

Layperson's Macro 101

Mark Terjeson
Russell, Tacoma, WA

ABSTRACT

There is a simple way to grasp the major concepts of the SAS Macro facility in order to easily learn how to create and troubleshoot macro code.

INTRODUCTION

The SAS macro facility is unlike macro facilities in other languages that are more like subroutines or callable subroutines during execution.

The SAS macro stuff, once described, may seem much less sophisticated, but is actually much more powerful in the long run.

The easiest way to get a grip on SAS macro stuff is to think of it as "text-substitution". Learning, creating and troubleshooting SAS macro goodies is far easier if not 100% if you remember text-substitution.

Disclaimer:

My apologies to all of the purists for not utilizing all the technical/proper terminology but I find most people learning don't have all that lingo down yet and they are desirous of understanding how to get themselves productive with the knowledge base they have. So it is okay to learn *how* to use macros well and let some the terminology and the esoteric descriptions come a little later. While specifics and exactness may not abound in the discussion to follow, I find most folks learning SAS macros become very effective (creating and troubleshooting) with this layperson's approach.

Thus, this is entitled Layperson's Macro 101.

THE MACRO STUFF

Prior to sending the SAS code to the compiler/runtime for execution (or whatever terminology works for you) the text that you submit gets run through the macro parser/compiler/preprocessor (or whatever terminology works for you). i.e. macro goodies (usually notated with & or % etc. are found in the chunk of text you have submitted and those references resolve which means what they resolve to is text and that text is substituted into your original text in the internal copy. One tricky but powerful feature is that the text will be processed through the macro parser as many times as necessary until ALL of the macro stuff has been resolved, and THEN the substituted text is sent to the datastep compiler/runtime that you may normally think of.

The macro variables and the sas macros are merely resolved and saved-off text that can be recalled later for text replacement(substitution) and then the resulting text as mentioned gets run.

Enclosed is a description written a long time ago. It is not intended to be an exact technical description of exactly how everything is really designed, but it is intended for a different view of how to grasp SAS macro concepts. It helps grant a picture of what you frequently hear mentioned sometimes as "timing issues".

So here goes...

UNDERSTANDING THE MACRO STUFF

SAS macro stuff may seem overwhelming. So the first step is to get rid of the big black cloud. The best way to understand SAS macro stuff is to think of it as very, very, very simple...

okay, ya ready...

"Text substitution".

That's it.

Everything to do with SAS macro stuff is text substitution.

MULTIPLE PASSES

The only other thing to know is that there are two phases that your text code goes through. First, it goes through a macro parser that ignores all of your regular SAS code. Yes, ignores the text that looks like regular SAS code. As it streams through all of your text it only looks for % and & items. When it finds a % or & it takes the following token and goes and finds the text that has been previously saved off to the side, and merely substitutes the text. When this phase has completed, if there were any % or & the text will have been modified with the text substitutions. The second phase, this resultant text is submitted to the SAS compiler/runtime just as you normally perceive.

THE BASICS

Example, no macro stuff, just a dataset,

```
data temp1;  
    a=1;  
run;
```

When you submit this code, the first phase, the macro parser looks through all of the three lines of text and finds NO % or & and so does nothing. Then the text is sent to the second phase, for execution and it runs the

```
data temp1;  
    a=1;  
run;
```

just as you would expect.

Now for example, let's try this,

```
%macro xxx;  
    hello  
%mend;
```

When this is submitted, the first phase macro parser, finds a % character followed by MACRO. All that is done at this time is that *all* of the text found between these two lines:

```
%macro xxx;  
%mend;
```

is saved off to the side and that chunk of save text is given the name 'xxx'.

That's it. No more. No less.

See I told ya it was simple. har har
Let's keep going.... :o)

If we have in our code this line,

```
%xxx
```

the first phase macro parser finds the % and it then looks at the next token word and finds XXX. It first looks through its library of macro statement word tokens, such as MACRO, MEND, etc. and it does not find a match. It assumes therefore that it must be a chunk of text that you have previously saved. So it goes and looks for XXX. If you had previously submitted the XXX macro, it will find the XXX where it had saved the text.

All that to say it goes and retrieves the chunk of text labeled XXX. And guess what, all it does is merely do text substitution.

```
%xxx
```

in the first phase gets replaced with

```
hello
```

which was the text that was saved off.

That's it. That's what macro stuff is all about in SAS. Why is it so simple. Because sometimes the most simple can become the most powerful!!!

MAKING IT MORE POWERFUL

Someone had the bright idea that if, during the first phase, it could do some conditional checks on what code text to change/substitute, then that could be pretty handy. Note that we are not talking about changing the SAS code that will execute per se, we are talking about changing the text that will *become* the SAS code that then gets executed. Boggle the mind? Let try something.

Let's go back to just the simple text-substitution.

If we had some regular SAS code such as,

```
* copy dataset from one library to another ;  
data bbb.myfile1;  
  set aaa.incomingfile1;  
run;  
  
* copy dataset from one library to another ;  
data bbb.myfile2;  
  set aaa.incomingfile2;  
run;  
  
* copy dataset from one library to another ;  
data bbb.myfile3;  
  set aaa.incomingfile3;  
run;
```

This looks normal. The code text was replicated three times and only the suffix number on our incoming and outgoing datasets is changed each time.

We can implement the pure text substitution as described above like this,

```
%macro cpyfile(thenum);  
    * copy dataset from one library to another ;  
data bbb.myfile&thenum;  
    set aaa.incomingfile&thenum;  
run;  
%mend;  
  
    %cpyfile(1)  
    %cpyfile(2)  
    %cpyfile(3)
```

During the first phase macro parser-----

This text found between %macro and %mend is saved off to the side internally for use later and given the name 'cpyfile'.

Then when it finds %cpyfile, it knows that there was one passed argument called 'thenum'. On the first line,

```
%cpyfile(1)
```

it takes the text characters in the parentheses () which is "1". It does not care if it is a number character or alpha character, it just grabs the characters. So macro variables are always just considered "text". At this point "1" is saved off to the side and called 'thenum'. Just like the other macro text above was saved off.

Then it goes and retrieves the saved off chunk of text called 'cpyfile' which is just as we left it, the four lines below,

```
    * copy dataset from one library to another ;  
data bbb.myfile&thenum;  
    set aaa.incomingfile&thenum;  
run;
```

See it has not changed, it is just as it was saved off. After retrieving the text it looks for % or & characters. It finds a & character with the token 'thenum' after it. So the macro parser goes and looks for saved off text called 'thenum' and it finds it. It takes all of the retrieved text and replaces all the retrieved text for the "&thenum". In this case, the 'thenum' saved text consists of just one character, "1". So when the all of the occurrences of &thenum get text substitution performed, the original

```
    * copy dataset from one library to another ;  
data bbb.myfile&thenum;  
    set aaa.incomingfile&thenum;  
run;
```

now looks like

```
    * copy dataset from one library to another ;  
data bbb.myfile1;  
    set aaa.incomingfile1;  
run;
```

Pretty tricky huh? Guess what, all that has gone on to this point is

- 1) save off chunks of text
- 2) retrieve chunks of text
- 3) replace text

See what I mean, very, very simple tasks. And easy to understand too. Just like simple building blocks that can be stacked up in a very complex design.

Therefore the code

```
%cpyfile(1)
%cpyfile(2)
%cpyfile(3)
```

turns into

```
      * copy dataset from one library to another ;
data bbb.myfile1;
    set aaa.incomingfile1;
run;

      * copy dataset from one library to another ;
data bbb.myfile2;
    set aaa.incomingfile2;
run;

      * copy dataset from one library to another ;
data bbb.myfile3;
    set aaa.incomingfile3;
run;
```

and only using the three simple tasks describe thus far. It looks just like the original code we manually wrote!

So far, just text substitution.

MAKING IT REPETITIOUS

Now, what if we had 500 of these to do?

We could manually write 500 lines like this,

```
%cpyfile(1)
%cpyfile(2)
%cpyfile(3)
%cpyfile(4)
%cpyfile(5)
%cpyfile(6)
.
.
.
%cpyfile(498)
%cpyfile(499)
%cpyfile(500)
```

and all would be performed just like we described above. So someone thought of the bright idea that if, during that first macro parser phase we could have some commands or utilities that are

- 1) *not* regular statements or functions, etc.
- 2) but kinda acted like some of the regular statements and functions.

example, what if, during the macro phase we could turn those 500 lines into a loop.

```
%macro do500times;  
    %do i = 1 %to 500;  
        %cpyfile(&i)  
    %end;  
%mend;
```

```
%do500times
```

This is not any more tricky. NOTE: USING THE SAME SIMPLE THREE TASKS ABOVE we actually have two layers to locate % and & and resolve the text substitutions. The only other thing you need to know is that in the background, invisible to you, is that the macro parser phase will look through the text as many times as it needs until all the % and &'s have been resolved. (in technical terms this actually seems like it is recursive, i.e. drilldown, or multiple passes)

So when %do500times is found, it retrieves the text saved under the name 'do500times' which looks like this,

```
%macro do500times;  
    %do i = 1 %to 500;  
        %cpyfile(&i)  
    %end;  
%mend;
```

So if the first look through this, the

```
%DO  
%TO  
%END
```

statements are a macro equivalent to the DO, TO, END statements in regular code. These % statements are only performed during the macro parser phase and ALL other non%, non& code is ignored. At this point everything is just 'text' to the macro parser.

The %do i = 1 %to 500; creates a macro variable called i, which is a saved off text chunk, and places the text character "1" off in the location called "i". Then the %cpyfile(&i) is found and the same simple tasks are performed and the &i fetches the saved off text in location "i" which is "1" and replaces the

```
    %cpyfile(&i)  
and turns it into  
    %cpyfile(1)
```

Reminder: all that is going on at this time is text-substitution!!!

Then it repeats and loads "2" into "i" and generates another resolved line of text
 %cpyfile(2)

and this repeats 500 times until it has substituted text and generated all 500 lines of text,

```
%cpyfile(1)  
%cpyfile(2)  
%cpyfile(3)
```

```

%cpyfile(4)
%cpyfile(5)
%cpyfile(6)
.
.
.
%cpyfile(498)
%cpyfile(499)
%cpyfile(500)

```

just as if you had manually written them. So that was one look through all of the text and it resolved some of the text substitution.

Now, as mentioned before, it looks through as many times as needed until all of the % and & character representing macro stuff have been resolved.

At this point we still have % characters in the

```

%cpyfile(1)
%cpyfile(2)
%cpyfile(3)
%cpyfile(4)
%cpyfile(5)
%cpyfile(6)
.
.
.
%cpyfile(498)
%cpyfile(499)
%cpyfile(500)

```

So now it goes through each %cpyfile and does it's own text replacements just as describe earlier. Then once ALL of the % and &'s have been resolved the macro parser ends up with substituted text looking like the many, many lines of text below (except it is stored internally and is invisible,

```

      * copy dataset from one library to another ;
data bbb.myfile1;
  set aaa.incomingfile1;
run;

      * copy dataset from one library to another ;
data bbb.myfile2;
  set aaa.incomingfile2;
run;

      * copy dataset from one library to another ;
data bbb.myfile3;
  set aaa.incomingfile3;
run;

.
.
.

      * copy dataset from one library to another ;
data bbb.myfile499;
  set aaa.incomingfile499;
run;

```

```
    * copy dataset from one library to another ;  
data bbb.myfile500;  
    set aaa.incomingfile500;  
run;
```

Yes, combining the three very simple tasks, it looked through the %'s and &'s twice and ended up with many, many lines of resolved text.

And then at this point all of the many lines of text are submitted for execution and really executed.

Just think: SAS-MACRO-stuff equals text-substitution

Very very simple and extremely powerful.

More specifically, it only seems that all the resolving takes place at resolution time, but many resolutions are probably done at the time the text is saved off. For the most part this is transparent to the user.

CONCLUSIONS

Hopefully you will find this way of understanding SAS Macro stuff as easy as many others have. To be able to create and troubleshoot macro code virtually always utilizes these simple concepts. You can become effective with simple macros using only what you have learned this far. The remaining learning curve for more advanced features of the SAS Macro capabilities is the typical memorizing or knowing where to look up functions and looping constructs in the documentation just like learning any other language. The basic concepts learned here carry through all the intermediate and advance facets as well. Even when advanced macro users stumble and need a second pair of eyes to troubleshoot their macro code, the concept of "text-substitution" invariably brings the solution to focus when "text-substitution" is mentally used to work through the logic, virtually every time. So when new users start learning SAS Macro stuff, they have an easier time when learning the concept of "text-substitution" first. Even intermediate and advanced users become better at using macro stuff when they consciously think through their task with "text-substitution" in mind.

ACKNOWLEDGMENTS

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.

CONTACT INFORMATION

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

Mark Terjeson

Email mterjeson@russell.com or mtnw@comcast.net