

Macro Loops from %Do to %End, or, Don't Get &&Loop&ed by Macros

Abstract

The SAS[®] macro language supports do loops analogous to those in data steps: *%do...%end*, *%do %while (condition)*, *%do %until (condition)*, and *%do i=1 %to N*. Using these loops can make macros very powerful. However, the %'s and &'s of the SAS macro language make loops there more intimidating than those in the data step. This tutorial will develop examples of the different types of macro loops, and look at the dreaded double ampersand. After working through some examples, this paper will discuss the benefits of macros and give some helpful ideas to follow when writing them.

Let's start with a simple example: *%do...%end*. This is pretty much the same as the familiar *do...end* of the SAS data step. Simply put a percent sign in front of the words "do" and "end" to tell the SAS compiler that you have macro code here rather than data step code. This macro creates a couple title statements, based on an input variable.

```
%macro maketitl(country);
  %let country=%upcase(&country);
  %if &country=CANADA %then %do;
    title1 'Data Are For Canada Only';
    title2 'In Canadian Currency';
  %end;
  %else %if %substr(&country, 1, 1)=U %then %do;
    title1 'U.S. Data Only';
    title2 'U.S. Currency';
  %end;
  %else %put ERROR: Bad value for COUNTRY: &country;
%mend maketitl;
```

This does illustrate a few things. Just as the data step has the *upcase* and *substr* functions, the SAS macro language has *%upcase* and *%substr*. There are several other macro functions analogous to those found in the data step. One difference is that the macro language does NOT use quote signs when comparing macro variables to text strings.

Let's take a look at the *%do...%while*. For an example, suppose you have several SAS data libraries. You would like everyone in your area to use the same libnames when referring to these libraries, but their physical names are hard to remember. How about a macro that will allocate the data library, given the desired libname?

```
%macro libname(librefs);
  /* put some nice documentation comments here;
  /*;
  /* Change the parameters to lower case, to avoid mismatches;
  %let librefs = %lowcase(&librefs);
  /* Set the loop counter;
  %let n = 1;
```

```

%* Create the value of THISLIB for the do-while to check;
%* The macro SCAN returns the N-th "word" of the first argument;
%let thislib = %scan(&librefs, &n);
%* The do-while condition is checked before entering the loop;
%do %while(&thislib ne %str() );
  %* Now check for all the libnames you might have;
  %if &thislib = usageog %then
    %str(libname usageog "userdata9:[br-proddata.usageog]");)
  %else %if &thislib = cangeog %then
    %str(libname cangeog "userdata9:[br-proddata.cangeog]");)
  %else %put
    Libref No. &n in the list: &thislib is unknown--not allocated;
  %* Increment the loop counter;
  %let n = %eval(&n + 1);
  %let thislib = %scan(&librefs, &n);
%end;
%mend libname;

```

A macro call like `%libname(usageog fred cangeog)` will allocate the usageog and cangeog libraries, and print a message saying that *fred* was unknown. In fact, a macro very much like this one is one of my company's most useful production macros. Recently, a production SAS data library used by 50 production programs was moved to a new disk. Because those programs used a macro to allocate the library, only the macro had to be altered, not the 50 programs.

Note that using the `%do...%while` required us to initialize the value to be checked before entering the loop, and also to update it within the loop. The `%do...%until` would not require us to initialize the value before entering the loop. However, using `%do...%until` will cause the loop to be executed at least once, which may not always be desirable. As with the analogous loops in a data step, take care to avoid "infinite loops."

```

%macro libname2(librefs);
%* Try a do-until instead of do-while ;
%*
%* Change the parameters to lower case, to avoid mismatches;
%let librefs = %lowercase(&librefs);
%* Set the loop counter;
%let n = 0;
%* No initializing of THISLIB before entering the loop;
%* The do-while condition is checked before entering the loop;
%do %until(&thislib = %str() );
  %* Increment the loop counter;
  %let n = %eval(&n + 1);
  %let thislib = %scan(&librefs, &n);
  %* Now check for all the libnames you might have;
  %if &thislib = usageog %then
    %str(libname usageog "userdata9:[br-proddata.usageog]");)
  %else %if &thislib = cangeog %then
    %str(libname cangeog "userdata9:[br-proddata.cangeog]");)
  %else %if (&thislib ne %str() ) %then %put
    Libref No. &n in the list: &thislib is unknown--not allocated;
  %end;
%mend libname2;

```

Now let's work on a do-loop with a counter. Take a very simple example:

```
%Macro veryeasy;
%* Just print a simple message 4 times;
  %do i=1 %to 4;
    %put This is iteration no. &i of the loop;
  %end;
%mend;
```

Suppose we had a series of macro variables, say, *&macvar1*, *&macvar2*, *&macvar3*, *&macvar4*. They could be global macro variables, or they could be local. For example:

```
%let macvar1=Crosby;
%let macvar2=Stills;
%let macvar3=Nash;
%let macvar4=Young;
```

How could we print the values of these macro variables in a loop?

```
%macro noteasy;
  %do i=1 %to 4;
    %put ?????
  %end;
%mend noteasy;
```

What do we put in for the question marks? Consider some (im)possibilities:

- *&macvari* won't work, because the *i* at the end of the word won't be recognized as a macro variable. Since there is no macro variable named *macvari*, the macro variable reference *&macvari* will be unresolved.
- *macvar&i* won't work, because it will be resolved to *macvar1*, *macvar2*, etc., without the crucial ampersand at the beginning. This would simply print *macvar1* instead of *Crosby* the first time through the loop. Still, the *&i* at the end is on the right track. After all, we need to get that counter involved somehow. This brings us to
- *&macvar&i*, which won't work either. The problem is that the macro compiler sees this as two macro variables strung together. The *&i* will resolve to 1, 2, 3, or 4, depending on how many times you have gone through the loop. That's exactly what you want, but the *&macvar* won't resolve to anything. Now let's look at what does work:
- *&&macvar&i*, which looks a little scary. Why this? Besides the fact that we've eliminated all the easy possibilities, consider what we are trying to do here. We need to get this macro variable reference to resolve in *two* stages. The first time through the loop, we want to print out *Crosby*. That means we want the macro compiler to see the macro variable *&macvar1*, without actually hardcoding the '1' at the end of the word. This means that we have to code something which will *resolve* to *&macvar1*, which in turn resolves to *Crosby*. We need a macro "token" which refers to the macro variable *&macvar1*. Why does our *&&* work? On the first pass, the macro compiler will resolve the *&i* to the proper iteration in the loop, while the double ampersand is resolved to a single ampersand. The first time through the loop, then, *&i*→1 and *&&macvar*→*&macvar*.

Thus, the macro compiler resolves `&&macvar&i` → `&macvar1` → *Crosby*, as desired. Our macro is

```
%macro noteasy;
  %do i=1 %to 4;
    %put &&macvar&i;
  %end;
%mend;
```

With the global macro variables `&macvar1` - `&macvar4` defined earlier, the macro call `%noteasy` will indeed print

```
Crosby
Stills
Nash
Young
```

What we have just seen is an indirect macro variable reference. Indirect references are by nature complicated, but we have a very powerful tool here, especially if used with the *call symput* command. Consider the following example:

```
data _null_;
  length num $ 4;
  input person $;
  num = put(_n_, 4.);
  call symput('total', num);
  call symput('name' || left(num), person);
  cards;
Crosby
Stills
Nash
Young
;

%macro whatever;
%do i=1 %to &total;
  %put Person No. &i is &&name&i;
%end;
%mend;
```

Look at the `_null_` data step in the preceding example. Besides creating the macro variables `&name1` - `&name4`, it also created `&total`, which spared us from having to know in advance how many "name" macro variables there were. The other *call symput* requires a little care. The first argument to the *call symput* must a valid SAS macro variable name. Among other things, it cannot have any spaces in it.

Now consider a more realistic example: You wish to print personalized reports for an unknown number of people. Suppose we have a SAS dataset with a variable named *person* in it. One possibility might be to print a report with a by-line with the variable *person* in it, but in this case that won't do. We need to send each person's report to a separate output file, where it can be sent

to the person using electronic mail. Our company uses this technique to send out about 400 personalized reports per month, automatically.

```
%macro personal(indata=);
/* Start by getting a list of names in the data ;
proc summary nway data=&indata;
  class person;
  output out=people(drop=_type_ _freq_);
/* The dataset PEOPLE has one observation per person */
/* As before, create &total and "name" macro variables */
data _null_;
  set people;
  num = put(_n_, 4.);
  call symput('total', num);
  call symput('name' || left(num), person);
run;
/* Now create one report for each person. First, use ;
/* PROC PRINTTO to send each report to its own file. ;
%do i=1 %to &total;
  proc printto new print="&&name&i.dat";
  proc report data=&indata;
    where person="&&name&i";
    title1 "Report for &&name&i";
    /* additional code for report goes here */
  %end;
/* Now route output back to the default file ;
  proc printto;
  run;
%mend personal;
```

That should do it. We are assuming here that the names have no blanks or weird characters in them. In our case, we use 8-character mail codes, so that is not a problem.

To sum up, macro loops are basically the same as loops in data steps. We can use indirect references to macro variables inside of loops. A powerful technique is to use *call symput* in a data step to create macro variables, which then can be referenced within a macro loop.

Is it worth the trouble to master the double ampersands? The truth is that macros have many benefits:

- Using a macro to handle a task done by many programs means that only the one macro has to be maintained instead of the many programs.
- By having everyone use the same macros, you guarantee that everyone uses the same variable names, labels, and computational methods.
- Macros can be programmed to be extremely flexible.
- Debugging is done when the macro is written. A programmer can regard a macro as already-debugged code.

Some Final Hints for Writing Macros

- Any time you see a task, even a simple one, that must be performed many times, write a macro to do it.
- When debugging macros, use *options mprint* to check on the SAS code generated by the macro, and *options mlogic* to trace the execution of the macro code itself.
- Make your macros flexible! Input and output dataset names should be macro parameters that user can determine. Provide lots of macro parameters that the user can control.
- Where possible, provide sensible default values for macro parameters.
- Document your macros. Comments at the beginning of each macro should describe what the macro does, how to use it, and the meaning of each of the macro parameters. That way the user can see how to use the macro just by looking at the macro code. We also have documentation stored on-line, in using a popular word processor.
- Include comments within your macro code, too, so a person doing maintenance on your code (such as yourself!) will know what the code is doing. This is often more important for macro code than ordinary SAS code.
- Note that you can use the %sysexec function to issue operating system commands and even check their return codes. You can even check what operating system you're in, and issue the appropriate command based on that. We use this capability to do such things as uploading data to our mainframe and submitting batch jobs on it.

SAS is a registered trademark of SAS Institute Inc. in the USA and other countries.

® denotes USA registration.

Contact information:

Roger Konyndyk CD-5W-08
Steelcase Inc.
P.O Box 1967
Grand Rapids, MI 49501-1967

Telephone: (616) 698-4518
FAX: (616) 698-4502