Paper 90-2010

Ways of Creating Macro Variables

Kelley Weston; Quintiles; Overland Park, KS USA

Abstract

There are many ways of creating a SAS® macro variable, many of which many programmers are aware, but perhaps some methods may be new. Additionally, the scope and length of the macro variables are things of which to be aware. This will give you many of the ways of creating macro variables in SAS, so you can be aware of when a macro variable is being created, as well as the scope and values, including their length. This will not necessarily give all of the possible methods of creating SAS variables, as it only addresses Base SAS; for example, there are also ways of creating macro variables using SAS/Connect®.

Introduction

Have you had to deal with using macro variables, and sometimes wondered about seemingly odd behavior, only to find that the value was something other than what you expected – too long, too short, or something completely different? There are many ways of creating macro variables in SAS – some obvious and straightforward, others not so much. This paper will list many of the common and some not-so-common methods of creating macro variables in SAS, and also address their scope (local vs. global) and their values.

%LET

This is perhaps the most common method of creating a macro variable:

```
%LET x = MWSUG;
```

In this case, the macro variable name is "x", and the value is "MWSUG", with a length of 5. Unlike a DATA step character variable, the length of a macro variable can be changed at will.

The scope of the variable is dependent on several things. If the statement is used outside of a macro, then it is a global variable. If the variable is defined inside of a macro, and that is its first appearance, then it will be local to that macro:

```
%MACRO test;
     %LET x = MWSUG 2010;
%MACRO test;
```

Assuming that the macro variable "x" has not been defined or referenced before, "x" will be a macro variable local to the macro "test", with the value "MWSUG 2010", with a length of 10.

The interesting thing is when macro variables are defined both inside and outside of a macro with the same name:

```
%LET x = MWSUG;
%MACRO test;
     %LET x = MWSUG 2010;
%MACRO test;
%test;
```

The %LET statement defined before the macro creates a global macro variable with a value of "MWSUG". The %LET statement inside the macro does not create a local macro variable, but changes the value of the first macro variable. If you were to %PUT the value of "x" inside the macro, you would find that before the %LET statement, the value is "MWSUG", but after the %LET statement, it is "MWSUG 2010". The interesting thing is to print the value after the macro has executed. You would find that the value is still "MWSUG 2010". The macro variable was initially created as a global variable, and changed inside the macro. We will expand on this example.

Be aware that the value of a macro variable created with a %LET statement will not include any leading or trailing spaces. The following statements all result in the macro variable x having a value of "MWSUG", with no leading or trailing spaces, with a length of 5:

```
%LET x = MWSUG;
%LET x =      MWSUG;
%LET x = MWSUG       ;
%LET x =         MWSUG       ;
```

If for some reason you wish to include spaces at the beginning &/or end of the value, just use the %STR function. For example:

```
%LET x = %str(    MWSUG);
```

%GLOBAL

The %GLOBAL statement creates a macro variable that is obviously global (never local), no matter where the statement appears. Since a macro variable definition that is outside of any macro is automatically global, it is redundant to use the %GLOBAL statement outside of a macro if the macro variable is defined in another way, although there is no problem with this. You cannot give a value to the macro variable with the %GLOBAL statement, so it always has a null value (length = 0).

%LOCAL

The %LOCAL statement can only be used inside of a macro. You cannot give a value to the macro variable with the %LOCAL statement, so it always has a null value (length = 0). There are a couple of reasons to use this statement:

1. Ensure that the variable being defined inside of a macro does not overwrite a global macro variable by the same name.
2. Ensure that the macro variable does not exist after the macro finishes executing.

Returning to our earlier example, let's look at a global and local variable by the same name:

```
%LET x = MWSUG; <= 1
%MACRO test;
     %LOCAL x;                <= 2
     %LET x = MWSUG 2010;     <= 3
%MACRO test;
%test;
                              <= 4 (after macro executes)
```

Examining the value of "x" at various points, you would find the following:

1. before the macro executes, after the %LET executes, x has the value of "MWSUG"

2. inside the macro, after the %LOCAL statement executes, before the %LET statement executes, x has a null value (length = 0).

3. inside the macro, after the %LET statement executes, x has a value of "MWSUG 2010", with a length of 10.

4: after the macro finishes, the variable x defined inside the macro no longer exists, so you see that the global macro variable x still has the value of "MWSUG".

Parameter of a macro

When a macro is defined with parameters, the macro variables that are created are always local to the macro. If they are not passed a value during the macro call, and do not have an initial value (as keyword parameters can have), then the macro variable has a null value (length = 0).

```
%MACRO test(mac =);
     <macro statements>
%MEND test;
```

In this case, if the macro %test is called without giving a value to the macro variable "mac", then "mac" has an initial value of null, with a length of 0.

%DO loop index variable

When you use a %DO loop in a macro, if the macro variable has not been defined previously, this will create it.
As an example:

```
%MACRO test;
        %DO cnt = 1 %TO 5;
                <macro statements>
        %end; %* do cnt loop;
%MEND test;
```

In this case, the %DO statement creates a macro variable called "cnt" if it does not already exist.

As a suggestion, you may consider doing two things:

1. Always using a %LOCAL statement explicitly declaring the loop counter as a local variable.
2. Using the declared macro variable only as a loop counter, so you can always keep track of it and know what variable(s) is/are being used as loop counters.

CALL SYMPUT / CALL SYMPUTX

CALL SYMPUT and CALL SYMPUTX are similar DATA step call routines. The difference between them is that CALL SYMPUTX is a call routine new to version 9, and it does not include any leading or trailing blanks.

They can be used as follows:

```
DATA _NULL_;
     CALL SYMPUT ("x1", "MWSUG");
     CALL SYMPUTX ("x2", "MWSUG");
RUN;
```

Examining the values of the macro variables "x1" and "x2" after the DATA step completes, you will find that both variables have the value of "MWSUG". The difference will be apparent if there are any leading or trailing spaces defined in the CALL routines:

```
DATA _NULL_;
     CALL SYMPUT ("x1", "   MWSUG   ");
     CALL SYMPUTX ("x2", "   MWSUG   ");
RUN;
```

There are three spaces both before and after the value of "MWSUG" in both statements, and so both values have a length of 11; however, in examining the values of the macro variables after the DATA step completes, we find that "x1" still has the leading and trailing spaces, and so has the value of "   MWSUG   ", with a length of 11, while "x2" does not maintain its spaces, and so has the value of "MWSUG", with a length of 5. The use of CALL SYMPUTX eliminates the need for the use of the LEFT and TRIM functions (or the single STRIP function) around the value, as we are used to doing when using the CALL SYMPUT routine.

As to the scope of the macro variables created with these routines (global or local), the macro variables are obviously global if the DATA step is used outside of a %MACRO definition. However, if the DATA step is defined inside of a %MACRO definition, that is not sufficient to define the macro variable as a local macro variable. If you want the macro variable to be local, be sure to precede the DATA step with a %LOCAL statement inside the %MACRO definition. There are also some other conditions that will determine whether or not variables created with these routines are local or global. Please see the SAS documentation for the details.

PROC SQL INTO Clause

The INTO clause of PROC SQL creates macro variables, and the attribute of extra blanks depends on how the variables are created.

For instance, this will create one macro variable:

```
PROC SQL;
        SELECT varx
                INTO :macx
                FROM work.test;
QUIT;
```

This will create a single macro variable called "macx", keeping any leading or trailing blanks. If varx is a number, the macro variable will be right-justified and 8 characters long, including leading spaces.

One way to check the exact value of macro variable is to print it similar to the following:

```
%PUT varx = /&varx/;
```

This will allow you to check if there are any leading &/or trailing spaces.

The easiest way to eliminate leading and trailing spaces is to use a %let statement:

```
%LET varx = &varx;
```

If you wish to put multiple values of a single SAS data set variable into a single macro variable, you could do it like this:

```
PROC SQL;
        SELECT varx
                INTO :macx
                SEPARTED BY ","
                FROM work.test;
QUIT;
```

This will create a single macro variable called "macx" containing all of the values of the data step variable "varx", separated by a comma, with no leading or trailing spaces.

If for some reason you need to maintain any leading or trailing spaces, you can use the NOTRIM option:

```
PROC SQL;
	SELECT varx
		INTO :macx
		SEPARTED BY "," NOTRIM
		FROM work.test;
QUIT;
```

If there is a possibility that there are duplicate values, then use the DISTINCT keyword:

```
PROC SQL;
	SELECT DISTINCT varx
		INTO :macx
		SEPARATED BY ","
		FROM work.test;
QUIT;
```

Additionally, this will also put the values in ascending order.

Instead of putting the values into one macro variable, you can put the values into several macro variables:

```
PROC SQL;
	SELECT DISTINCT varx
		INTO :macx1 - :macx3
		FROM work.test;
QUIT;
```

In order to make this dynamic, first count the number of values, then eliminate the leading spaces from the count, then use the number of values to create the macro variables:

```
PROC SQL;
	SELECT COUNT(DISTINCT(varx))
		INTO :cnt_varx
		FROM work.test;
	%let cnt_varx = &cnt_varx;
	SELECT DISTINCT varx
		INTO :macx1 - :macx&cnt_varx
		FROM work.test;
QUIT;
```

Just like with most other situations, if the macro variables are created with PROC SQL inside a macro, then the macro variables will be local if they did not exist before the macro call, or if they are not explicitly declared to be global.

ODS OUTPUT statement using the MATCH_ALL option

When running a PROC that will create multiple output objects, you can use the ODS OUTPUT statement with the MATCH_ALL option. This will create separate data sets for each object, and will put the names into a macro variable, when used as follows:

```
ODS OUTPUT <object-name>(MATCH_ALL = <macro_variable>) = SAS-data-set-prefix
```

For instance, with a line like this

```
ODS OUTPUT SelectionSummary (MATCH_ALL = ds_list) = prefix
```

This will put the list of data set names from the output object "SelectionSummary" into a macro variable called "ds_list"; the names will start with "prefix", and if there is more than one data set, the names will then proceed with numeric suffixes; thus the names of the data sets will start with "prefix", then the second one would be "prefix1", and so on. The data set names would be separated by a space in the macro variable.

If there is more than one data set, you could then parse the macro variable, if necessary. However, if you were to use then in a SET statement, it might look like this:

```
SET &ds_list;
```

Which might then be parsed like this:

```
SET prefix prefix1 prefix2;
```

Conclusion / Summary

It is certainly useful to know various ways that macro variables can be created, and the nuances of them. This paper discussed the following methods:

1. %LET
2. %GLOBAL
3. %LOCAL
4. Macro parameter
5. %DO loop index variable
6. CALL SYMPUT / CALL SUMPUTX
7. PROC SQL INTO clause
8. ODS OUTPUT statement using the MATCH_ALL option

References

SAS Online Doc

About the Author

Kelley has over 18 years experience with SAS programming. Kelley has been a programmer with a major telecommunications company and a SAS instructor teaching SAS classes nationwide, as well as teaching hands-on workshops at regional SAS conferences and PharmaSUG. His programming experience also includes clinical trials with a major pharmaceutical company and an international CRO. He is an active member of KCASUG (Kansas City Area SAS Users Group), including stints as Webmaster (2006 & 2007) and Chairman (since 2008).

Contact Information

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

Kelley Weston
c/o Quintiles
6700 W. 115th St.
Overland Park, KS 66211
Kw.mwsug@gmail.com

Trademarks

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are registered trademarks or trademarks of their respective companies.