

An Introduction to Macros

Linda M. Quinn
QED Industries, Inc.

ABSTRACT

The Macro facility in SAS is one of the most powerful tools of the SAS System. This tutorial is designed for the SAS user that has never used macros or has used them without knowing what they do. Macros allow us to set session and program specific variables, do repetitive tasks, and process selectively. This tutorial will walk through the basic use of macros and allow you to begin using macros.

INTRODUCTION

This is an introduction to macros, targeted at SAS programmers new to macros. It will be designed to learn by example. The macro facility is used to manipulate SAS source statements. The macro facility has two components, macro variables which can be identified with a leading ampersand (&) and macros which are identified with leading percent signs (%).

The macro language is best for:

- For programs that perform repetitive task, such as processing similar datasets in the same fashion; or processing a single dataset using several procedures.
- For conditional execution of SAS code, perhaps based on the desires of the end user.
- For grouping blocks of code together for retrieval at a later time.

The macro language should not be used

- Because it's there. Any program could be considered a macro and a single macro invocation be used to run it. This adds to overhead since the macro facility processes before the SAS language processor.
- To replace a %INCLUDE statement which does not require the additional overhead of the macro processor.
- For job security. Debugging and maintaining macros, like programs, depend on the degree of difficulty.

MACRO VARIABLES

A macro variable is simply a text string that is given a macro name. Once defined, a macro variable can be used over and over. There are a variety of ways to create a macro variable, but the simplest is with a %LET statement.

```
%LET INDATA = TEST;
```

Notice that we are using a %LET statement. Similar to the LET statement in the standard DATA statements, it creates a variable. The difference between LET and %LET is that %LET creates a macro variable and LET creates a SAS dataset variable.

The name of the macro variable is &INDATA. However, the leading ampersand is not needed in the %LET statement because SAS knows that you are creating a macro variable. Now we can use &LNAME anywhere we would have type MYLIB in our SAS code.

```
proc means data=&INDATA; run;
```

SAS would see this statement as:

```
proc means data=TEST; run;
```

Why would we want to do this? After all it now has taken two statements to do what we could have done in one. Suppose we now change the dataset being used to a production dataset PROD? We would have to go through and change TEST to PROD everywhere. This may not be a problem if there was only one instance, but what if this dataset is used in various procedures, set statements, and so on. Sure, we could use some global change commands, but we all know the caveats with those. Instead, we can create a macro variable and use it throughout the entire program.

```
%LET INDATA = TEST;
proc print data=&INDATA;
run;
```

```
proc means data=&INDATA;
run;
proc sort data=&INDATA;
  by mergevar;
run;
data newdata;
  merge &INDATA
        othrdata;
  by mergevar;
run;
```

So now, a variable is set once at the top of the program and its value is substituted in wherever the macro variable appears in the subsequent code.

What are some uses for macro variables? I use them for a variety of reasons:

- Specifying cutoff values for calculations or subgroups, so that I can re-do analyses if the cutoff value may change:

```
%LET cutoff = 75;
...
data testdata;
  set testdata;
  if testscor >= &cutoff then pass=1;
  else pass=0;
run;
```

- Specifying a where clause that I may want to change to do different things:

```
%LET WHERECLS = where pass eq 1;

proc print data=testdata;
  &wherecls;
run;
```

Now, I will get only those observations where pass=1. If I change the value in the &LET clause to

```
%LET WHERECLS =;
```

the code SAS would generate would be:

```
proc print data=testdata;
run;
```

- Specifying a variable that I might be interested in varying:

```
%LET MYVAR = age;
```

```
proc ttest;
  var &MYVAR;
  class group;
run;
proc npar1way;
  var &MYVAR;
  class group;
proc glm;
  model &MYVAR = group;
run;
```

Now I can easily run the same analyses for weight, height, and blood pressure by changing one %LET statement.

The use of macro variables at the beginning of a program, is just the beginning of what can be done.

MACROS

A macro is only slightly more complicated than a macro variable. Similar to the macro variable, the result of a macro call is generated SAS code. The power, however, of the macro is the ability to use executable type command and alter the generated text with logic control. Another way of thinking about a macro is to think of it as a subroutine call.

Subroutines or macros are used to do repetitive tasks over and over with the code existing and being maintained in only one spot. The simplest macro call, or subroutine, has no parameters passed. Macros are defined between two statements, the %MACRO and %MEND statements.

Suppose we looked through our code and we saw the following many times:

```
proc print; run;
proc means; run;
```

This code follows every dataset we create. Let's create a macro. The macro needs a name, let's call it PRINMEAN. The macro definition looks like:

```
%MACRO PRINMEAN;
proc print; run;
proc means; run;
%MEND PRINMEAN;
```

This code has to precede any calls to it. After all, how could the SAS System know what to substitute if it hasn't been defined yet! Some notes: the name of the macro appears on the %MACRO statement. This is the name that will be used to call it. Remember to not use reserved SAS names or you could have a nightmare on your hands. I have also put the name in the %MEND statement. That is optional, but for me, it is a nice delineation of where the macro ends.

Now we want to replace the two lines of code in the program with a macro call. The call looks like:

```
%PRINMEAN;
```

Remember that the macro call starts with a percent sign, not an ampersand. So code that looked like:

```
data one;
  set master;
run;
proc print; run;
proc means; run;
data two;
  set one;
run;
proc print; run;
proc means; run;
```

becomes:

```
data one;
  set master;
run;
%PRINMEAN;
data two;
  set one;
run;
%PRINMEAN;
```

and the results are the same.

But let's take advantage of the power of macros! Let's complicate what we have done by passing

parameters to the macro. This is done on the macro call and in the %MACRO statement.

Suppose we wanted to make the code above a little more specific. Instead of relying on the last dataset created to be the default for the proc prints and means, we want to specify the dataset. Let's pass the dataset name as a parameter. The macro definition would look like:

```
%MACRO PRINMEAN (DSNAME);
proc print data=&DSNAME;
run;
proc means data=&DSNAME;
run;
%MEND PRINMEAN;
```

The macro variable &DSNAME is created only for the duration of the macro execution (i.e., it is a local macro variable). Once again, the ampersand isn't needed in parameter list because the %MACRO tells SAS that the parameters will all be macro variables. When the macro variable is used within the macro, the ampersand is needed.

The call

```
%PRINMEAN (one);
```

would produce the following SAS code:

```
proc print data=one;
run;
proc means data=one;
run;
```

I use macros to pass variables as well as dataset names. Suppose we want to add a var statement with a variable list to the above macro. Now the definition looks like:

```
%MACRO PRINMEAN (DSNAME, VARLIST);
proc print data=&DSNAME;
  var &VARLIST;
run;
proc means data=&DSNAME;
  var &VARLIST;
run;
%MEND PRINMEAN;
```

Now a call can look like:

```
%PRINMEAN (one, age height weight);
```

which would produce:

```
proc print data=one;
  var age height weight;
run;
proc means data=one;
  var age height weight;
run;
```

Notice here that the parameters being passed, and the macro variable within the call &VARLIST is simply a text string, with several actual dataset variables. How would the call look if we wanted all the variables, and didn't want a var statement? We would have to use the key word all otherwise we can't do it because the keyword VAR is hardcoded into the macro. If we really wanted to allow that option, we could alter the macro definition:

```
%MACRO PRINMEAN (DSNAME, VARLIST);
proc print data=&DSNAME;
  &VARLIST;
run;
proc means data=&DSNAME;
  &VARLIST;
run;
%MEND PRINMEAN;
```

but now the parameter must have the keyword VAR in it:

```
%PRINMEAN (one, var age height weight);
```

and we would get all the variables with the second parameter missing:

```
%PRINMEAN (one, );
```

Now notice that we still had to hold the parameter position in the macro call. Also, notice that the order of the parameters is implicit. There is also an explicit way to specify parameters. The same macro definition and call would look like:

```
%MACRO PRINMEAN (DSNAME=, VARLIST=);
proc print data=&DSNAME;
  &VARLIST;
run;
proc means data=&DSNAME;
  &VARLIST;
run;
%MEND PRINMEAN;
```

```
%PRINMEAN (DSNAME=one,
  VARLIST=var age height weight);
```

Alternately,

```
%PRINMEAN (VARLIST=var age height weight,
  DSNAME=one);
```

Or in the case of not specifying a variable list:

```
%PRINMEAN (DSNAME=one);
```

That gets us through a lot of the basics. Now let's try some of the control and condition execution logic.

%IF - %THEN - %ELSE AND %DO - %END

Similar to the IF-THEN-ELSE and DO-END constructs in the data step, the same constructs exist in the macro facility, only with percent signs in front of them. In fact, many of the same logical constructs are in the macro facility. These constructs are only valid to use within a macro definition. They will give errors in open code.

Suppose I have 5 datasets, DATA1, DATA2, ..., DATA5 that I want to process the same way as macro PRINMEAN would do. I could call PRINMEAN five times, or I could use a macro that looks like:

```
%MACRO DATALOOP (N);
%DO I = 1 %TO &N;
  proc print data=DATA&I;
  run;
  proc means data=DATA&I;
  run;
%END;
%MEND DATALOOP;
```

and the call would look like:

```
%DATALOOP(5);
```

This would run a proc print and a proc means on all five datasets and would actually generate the following SAS code;

```
proc print data=DATA1;
run;
```

```

proc means data=DATA1;
run;
proc print data=DATA2;
run;
proc means data=DATA2;
run;
proc print data=DATA3;
run;
proc means data=DATA3;
run;
proc print data=DATA4;
run;
proc means data=DATA4;
run;
proc print data=DATA5;
run;
proc means data=DATA5;
run;

```

Let's look at a few things that happened here.

- There is a macro variable `I` created by the use of the `%DO` loop. Remember that typically an ampersand isn't needed when creating a variable with a macro statement.
- The macro variable `I` is used to form the dataset name by adding the value of `I` to the word `DATA` to form `DATA1`, `DATA2`,...
- Macro variables can be used almost anywhere there is text, not just as standalones.

MACRO FUNCTIONS

Many of the same string functions available in the data step are also available in the macro facility, of course preceded by a percent sign. For example, `%SUBSTR()` works similar to the `SUBSTR()` function.

I want to describe one of the most useful functions. `%EVAL` evaluates arithmetic and logical expressions using integer arithmetic. I use this function when I am changing the value of a macro variable that contains a number. Remember, to SAS, a macro variable is just a character string. So, a statement

```
%LET MACROVAR = count + 1;
```

would actually create a macro variable called `macrovar` that would have a value of "count + 1" so if I wanted to use it to reference dataset `DATA2` and `count` was 1 the code

```

data new;
  merge old
        data&MACROVAR;
  by id;
run;

```

would end up looking like:

```

data new;
  merge old
        datacount + 1;
  by id;
run;

```

which wouldn't make any sense and give an error.

Instead, if we use the statement

```
%LET MACROVAR = %EVAL(count + 1);
```

SAS would evaluate the statement and put the result into macro variable `&MACROVAR`. And the resulting code would be correct:

```

data new;
  merge old
        data2;
  by id;
run;

```

Two other functions that are extremely useful are `CALL SYMPUT` and `%SYMGET`. The `SYMPUT` routine allows you to create a macro variable or assign a new value to an existing one from a dataset variable. The macro variable is then available for use in a subsequent data step or procedure. The `SYMGET` function allows you to extract values from a macro variable that may in turn be assigned to a dataset variable.

PUTTING IT ALL TOGETHER

This only touches on the beginnings of the macro facility. Some more complex examples will be walked through in the presentation. Macros can be used in every aspect of the SAS System and should be considered a valuable tool for the SAS programmer.

Like my SAS code, I try to get a macro that works first, and then build in more and more complexity.

ERRORS AND DEBUGGING

Make sure the macro facility is accessible to you. One of the system options is MACRO (or NOMACRO). This should be set to MACRO if you are intending to use macros.

Errors are inevitable! Macros need not be intimidating. I find that there are many ways errors creep into my macro programming. Some are silly like missing an ampersand or a percent sign. Others are more difficult to find, such as quoting in macros. Error messages upon execution are more often than not either cryptic or pointing to something that doesn't look like an error. Some system options that should be in effect when debugging the code are:

OPTIONS MPRINT SYMBOLGEN MLOGIC;

MPRINT specifies that the SAS code generated by the macro prints in the log. Use this to verify that the code you want to generate is in fact being generated.

MLOGIC specifies whether or not the macro language processor traces its execution. This will give information on conditional execution and what the resolution of %IF statements are.

SYMBOLGEN specifies whether the macro processor displays the result of resolving macro variable references. Use this to verify that your macro variables are resolving correctly.

I found that the best way to learn is to jump in and do it. Yes, they can be frustrating, but they can also save an immense amount of time and coding.

TIPS

- Provide some error checking or macro progression messages using the %PUT statement.
- Explicitly define macro variables in a %LOCAL or %GLOBAL statement unless it is a parameter, which by definition is local. This will remind you what the intended duration of the macro variable.
- Using keyword parameters rather than positional parameters all for setting default values, order-free specification, and easier understanding and documentation.

- Parameters keep your macros flexible and adaptable.
- Start simple and build up. Like code, it is better to have something work first and then add options and features.
- Don't get discouraged with the error messages. Use the macro system options to help in the debugging process. Don't forget to turn them off if you don't want all the lines in the log.
- Share your macros with colleagues to save their time as well. Develop macro libraries.

REFERENCES

Carpenter, Art (1998), *Carpenter's Complete Guide to the SAS(R) Macro Language*, Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1994), *SAS Macro Facility Tips and Techniques, Version 6, First Edition*, Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1997), *SAS Macro Language: Reference, First Edition*, Cary, NC: SAS Institute, Inc.

PRODUCT INFORMATION

SAS and the Macro Facility are registered trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

AUTHOR CONTACT

Linda M. Quinn
QED Industries, Inc.
1286 Gary Blvd.
Brunswick, OH 44212-2912

Phone: 330-225-4184
Fax: 330-220-9036
e-mail: lmq2@po.cwru.edu