Beginners Guide to Flexibility: Macro Variables

Mel Widawski, Office of Academic Computing, UCLA, Los Angeles, California

ABSTRACT

Expert programmers rarely program from scratch, rather they use existing code and modify it. Use macro variables to facilitate writing code that is flexible and easy to re-use.

Put information such as file locations, lists of variables, and selection criteria in macro variables at the beginning of a program and simply refer to them as needed throughout the program. You will learn how to create and use these macro variables.

Please be aware that you can use these macro variables without having to learn the SAS® macro language. Using macro variables is extremely simple.

INTRODUCTION

I have seen beginners take a look at the SAS macro language and be thoroughly daunted by it. This even happens when they approach learning macros through the truly excellent book by Art Carpenter, *Carpenter's Complete Guide to the SAS Macro Language.* As Art shows in Chapter 2, the use of macro variables, alone, can increase the flexibility of programs greatly. You do not have to learn the macro language and use macros to make use of macro variables. They can be used in the body of any program.

Using macro variables is almost trivially simple. After showing you how to create and use macro variables, the bulk of this presentation will be devoted to teaching you to recognize where the use of macro variables is warranted and will result in programs that are easier to maintain.

There are three main situations where the use of macro variables is indicated, and can increase flexibility. Use of macro variables also decreases the time it takes to adapt the program to new situations. You will find that the more flexible your code, the less time you will have to spend in repetitive programming.

First: When you have a long program and are repeating file names, lists of specific variables, or settings. You can of course search through the entire program each time you want to change one of these settings, but that is tedious, and prone to error. A better way is to set a macro variable at the beginning of the program and then invoke the variable each time within the program. If you want to change the specification you simply change it once where it is set at the top of the program. The macro variable then makes the new value available wherever it is referenced.

Second: Debugging statements that are useful especially with **DATA** steps, that you want available if there are problems, but don't want to use when the program is working well. This makes it easy to comment out sections of code, without searching through the file for each instance. You simply change the macro variable definition at the beginning of the program.

Third: Macro variables are useful when you have a program segment that is essentially repeated with different lists of variables or different files a number of times in the same program. Also, if the code is re-used often as separate programs with different variables or files, then it is useful to change the setting at the beginning. You will be shown a technique where the code does not have to be repeated, but is just included in the program as many times as you need.

When you start using macro variables you will find many more uses for them, but all are essentially variations of these three situations.

USING MACRO VARIABLES: BASIC

You can do much with macro variables just knowing how to assign a macro variable with **%LET** and how to reference a macro variable with the **&** (ampersand).

%LET

The statements that begin with **%LET** establish macro variables for use in the program. They also assign values to these macro variables. I usually place most of the **%LET** statements at the beginning of the program. This practice also allows you to better document your program. If the main variables that may need changing are at the beginning of the program, then it is easy to make changes without a lot of searching.

Notice the following example creating two macro variables, **a** and **b**.

```
%LET a=one;
%LET b=two;
```

These statements assign the value of **one** to the macro variable **a**, and **two** to the macro variable **b**. One thing to remember is that the contents of macro variables are always treated as a character string. This becomes more important when you are actually writing a macro program. But for our purpose here, all it means is that you only have to use quotes in a **%LET** statement if you want the quotes to appear in the statement where you reference the macro variable.

%PUT

The statements that begin with **%PUT** allow you to write the comments and values of macro variables to the **LOG**.

It is useful to check that the macro variable was assigned a correct value.

& Macro Reference

Once you have defined a **macro variable** with **%LET** statements, you can reference the **macro variable** by preceding it with an ampersand. When the program is run, the **macro variable** is replaced by the string assigned in the **%LET** statement. The form of the reference for a **macro variable** is

```
&macro_variable_name
```

Where **macro_variable_name** was previously assigned in a **%LET** statement. Thus, to reference the macro variable **a** above you simply include **&a** in your code. Notice the following example using the two macro variables **a** and **b**.

```
%PUT &a&b;
%PUT &a &b;
```

These statements result in two lines being printed.

```
onetwo
one two
```

Occasionally a **macro variable** reference needs to be followed by a period. This is necessary to separate a **macro variable** from text that follows it when referenced. Since it never causes misinterpretation to follow a **macro variable** reference with a period, I recommend that you always follow this practice.

```
%PUT &a.&b.;
%PUT &a. &b.;
```

These **%PUT** statements yield exactly the same results as those presented above.

Simple Sample Program

The following program demonstrates the creation of, and reference to **macro variables**. It also demonstrates the results when a period follows the **macro variable** reference.

```
%LET a=one;
%LET b=two;

%PUT &a&b;
%PUT &a.&b;
%PUT &a..&b;
%PUT &a.&b.;
%PUT &a. &b.;
%PUT &a &b;
%PUT &a.b;
%PUT &ab;
%PUT &a..b;
```

This program results in the following output. For your convenience the program line is presented to the left, and the output to the right. Also, each line is numbered for easy reference.

```
1  %PUT &a&b;          onetwo
2  %PUT &a.&b;         onetwo
3  %PUT &a..&b;        one.two
4  %PUT &a.&b.;        onetwo
5  %PUT &a. &b.;       one two
6  %PUT &a &b;         one two
7  %PUT &a.b;          oneb
8  %PUT &ab;           &ab
9  %put &a..b;         one.b
```

This example shows the effect of including a period after the **macro variable** reference. In most cases the result is the same. In line 7 the period is necessary or the **macro variable** name is assumed to be **ab** as in line 8. With line 8 you also get the following warning since **ab** is not defined.

```
WARNING: Apparent symbolic reference AB
         not resolved.
```

If you want a period to appear after the **macro variable** reference is resolved, then two periods must be typed as in line 3 above.

USING MACRO VARIABLES : EASY UPDATES

Using just the elements we have already learned, let us take a look at an example where defining elements of the program at the beginning allows easier and quicker updates to the program. The program used in this example takes a SAS listing file (**vpagx.lst**) and condenses the output on as few pages as possible. This is accomplished without breaking the output in the middle of any of the original pages.

The program requires you to know how many lines of output per page your printer assumes, given the printing font you have chosen, **ps**. You will also specify the title string that signals a new page of output, **str**. Finally, you specify the name of the file containing the output (**.\vpagx.lst**), and the name of the file to write the new output (**.\vpagxo.lst**).

Each of these are specified at the beginning of the program as **macro variables**. Notice the following **%LET** statements.

```
%LET ps=78;
%LET str=The SAS System;
%LET infile=.\vpagx.lst;
%LET file=.\vpagx.lst;
```

Now look at the same statements in context in the following sample program on the next page.

```
/* ********************************************** */
/* ***          OUTPUT CONDENSE                *** */
/* ***                                         *** */
/* ***  Set the line length PS= and the search *** */
/* ***  string STR= at the beginning of the    *** */
/* ***  program using Macro Variables.         *** */
/* ********************************************** */

%let ps=78;     /* This should equal # of lines in  */
                /* an output page for your printer. */
%let str=The SAS System;
%let infile=.\vpagx.lst;
%let file=.\vpagxo.lst;

/* ********************************************** */
/* ***  This DATA step determines the lenght   *** */
/* ***  in lines of each input page, and saves *** */
/* ***  that value in a data set called "a"    *** */
/* ********************************************** */
data a;
    infile "&infile" length=len end=eof;
    input test $varying200. len  @;
    if index(test,"&str")>0 then do;
         if n>0 then output ;
         n=0;
    end;
    n+1;
    if eof then output;
    keep n;
run;

/* ********************************************** */
/* ***  This DATA step determines how many     *** */
/* ***  input pages fit on an output page.     *** */
/* ***  It then fills the rest of the page     *** */
/* ***  with blank lines up to the output page *** */
/* ***  size.                                  *** */
/* ********************************************** */
data _null_;
    infile  "&infile" length=len end=eof;
    file    "&file";

    input test $varying200. len  @;
    if index(test,"&str")>0 then do;
                set a;
                if (nn+n)>&ps then do;
                       do i=1to (&ps-nn);
                           put ;
                         end;
                    nn=0;
                    end;
       end;
    nn+1;
    put _infile_;
    if nn=&ps or eof then do;
         nn=0;
    end;
run;

/* ********************************************** */
/* ***  Just open the output file "vpagxo.lst" *** */
/* ***  in the program editor. Do a print      *** */
/* ***  preview to insure that the paging is   *** */
/* ***  correct and then print the output.     *** */
/* ********************************************** */
```

In order to modify the program for a new use, all you would have to do is change the specifications on the **%LET** statements at the beginning of the program. You don't have to search through the program for each value of **78**, and then change each one individually. Every where in the program that **&ps** occurs, it will be replaced automatically with the value you set on the **%LET** statement.

If you have a different title string in your output, then you can specify it without searching through the entire program. Notice that if a **macro variable** reference is enclosed in quotes, then **double quotes** are always used. A **macro variable** reference enclosed in single quotes is not replaced in the program with that variable's value. Consider the line of code:

```
IF index(test,"&str")>0 then do;
```

&STR will be replaced by **The SAS System** since it is enclosed in **double quotes**. But if the line read as follows:

```
IF index(test,'&str')>0 then do;
```

Then **&STR** would not be replaced.

USING MACRO VARIABLES: FOR DEBUGGING

The same concepts can be used to aid debugging. Sometimes code needs to be included for debugging, but is not needed for production runs. It becomes cumbersome to keep inserting and removing the lines of code for debugging. Commenting out these lines helps, but once again, when debugging is necessary it is time consuming and prone to errors to go through the program removing or inserting the asterisks to comment out the lines.

Once again **macro variables** can be used to accomplish this task easily and efficiently. Let us take the second **DATA** step in the above program. We will insert lines that will count various components of the output. We will count the lines on the new output page, the number of input pages on each output page, and count the number of printed lines from the input pages that are on the output page.

Each of these lines used for debugging will be preceded by a **macro variable** called **debug** which will either contain a null value or will contain an asterisk (*). When **debug** contains an asterisk, then no debugging will occur. But when **debug** contains a null value then the debugging statements will execute.

```
%LET debug=*;
```

The above statement turns debugging off. To turn debugging on use the following **%LET** statement.

```
%LET debug=;
```

The example of debugging in context follows.

```
%LET debug=*; /* No Debugging */

data z;
    infile  "&infile" length=len end=eof;
    file    "&file";

    input test $varying200. len  @;
    if index(test,"&str")>0 then do;
                set a;
                if (nn+n)>&ps then do;
                       do i=1to (&ps-nn);
                           put ;
 &debug                    nnn+1;
                         end;
 &debug                OUTPUT;
                       nn=0;
 &debug                nnn=0;
 &debug                aa=0;
 &debug                po=0;
                      end;
 &debug           aa+n;
 &debug           po+1;
          end;
    nn+1;
    put _infile_;
    if nn=&ps or eof then do;
 &debug   OUTPUT;
          nn=0;
 &debug   nnn=0;
 &debug   aa=0;
 &debug   po=0;
     end;
 &debug KEEP i n nn nnn aa po;
run;

&debug.PROC PRINT DATA=z;
&debug.RUN;
```

In this program any line preceded with **&debug** will be executed when **debug** is set to null, and will be commented out when **debug** is set to asterisk.

&&

There is another way to accomplish debugging that involves using **&&** for the root **macro variable**. It requires more code, but has the advantage of using the words **on** and **off** for setting the macro variables. I present it here primarily so you can see the function of **&&** for use with macro variables. Consider the following program segment.

```
%LET dbgoff=*;
%LET dbgon=;
```

If you replace **&debug** with **&&dbg&debug** everywhere that **&debug** appears in the previous program, debugging can then be turned off with the following.

```
%LET debug=off;
```

And debugging can be turned on with this statement.

```
%LET debug=on;
```

This is because the double ampersand (**&&**) postpones the substitution of the **macro variable** and retains an ampersand so that the resulting string can be evaluated as a **macro variable**. Let us look at this step by step.

First, the **&&dbg&debug** is evaluated to remove the first set of ampersands. Thus it becomes **&dbgoff** when **debug** is set to **off**. Then **&dbgoff** becomes * because **dbgoff** has been set to asterisk above.

On the other hand, when **debug** is set to **on** then **&dbgon** resolves to null, and the statements are not commented out.

MACRO VARIABLES: REPEATING CODE

With one additional concept, **macro variables** can be used with repeating segments of code within a program. It would be useful in this case to have the code appear only once, and to bring it in when necessary. The additional feature is **%INCLUDE**.

%INCLUDE

You use the **%INCLUDE** statement to bring a program segment stored in a file into your program. The complete path to the code you want to include is specified in quotes on the **%INCLUDE** statement. For example, if the program that puts multiple pages of output on a single page if they fit was stored in a file called **fixout.sas** in your current directory, then you could bring that code into your program with the following statement.

```
%INCLUDE '.\fixout.sas';
```

Using this feature extends the power of **macro variables** greatly. It has as much power as some simple **macros**.

A PROGRAM WITH REPEATING CODE

Assume that the program for condensing output presented above is stored without the **%LET** statements in a file called **fixout.sas**.

Now, let us assume that we have four files containing listings that we would like to condense so that we are using the minimum number of pages for output, but that no original page is split across new pages. The four files are **out1.lst**, **out2.lst**, **out3.lst**, and **out4.lst**.

The variables **ps** and **str** are set at the beginning of the program. Then **infile** and **file** are set for each output file just before the program with the **macro variables** references is included. Thus, the following program is all that you need to condense each output.

```
/* ********************************************** */
/* ***          OUTPUT CONDENSE                *** */
/* ***                                         *** */
/* ***  Set the line length PS= and the search *** */
/* ***  string STR= at the beginning of the    *** */
/* ***  program using Macro Variables.         *** */
/* ********************************************** */

%let ps=78;    /* This should equal # of lines in  */
               /* an output page for your printer. */
%let str=The SAS System;
%let infile=.\out1.lst;
%let file=.\out1o.lst;
%INCLUDE '.\fixout.sas';    /* Include Program    */

/* ********************************************** */
/* ***  Next set of output     OUT2.LST        *** */
/* ********************************************** */
%let infile=.\out2.lst;
%let file=.\out2o.lst;
%INCLUDE '.\fixout.sas';    /* Include Program    */

/* ********************************************** */
/* ***  Next set of output     OUT3.LST        *** */
/* ********************************************** */
%let infile=.\out3.lst;
%let file=.\out3o.lst;
%INCLUDE '.\fixout.sas';    /* Include Program    */

/* ********************************************** */
/* ***  Next set of output     OUT4.LST        *** */
/* ********************************************** */
%let infile=.\out4.lst;
%let file=.\out4o.lst;
%INCLUDE '.\fixout.sas';    /* Include Program    */
```

Compare this to a program where the code for condensing output is repeated four times. While that can be done easily, maintenance is much more difficult. You can cut down typing, cut down the length of programs, and cut down the potential for errors by using **macro variables** and **%INCLUDE** to simplify your programs.

CONCLUSION

You now have a small number of simple tools which can aid you in writing more flexible and reusable code. I have just touched on the surface of the many uses of these techniques, but I hope you can now start to recognize when these tools can be used to save you time and energy.

In summary, I will review the elements you have learned here.

%LET This statement is used to assign values to **macro variables**.

&vnam. This is the way that you would reference the **macro variable** in your program. The period disappears when the variable is referenced.

"&vnam." If you need to include a **macro variable** reference in quotes, then those quotes have to be **double quotes** (").

&&vnam&suf. When you need to create a **macro variable** reference using two components, a prefix and a suffix, then you must use **&&** in front of the prefix. First **&suf.** resolves to its value (e.g. 2) leaving **&vnam2.** and then this is resolved to it's value. In this example **vnam2** must have been set to a value with a **%LET**. Also, **suf** would have been set to a value of **2**.

%INCLUDE This statement is used to bring a program segment saved in a file into a program. It allows for code to be repeated without typing it over and over again. It makes for shorter and easier to maintain programs.

REFERENCES

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

SAS Institute Inc. (1990), *SAS® Language: Reference, Version 6, First Edition*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1990), *SAS® Guide to Macro Processing, Version 6, Second Edition*, Cary, NC: SAS Institute Inc.

ACKNOWLEDGMENTS

I would like to acknowledge Anita Drever who served as a test subject for introducing a beginner to macro variables. Thanks to Barbara Widawski without whose editing this manuscript would be illegible.

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Windows, and Windows NT are registered trademarks or trademarks of Microsoft Corporation in the USA. ® indicates USA registration.

CONTACT INFORMATION

Mel Widawski
Office of Academic Computing
UCLA
Los Angeles, CA

mel@ucla.edu