. That process will be repeated for every observation from the SASHELP.CLASS.

Submitted code:

```
data _null_;
  set sashelp.class;
  call execute('data work.'||strip(name)||';
               set sashelp.class;
               where name="'||strip(name)||'";
               run;');
run;
```

Code generated and executed by Call Execute:

```
First observation with name Alfred

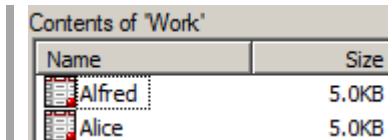
data work.Alfred;
  set sashelp.class;
  where name="Alfred";
run;

Second observation with name Alice

data work.Alice;
  set sashelp.class;
  where name="Alice";
run;

<...et cetera>
```

Datasets created:



Name	Size
Alfred	5.0KB
Alice	5.0KB

MACRO INSTEAD OF DATA STEP

In the previous example a full DATA Step was executed with Call Execute which created one dataset per student name read from SASHELP.CLASS. As complexity and size of code which has to be executed becomes larger it becomes difficult not only to execute this code within a data step but also to validate and maintain it. Macro language facility with Call Execute could be used instead by executing macros while passing macro parameters from a dataset.

Consider the above example where one wants to create a dataset for each student from the SASHELP.CLASS dataset. Instead of generating the full code within a dataset, a separate macro DATASETS is created for this purpose.

The macro DATASETS is used for the executing a DATA step which creates a dataset per student name. The macro parameter NAME is the student's name which is passed to the macro.

To execute the macro for each name a Call Execute is used inside the DATA _NULL_ step. The data step first reads observations from the SASHELP.CLASS with a SET statement. Once SAS reaches the Call Execute statement it uses the values inside that dataset to generate lines of code that include a call to the macro %datasets, like "%datasets(name=Alfred);". This process is repeated for each observation read from the SASHELP.CLASS.

This example produces the same output as the one with the code executed with the data step but with a much more elegant way which is easier to maintain and validate. Having the program logic, which creates the named datasets,

inside a macro, allows the programmer to execute the logic like regular SAS code, which allows for much easier debugging.

The following section would demonstrate a use of Call Execute in a macro with multiple parameters.

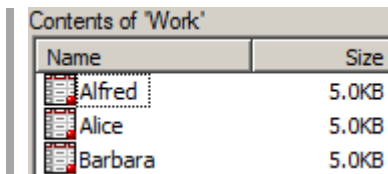
Submitted code:

```
%macro datasets(name= );  
    data work.&name;  
        set sashelp.class;  
        where name="&name";  
    run;  
  
%mend datasets;  
  
data _null_;  
    set sashelp.class;  
    call execute('%datasets(name='||strip(name)||')');  
run;
```

Code generated and executed by Call Execute:

```
%datasets(name=Alfred); /* Generated from the 1st record. */  
%datasets(name=Alice); /* Generated from the 2nd record. */  
%datasets(name=Barbara); /* Generated from the 3rd record. */
```

Datasets created:



Name	Size
Alfred	5.0KB
Alice	5.0KB
Barbara	5.0KB

MACRO WITH MULTIPLE PARAMETERS

As the number of parameters in the macro increases, it becomes more inconvenient to use do loops or large blocks of Call Execute code within a data step for automated re-execution of the macro.

Consider an example where one wants to produce descriptive statistics for all character and numeric variables located in the specified library. For this example SASHELP.CLASS will be used as a data source.

The REPORT macro is used for calculating descriptive statistics for numeric using PROC MEANS and for character variables using PROC FREQ. Here VAR is a variable name, and TYPE is the type of the variable.

First, a list of variables and datasets can be obtained from DICTIONARY.COLUMNS with a PROC SQL. It will save the list of variables in dataset called VARS in the WORK library.

Then a Call Execute within a DATA _NULL_ step is used for passing the parameters to the macro and executing it. The DATA step first reads observations from the WORK.VARS with a SET statement. For each record in the input dataset it generates a call to the macro %report, which in turn interprets the parameters and calls the required statistical procedure.

Submitted code:

```
proc sql;
  create table work.Vars as
  select name,type
  from dictionary.columns
  where memname="CLASS" and libname="SASHELP";
quit;

%macro report(var= , type= );

  %if &type=char %then %do;
    proc freq data=sashelp.class;
      table &var;
    run;
  %end;

  %else %do;
    proc means data=sashelp.class;
      var &var;
    run;
  %end;

%mend report;

data _null_;
  set work.Vars;
  call execute('%report(var='||strip(name)||' , type='||strip(type)||');');
run;
```

Code generated and executed by Call Execute:

```
%report(var=Age , type=num);
%report(var=Sex , type=char);
%report(var=Age , type=num);
%report(var=Height , type=num);
%report(var=Weight , type=num);
```

Output:

The MEANS Procedure				
Analysis Variable : Age				
N	Mean	Std Dev	Minimum	Maximum
19	13.3157895	1.4926722	11.0000000	16.0000000

The strength of this approach is in the fact that you can have a very short DATA step that generates tens or hundreds of calls to the macro, without having to write any additional code. This automated process is then independent of the content or structure of the input dataset, and should the input dataset change, the program will adapt automatically.

MACRO WITH USER INPUT

There are cases where parameters which are passed to macro have to be defined by the user executing the SAS program. In such cases Excel can be used as a user interface to provide the macro inputs.

Consider an example where one is asked to perform a specific test for a specific variable. The list of test and variables is provided by the user and is subject to changes.

For this purpose Excel can be used as a user interface where the user would provide variable name, dataset, library name, and test name. An example of such Excel file is provided below.

PhUSE 2014

libname	ds	var	test	class
sashelp	class	age	ttest	sex
sashelp	cars	type	chisq	origin

A macro TESTS is used for performing statistical test for the specified variables. Here LIBNAME is the library name, DS is the dataset name, VAR is the variable name, TEST is the test name, and CLASS is the grouping variable used in the procedure.

Using PROC IMPORT we can import the provided Excel data. Alternatively one might export the Excel file to a CSV file and read that using the *infile* statement, should the appropriate license to read Excel files not be available. Once this data is imported and stored in the SAS dataset, Call Execute is used within the data step to execute the TESTS macro. SAS will execute the Call Execute in the same manner as in previous examples.

Submitted code:

```
proc import out=work.tests
    datafile="C:\vars.xlsx"
    dbms=xlsx replace;
    getnames=yes;
run;

%macro tests(libname=, ds=, var=, test=, class=);

    %if &test=ttest %then %do;
        proc ttest data=&libname..&ds;
            class &class.;
            var &var;
            run;
        %end;

    %else %if &test=chisq %then %do;
        proc freq data=&libname..&ds;
            table &class * &var/chisq;
            run;
        %end;

%mend tests;

data _null_;
    set work.tests;
    call execute('%tests(libname='||strip(libname)||',
                        ds='||strip(ds)||',
                        var='||strip(var)||',
                        test='||strip(test)||',
                        class='||strip(class)||');');
run;
```

Code executed by Call Execute:

```
%tests(libname=sashelp,ds=class,var=age,test=ttest, class=sex);
%tests(libname=sashelp,ds=cars,var=type,test=chisq, class=origin);
```

Partial output:

The TTEST Procedure						
Variable: Age						
Sex	N	Mean	Std Dev	Std Err	Minimum	Maximum
F	9	13.2222	1.3944	0.4648	11.0000	15.0000
M	10	13.4000	1.6465	0.5207	11.0000	16.0000
Diff (1-2)		-0.1778	1.5331	0.7044		

The FREQ Procedure			
Table of Origin by Type			
Statistics for Table of Origin by Type			
Statistic	DF	Value	Prob
Chi-Square	10	35.6659	<.0001
Likelihood Ratio Chi-Square	10	42.1254	<.0001
Mantel-Haenszel Chi-Square	1	0.0808	0.7762
Phi Coefficient		0.2887	
Contingency Coefficient		0.2773	
Cramer's V		0.2041	

Sample Size = 428

LIMITATIONS

Call Execute has a limitation when executing macro code. When Call Execute is used to call a macro, and inside that macro there is a PROC SQL with INTO, or a CALL SYMPUT, to create a new macro variable, then the creation of this macro variable is delayed until the macro finished execution. As a result, the macro variable cannot be used inside the same macro. Since Call Execute is usually used to perform multiple calls to the same macro this problem can easily be overlooked because the *second* run of the macro will use the macro variable value from the *first* run, the *third* run will use the *second* macro, et cetera.

This problem can easily be avoided by applying the %NRSTR function around the argument to Call Execute. A side effect of this function is that it will change the timing of the execution of the code from immediately after the Call Execute statement to the end of the DATA step.

Consider a macro %writeAge which creates a macro variable and writes its value in the log using a PUT macro statement. The code below demonstrates the results of executing the macro with and without %NRSTR function.

```
/* This macro will use PROC SQL with INTO to
   obtain the maximum AGE in SASHELP.CLASS. */
%macro writeAge;

  proc sql noprint;
    select max(age)
      into :age
    from sashelp.class;
  quit;

  %put The age inside the macro is: &age;

%mend writeAge;

/* Prior to using the Call Execute, assign a
   value to AGE. */
%let age=Initial value;
```

```
/* Run the macro. */  
data _null_;  
  call execute('%writeAge');  
run;  
  
/* Show the value of AGE after all processing  
   has been done. */  
%put The age after all processing: &age;
```

The result in the log, after removing some standard notes written by SAS:

```
The age inside the macro is: Initial value  
  
The age after all processing:          16
```

You will see that while inside the macro, the program puts the initial value of the macro variable in the log, and not the value that is expected to be generated by the PROC SQL statement. However, after the macro finished, the macro variable does contain that generated value.

By including the %NRSTR function, like below, in the Call Execute statement, this problem can be resolved.

```
/* Run the macro. */  
data _null_;  
  call execute('%nrstr(%writeAge)');  
run;
```

The result then becomes:

```
The age inside the macro is:          16  
  
The age after all processing:          16
```

CONCLUSION

The usage of the SAS macro facility is very efficient and can save a lot of code space, reduce number of errors and simply make the code more elegant. Call Execute offers an elegant solution to automate the execution of macros. By using Call Execute to call macros instead of having it generate the code directly inside the data step your data driven programs suddenly become as easy to maintain and interpret as any other program.

REFERENCES

Ruelle, A., & Moses, K. (2006). CALL EXECUTE: A Primer. Retrieved from <http://www.lexjansen.com/pharmasug/2006/technicaltechniques/tt16.pdf>

Liu, L. (2007). Passing Data Set Values into Application Parameters. Retrieved from <http://www.mwsug.org/proceedings/2007/appdev/MWSUG-2007-A02.pdf>

Sharma, R. (2008). CALL EXECUTE: A Hidden Treasure and Powerful Function. Retrieved from <http://www.lexjansen.com/pharmasug/2008/po/PO11.pdf>

Boisvert, D., & Chowdhury, S. (2006). CALL EXECUTE for everyone! Examples for programmer, statistician, and data manager. Retrieved from <http://www.lexjansen.com/pharmasug/2006/TechnicalTechniques/TT05.pdf>

ACKNOWLEDGMENTS

I would like to thank Jules Van der Zalm for support and guidance.

CONTACT INFORMATION

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

Artur Usov
OCS Consulting
P.O. Box 3434
's-Hertogenbosch
+31 (0)73 523 6000
sasquestions@ocs-consulting.com
<http://www.ocs-consulting.nl/>

Brand and product names are trademarks of their respective companies.