# SAS® Macro Dynamics: from Simple Basics to Powerful Invocations

Rick Andrews, Office of Research, Development, and Information, Baltimore, MD

## ABSTRACT

The SAS Macro Facility offers a mechanism for expanding and customizing the functionality of the SAS System. It allows for the abbreviation of a large amount of program code conveniently and makes textual substitutions easy. The facility contains a programming language that will enable the execution of small sections of a program or entire steps conditionally. This paper assumes a basic knowledge of the DATA STEP and the use of programming logic and will provide simple to dynamics views of the powerful capabilities of SAS macros.

## INTRODUCTION

SAS macros are evaluated before compile time. When a SAS program is submitted the system routes the code to the macro processor and reviews the text to see if any SAS macros or macro variables have been included. The processor cycles through the macro calls, replaces any macro references with actual values, and releases the updated program to the compiler for further processing. The resulting "effective code" will be revealed throughout the paper.

In general, the SAS macro language is portable across all operating systems with few exceptions. Typically, SAS statements that begin with a percent sign (%) are part of the macro language and macro variables can be identified by a proceeding ampersand (&).

## CONTENTS



## MACRO VARIABLES

The simplest use of the SAS Macro Facility is the creation of macro variables. Macro variables should be used if there are multiple instances of the same value that need to be changed each time a program is submitted. Macro variables located at the top of your program centrally locate any necessary changes required by each execution. There are multiple ways in which to create a macro variable, the easiest of which is the use of %LET.

### %LET

Without Macro Variable

```
DATA TEMP1;
  SET TABLE1;
  WHERE CPT_CODE = '21081';
RUN;

DATA TEMP2;
  SET TABLE2;
  WHERE CPT_CODE = '21081';
RUN;
```

With Macro Variable

```
%LET CPTCODE = '21081';

DATA TEMP1;
  SET TABLE1;
  WHERE CPT_CODE = &CPTCODE;
RUN;

DATA TEMP2;
  SET TABLE2;
  WHERE CPT_CODE = &CPTCODE;
RUN;
```

Notice the use of single (') quotes surrounding the character value in the **CPTCODE** macro variable in the example above. When resolved, the value of **'23455'** will be passed including the quotes. If the single quotes are left out during the creation of the macro variable and instead are included in the **WHERE** clause, such as in the example BAD below left, the SAS system will pass the literal value of **'&CPTCODE'** NOT **'23455'**. To resolve the macro variable in this manor it should be surrounded with double (") quotes as shown in the GOOD example below right.

BAD

```
%LET CPTCODE = 21081;

DATA TEMP1;
  SET TABLE1;
  WHERE CPT_CODE = '&CPTCODE';
RUN;
```

GOOD

```
%LET CPTCODE = 21081;

DATA TEMP1;
  SET TABLE1;
  WHERE CPT_CODE = "&CPTCODE";
RUN;
```

## PROC SQL

Another way to create a macro variable is to use the SQL procedure. This is a good way to use values from within a DATA SET that are needed for another process. In the DATA SET, **TABLE1**, a given value for the **MEMBERS** variable can be placed into a macro variable depending on the requirements of the particular process being evaluated.

In this example, the PROC SQL will grab the **FIRST** record in the table. The macro variable **MACVAR1** will resolve to 11000.

In this example, the PROC SQL will grab the record meeting the **WHERE** criteria. The variable **MACVAR2** will resolve to 12000.

```
PROC SQL NOPRINT;
  SELECT MEMBERS
  INTO : MACVAR1
  FROM TABLE1;
QUIT;


%PUT &MACVAR1;
*Result = 11000;
```

MACVAR1

**TABLE1**

| YEAR | MEMBERS |
|---|---|
| 1999 | 11000 |
| 2000 | 12000 |
| 2001 | 13000 |

```
PROC SQL NOPRINT;
  SELECT MEMBERS
  INTO : MACVAR2
  FROM TABLE1
  WHERE YEAR = 2000;
QUIT;

%PUT &MACVAR2;
*Result = 12000;
```

MACVAR2

## CALL SYMPUT

The CALL SYMPUT can be used to interact with the macro facility during the execution of a DATA step. Because macros are evaluated before compilation, macro information has already been processed. The CALL SYMPUT allows you to create macro variables based on values within a table similar to the PROC SQL example shown above. Note a major difference is that one process will resolve to the value of the **LAST** record in lieu of the **FIRST**!

In this example, the CALL SYMPUT will grab the **LAST** record in the table. The macro variable **MACVAR3** will resolve to 13000.

In this example, the CALL SYMPUT will grab the record meeting the **WHERE** criteria. The variable **MACVAR4** will resolve to 12000.

```
DATA _NULL_;
  SET TABLE1;
  CALL SYMPUT('MACVAR3',
              MEMBERS);
RUN;


%PUT &MACVAR3;
*Result = 13000;
```

MACVAR3

**TABLE1**

| YEAR | MEMBERS |
|---|---|
| 1999 | 11000 |
| 2000 | 12000 |
| 2001 | 13000 |

```
DATA _NULL_;
  SET TABLE1;
  WHERE YEAR = 2000;
  CALL SYMPUT('MACVAR4',
              MEMBERS);
RUN;

%PUT &MACVAR4;
*Result = 12000;
```

MACVAR4

### DYNAMIC IN (LIST)

PROC SQL can also be used to string all of the values together. This is particularly useful if an IN list is needed for use within a WHERE clause. Consider the data set, **TABLE2**, where a list of CPT codes has been stored. This list can be placed into a macro variable and used as such: **WHERE CPT_CODE IN ( &MYLIST )**.

**TABLE2**

| CPT_CODE |
|---|
| 21081 |
| 21082 |
| 21083 |
| 21084 |
| 21085 |

```
PROC SQL NOPRINT;
  SELECT "'" || CPT_CODE || "'"
  INTO : MYLIST
  SEPARATED BY ','
  FROM TABLE2;
QUIT;


%PUT &MYLIST;
*RESULT = '21081','21082','21083','21084','21085';
```

MYLIST

## SIMPLE MACROS

Even the most simple of macros have various advantages. 1) They reduce programming code. 2) Eliminate repetitive changes. 3) Provide conditional execution. 4) Reduce typing errors. 5) Allow utilization of the macro language itself.

### %MACRO

This simple example of a SAS macro actually does nothing more than execute the DATA STEP.

```
%MACRO EXAMPLE1;

  DATA TEMP1;
     SET TABLE1;
  RUN;

%MEND EXAMPLE1;

%EXAMPLE1;
```

Program Code: EXAMPLE1

This example demonstrates how to pass values into the macro for use within the DATA STEP.

```
%MACRO EXAMPLE2(MACVAR,CPTCODE);

  DATA TEMP&MACVAR;
    SET TABLE&MACVAR;
    WHERE CPT_CODE = "&CPTCODE";
  RUN;

%MEND EXAMPLE2;

%EXAMPLE2 ( 1, 21081 );
%EXAMPLE2 ( 2, 21082 );
```

Program Code: EXAMPLE2

In order to signify the beginning of a SAS macro the keyword, **%MACRO**, tells the processor the word that follows, such as **EXAMPLE1**, will be the reference used to CALL the macro later in the program. The processor keeps reading the program until it reaches the keyword, **%MEND**, at which point it knows the macro creation has ended. The syntax at right demonstrates the "effective code" created after the program has passed through the macro processor. This is the code the SAS compiler will see. Note the difference between the DATA SET names and the values passed into the WHERE clause.

```
DATA TEMP1;
 SET TABLE1;
 WHERE CPT_CODE="21081";
RUN;
```

```
DATA TEMP2;
 SET TABLE2;
 WHERE CPT_CODE="21082";
RUN;
```

Effective Code: EXAMPLE2

### %IF - %THEN - %ELSE

The macro in **EXAMPLE1** shown above may seem superfluous for there is no need for the macro given that no values are being passed and nothing has changed. One useful purpose for creating a macro with no parameters is to conditionally execute program code. The next example shows how to check whether a file already exists before trying to create it. Checking for the existence of a file can be done for SAS data sets, libraries, directories, and external files as shown.

### OPEN CODE

A statement submitted using the macro language instructs the SAS macro processor to evaluate a given operation. Some statements are only allowed inside of a macro definition. Essentially between the **%MACRO** and **%MEND** statements. There are some instances when parts of the macro language can be submitted in OPEN CODE, which essentially means outside of the macro definition. Examples of this functionality include, though are not limited to, **%LET** and **%PUT**. The **%IF** and **%DO** in the example shown here cannot be executed in OPEN CODE.

```
%MACRO EXAMPLE3;

  DATA _NULL_;
    CALL SYMPUT( 'EXISTS', OPEN('TEMP1','I') );
  RUN;

  %IF NOT &EXISTS %THEN %DO;
      DATA TEMP1;
        SET TABLE1;
      RUN;
  %END;
  %ELSE %DO;
      %PUT ERROR WILL ROGERS!;
  %END;

%MEND EXAMPLE3;
%EXAMPLE3;
```

OPEN - Opens SAS data set.

See also:

FOPEN – Opens external file.
DOPEN – Opens a directory.
FEXIST – Determine if external file exists.

Program Code: EXAMPLE3

## DYNAMIC MACROS

Creating a dynamic macro call based on values within a data set is relatively straight forward. The example below will demonstrate how use a list of CPT codes from a DATA SET and create a macro that will cycle for each instance. The macro **EXAMPLE4** will accept two positional parameters, **MACVAR** and **CPTCODE**, which are used in the name of the temporary SAS data sets and to populate the **GEE WHIZ** variables.

**TABLE2**

| CPT_CODE |
|---|
| 21081 |
| 21082 |
| 21083 |
| 21084 |
| 21085 |

```
*-----------------------------------------*;
* CREATE A MACRO TO PERFORM A GIVEN TASK   *;
*-----------------------------------------*;
%MACRO EXAMPLE4 ( MACVAR , CPTCODE );

  DATA TEMP&MACVAR;
    GEE  = "&CPTCODE";
    WHIZ =  &MACVAR;
  RUN;

%MEND EXAMPLE4;

*-----------------------------------------*;
* CREATE OUTPUT FILE CONTAINING MACRO CALL *;
*-----------------------------------------*;
DATA _NULL_;
  SET TABLE2;
  FILE 'D:\TEMP\EXAMPLE4.SAS';
  MAC_VAR + 1;
  PUT '%EXAMPLE4(' MAC_VAR ',' CPT_CODE ');' ;
RUN;

*-----------------------------------------*;
* INCLUDE FILE CONTAINING MACRO CALL       *;
*-----------------------------------------*;
%INCLUDE 'D:\TEMP\EXAMPLE4.SAS';
```

Program Code: EXAMPLE4

The function of the **DATA _NULL_** step in this example is to output the SAS program **EXAMPLE4.SAS** using the **FILE** statement. The **MAC_VAR** variable is used to count the number of observations to be processed, in this instance, the count of **CPT_CODE** values within **TABLE2**. The two variables are used within the **PUT** statement to create the appropriate number of SAS macro calls based on the information within the DATA SET.

### %INCLUDE

When the **%INCLUDE** statement is executed the syntax of the SAS program **EXAMPLE4.SAS** is incorporated by the macro processor and the SAS macro calls are evaluated. The subsequent "effective code" is passed to the compiler. At execution the system will process the following data steps, 5 in this case. If the number of records in **TABLE2** changes, the macro will dynamically execute as many calls as necessary.

```
%EXAMPLE4 ( 1 , 21081 );
%EXAMPLE4 ( 2 , 21082 );
%EXAMPLE4 ( 3 , 21083 );
%EXAMPLE4 ( 4 , 21084 );
%EXAMPLE4 ( 5 , 21085 );
```

SAS Program: EXAMPLE4.SAS

```
DATA TEMP1;
  GEE="21081";
  WHIZ=1;
RUN;
```

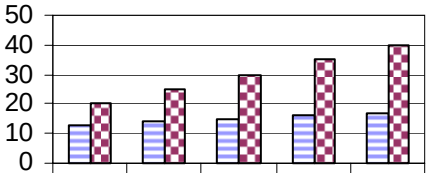
```
DATA TEMP2;
  GEE="21082";
  WHIZ=2;
RUN;
```

```
DATA TEMP3;
  GEE="21083";
  WHIZ=3;
RUN;
```

```
DATA TEMP4
  GEE="21084;
  WHIZ=4
RUN;
```

```
DATA TEMP5;
  GEE="21085";
  WHIZ=5;
RUN;
```

Effective Code: Example4



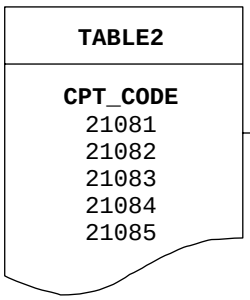

| | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|
| WITH | 13 | 14 | 15 | 16 | 17 |
| WITHOUT | 20 | 25 | 30 | 35 | 40 |

Program Code Reduction

The advantage of SAS macros for reducing the amount of programming code can now begin to be seen as well as the reduction in the potential for typing errors and elimination of repetitive changes. In this example, there are 13 lines of actual programming code in **EXAMPLE4**. The amount of effective code is 20 lines. If another CPT code needed to be extracted the number of lines for the macro would go up by 1, whereas the number of lines of effective code would go up by 4. It is easy to see how a macro program can save an untold amount of coding.

### %DO-%TO-%END

The program that follows will produce the same "effective code" as in **EXAMPLE4**. The result being five DATA STEPS passed to the SAS compiler for processing. The major difference being the external file is not required whereas in the previous step the program ***'D:\TEMP\EXAMPLE4.SAS'***, was created and subsequently included. Here the process will dynamically **LOOP** through the data without the need to the creation of the external file.




```
*-----------------------------------------*;
* CREATE A MACRO TO PERFORM A GIVEN TASK   *;
*-----------------------------------------*;
%MACRO EXAMPLE5;

  DATA TEMP&I;
    GEE  = "&&MACVAR&I";  *<-- double ampersand;
    WHIZ = &I;
  RUN;

%MEND EXAMPLE5;

*-----------------------------------------*;
* DEFINE MACRO VARIABLES                  *;
*-----------------------------------------*;
DATA _NULL_;
  SET TABLE2 END=EOF;
  CALL SYMPUT( 'MACVAR' || LEFT(TRIM(_N_)) , CPT_CODE );
  IF EOF THEN
  CALL SYMPUT( 'TOTOBS' , _N_ );
RUN;

*-----------------------------------------*;
* SUBMIT EXAMPLE5 FOR EACH CPT CODE       *;
*-----------------------------------------*;
%MACRO LOOP;
  %DO I=1 %TO &TOTOBS;
    %EXAMPLE5;
  %END;
%MEND LOOP;
%LOOP;
```

Program Code: EXAMPLE5

This process is not as intuitive as the previous example. There are a number of differences that all need to be explained. In the order in which they appear, there are no positional parameters, the use of the "**&I**" macro variable, the double ampersand "**&&**" value, the DATA STEP option **END=**, the concatenation "**||**" in the creation of **MACVAR**, the use of **_N_**, the use of the **EOF** variable within the **IF** statement, and the **%DO** in the **MACRO LOOP** at the end.

### DATA _NULL_

The **END=EOF** DATA SET option will tell the DATA STEP when it has reached the LAST record in the table. When the last record is reached the step creates the **TOTOBS** macro variable using the **_N_** DATA SET variable, which is a SAS system counter that is incremented during each iteration.

**_N_** is also used during the creation of the **MACVAR** variables. During the first iteration of the DATA STEP, the **CALL SYMPUT** creates a macro variable called **MACVAR1** that contains the value of the CPT code for the FIRST record in the table. Each subsequent iteration creates another macro variable corresponding to the observation currently in the Program Data Vector. The "effective code" of this step is shown at right.

```
%LET MACVAR1 = 21081;
%LET MACVAR2 = 21082;
%LET MACVAR3 = 21083;
%LET MACVAR4 = 21084;
%LET MACVAR5 = 21085;
%LET TOTOBS  = 5;
```

Effective Code: EXAMPLE5

### MACRO LOOP

The next step in the evaluation of this process will look at the **%DO** in the SAS macro called **LOOP**. Notice the macro has no positional parameters. Nothing is being passed into it. The **%DO-%TO-%END** are SAS macro code and can only be executed from within a SAS macro. In other words, they cannot be executed in OPEN CODE. Here the process will execute the **%EXAMPLE5** macro for the total number of observations within the **TOTOBS** macro variable, 5 in this case as shown in the "effective code".

```
%MACRO LOOP;
  %DO I=1 %TO &TOTOBS;
    %EXAMPLE5;
  %END;
%MEND LOOP;
%LOOP;
```

Macro Loop: EXAMPLE5

### DOUBLE AMPERSAND "&&"

The final stage evaluates the double ampersand "**&&**" used in **EXAMPLE5**. The first thing to recognize is during the execution of the DO LOOP, a new macro variable "**I**" was created. This macro variable contains the current iteration of the LOOP. The first time through, it evaluates to 1, then 2, and so on until it reaches 5. The evaluation of the "**&&MACVAR&I**" variable is more complicated. The result created by the double ampersand during the first iteration will be, "**&MACVAR1**", which resolves to "**21081**". Each subsequent iteration of the DO LOOP yields an increasing number, "**&MACVAR2**", "**&MACVAR3**", based on the value of "**I**".

```
%MACRO EXAMPLE5;

  DATA TEMP&I;
    GEE  = "&&MACVAR&I";
    WHIZ = &I;
  RUN;

%MEND EXAMPLE5;
```

Macro EXAMPLE5

## CONCLUSION

The SAS Macro Facility is one of the most powerful scripting languages on the market and can be employed to create very dynamic processes with ever increasing functionality. It would behoove every SAS programmer and statistician to learn more about this robust facility. The applications shown here range from the most basic to the somewhat intricate, though only begin to scratch the surface of the capabilities.

## REFERENCES

SAS Guide to Macro Processing, Version 6, Second Edition
SAS Macro Facility Tips and Techniques, Version 6, First Edition
Chapter 41, "Macro Language Features" in SAS Technical Report P-222
Chapter 42, "The Autocall Facility" in SAS Technical Report P-222
Chapter 43, "The Stored Compiled Macro Facility" in SAS Technical Report P-222

## ACKNOWLEDGMENTS

The author wishes to thank David Gibson and Fran Alfano for presenting the challenges that facilitated the need to create dynamic macros as well as the SAS Help Desk for their tireless efforts to answer seemingly endless questions.

## CONTACT INFORMATION

Your comments and questions are valued and encouraged.

**Rick Andrews**
Centers for Medicare and Medicaid Services
Office of Research, Development, and Information
7500 Security Boulevard
Baltimore, MD 21244
Phone: (410) 786-4088
E-mail: Richard.Andrews@cms.hhs.gov