ABSTRACT
Some people love to use macros while other people would love to see them eliminated from SAS® altogether and replaced by the use of SCL or CALL EXECUTE. Part of the negativity about macros stems from poorly written macros or using them when they aren’t really beneficial. This paper will cover some examples of “bad” macros, why they would be considered “bad” and how to improve them. This paper is focused on using macros within individual programs. It does not cover writing a system of macros.

INTRODUCTION
Macros can be very helpful especially in the pharmaceutical industry. However, I’ve also seen some that have given me a big headache. Macros are great when you have repetitive work. For example, you need to create a demographics table for all screened subjects, all randomized subjects, all safety subjects and all intent-to-treat subjects. Perhaps you need to have identical tables for different variables such as systolic blood pressure and diastolic blood pressure. Maybe your identical tables are split up by visits. Why write the same code multiple times in different programs? If you need to make a change, even something as simple as changing from 1 to 2 decimal places, it is easier to do it in one place instead of several places. I believe my record is one program that created 128 tables.

These tips are based on efficiency. Efficiency can be viewed in several ways.

Original programmer – How long does it take to write the program the first time?

Run time – How long does it take to run? Even if the program is for a one-time study, you’ll still need to run multiple times. At the very least, most companies will have the initial development run time, rerunning after reviewer’s changes are fixed, and the final production run so everything has the same date/time stamp. In many cases, you’ll be running several other times because you’ll have one or more versions of blinded, draft data followed by unblinded final data.

Updates - There may be a need for additional changes later due to reviews by other project team members or new data. The original programmer may not be available. This is especially true for long-term studies where there can be months or years between the initial programming and the need for updates. Even if the original programmer is doing the updates, it isn’t always easy to remember what you did months ago.

These tips are based on things I’ve seen in the 20 years I’ve used macros. The examples come from real programs with the code modified to show a general example. The libname PERM is used to represent datasets that are permanent datasets. No libname is used for datasets that will only exist within your program.

You may have programming practices in place at your company that might conflict with these tips. By all means, you need to follow your company practices. However, if you don’t know the reasoning behind your company practice, this would be a good reason to ask. Sometimes practices are simply hold-overs that no longer apply to the current software/hardware/network situation while others have a valid justification.
TIP #1 Don’t turn an entire program into a macro

Let’s say you’ve written the following code.

```sas
data demographics;
  set perm.demographics;
  . . . some code here . . .
run;

ods . . . table1.rtf;
proc report . . .;
run;
```

This will give you all subjects in the study. Now you need the exact same table except for SAFETY, RANDOMIZED and ITT subjects. You might be tempted to do this.

```sas
%macro my_table (table=, where=);
data demographics;
  set perm.demographics;
  &where;
  . . . some code here . . .
run;

ods . . . &table..rtf;
proc report . . .;
run;
%mend my_table;

%my_table(table=1);
%my_table(table=2, where=where SAFETY=1);
%my_table(table=3, where=where RAN=1);
%my_table(table=4, where=where ITT=1);
```

While it does run and it may have been efficient for you to write, it isn’t efficient from a run time perspective. You don’t want to read in permanent datasets more than once. For a small study, this might take only a couple seconds. However, the more subjects, the longer it will take. Where your permanent data are stored can also make a big impact especially if you are working on a pc and the data are stored on a server. For example, let’s say it takes 5 minutes to read in the permanent dataset and another minute to do the rest of the processing. To create these 4 tables as shown above will take at least 24 minutes every time to run – even if all you were doing was updated a typo where you spelled “Week” as “Wekk”.

Here’s a better way that will get your run time down to about 9 minutes for the same 4 tables.

```sas
data all_data;
  set perm.demographics;
run;

%macro my_table (table=, where=);
data demographics;
  set all_data;
  &where;
  . . . some code here . . .
run;

ods . . . &table..rtf;
proc report . . .;
run;
%mend my_table;

%my_table(table=1);
%my_table(table=2, where=where SAFETY=1);
%my_table(table=3, where=where RAN=1);
%my_table(table=4, where=where ITT=1);
```

Here’s a better way that will get your run time down to about 9 minutes for the same 4 tables.
You might not think saving 15 minutes is a big deal but when you are trying to meet a deadline it matters. We all have deadlines even if it is just the deadline of leaving work on time! Earlier I noted my record was a single program that created 128 tables. That program used a large dataset that really did take about 5 minutes to read in each time. It took about another 4 minutes to do the processing. Had I used individual programs or turned the entire program into a macro, it would have required 128*5 minutes just to read in data – that's 640 minutes or 10.7 hours! My program had other efficiencies and usually took under 30 minutes to do those 128 tables. This particular study had 64 lab tests and each lab test was reported in SI and standard units. Some companies do one very large table with all the tests. In this case, there were several visits being reported so there was a separate table number for each test.

**TIP #2 Use named parameters rather than positional parameters**

I’m in the group that used positional parameters for many years. However, I rarely had more than 3 or 4 parameters and all parameters were required. My possible values were also very distinctive. After I encountered a macro like:

```
%make_flags(flag,5,a,b,d,,c);
```

I realized the benefit of named parameters and have very rarely used positional parameters since then. What if you really needed to

```
%make_flags(flag,5,a,b,,d,c);
```

Two characters were accidentally switched but it could make a big difference in the outcome.

```
%make_flags(varname=flag,pos1=a,pos2=b,pos4=d,pos5=c);
```

makes it very obvious that you aren't passing a parameter for pos3. Named parameters also mean you don't need to have the parameters in a given order. Especially when you are doing several macro calls together, it can be beneficial to put the parameters in a different order to make it easier to read and check your work.

**TIP #3 Use meaningful variable/parameter names**

Some people have been led to believe that variable names can only be 8 characters. I’ve even seen “good programming practices” documents make that statement. This was true in the early days of SAS. For those of us who started out on using a keypunch, 8-character limits were fine. Once a full-screen environment became standard and other languages used longer variable names, most people saw a benefit and SAS introduced longer variable names that can now be 32 characters. However, SAS still has a limit of 8 characters if you will be creating Version 5 transport files. If you’ve even sent anything to the FDA, you know about Version 5 transport files. So you are still limited to 8-character names in your permanent dataset that will become a transport file. However, that limit doesn’t need to be imposed on your temporary datasets.

Here’s an example of how short variable names can cause problems especially for someone taking over existing code.

```
%macro my_table(where=,a=,b=,c=);
```

The programmer needed to change the visits to exclude unscheduled visits. Most people would probably think it is the WHERE that would need to be changed and perhaps not look too closely at the rest of the code. In this case, the programmer was fortunate that the program didn’t run so the entire program had to be reviewed more closely. It turned out there really were two different WHERE clauses that were used at different places within the code. There was a sound reason for having the two clauses given the data. What wasn’t as reasonable was the fact that the second WHERE clause came from the macro variable C. Perhaps the use of C was logical to the original programmer but I doubt if most people would find it logical. Even looking at the actual call with the parameters filled in didn’t make the use of C obvious. A and B happened to be variables that were being passed so it looked like C was as well. A better choice would have been
An even better choice would be

```
%macro my_table(where_main=,where_visit=,a=,b=);
```

Using WHERE1 and WHERE2 at least lets you know there are two different WHERE statements prompting you to look for the distinction in their use. Using something like WHERE_MAIN and WHERE_VISIT not only tells you there are two different statements but gives you a better idea how they are used and how they differ. The first is probably on the MAIN dataset and the second is used at a place restricting to visits. Even these names may still force you to read the code to know exactly what they really do. Notice that the order of the parameters was also switched so the related parameters were together.

**TIP #5 Ensure datasets are not being carried over from an earlier macro call**

Over the years I've seen many people have this problem and almost all of them noticed they had a problem. The hard part was figuring out how to fix it. I've usually seen this happen with p-values. If the subset doesn't have sufficient records/combinations to be able to calculate the p-values, the dataset will not be written out. Here is a very simple example.

```
%macro run_stats(where=);
    proc freq data=one;
        &where;
        table x*y/chisq;
        ods output chisq=chisq;
        run;
    . . . rest of code for table . . .
%mend;
```

```
%run_stats(where= itt=1);
%run_stats(where= itt=1 and age>55);
```

What could go wrong? If you don’t have many subjects over 55, you might end up with the following message in the log:

```
No statistics are computed for x * y since x has less than 2 nonmissing levels.
WARNING: Output 'chisq' was not created.
```

However, your CHISQ dataset from the first macro call is still present and will be used for the rest of the processing. There are several ways around this.

**Method 1: Delete the datasets as the last step in the macro**

This code will delete all the datasets.

```
proc datasets library=work kill memtype=data;
quit;
```

If you follow the first tip and want to keep certain datasets, you just need to change PROC DATASETS to SAVE the desired datasets. All other datasets will be deleted

```
proc datasets library=work memtype=data;
    save main_dataset;
quit;
```
Method 2: Make all dataset names unique

This method makes all the datasets have a unique name by adding a unique macro variable value to the name.

```sas
%macro run_stats(table=,where=);
  proc freq data=one;
    &where;
    table x*y/chisq;
    ods output chisq=chisq&table;
    run;
  . . . rest of code for table . . .
%mend;

%run_stats(table=1,where= itt=1);
%run_stats(table=2,where= itt=1 and age>55);
```

Method 3 – Conditional code

This method depends on the criteria that must be met to have the proc create the dataset. Exactly how you write the code will also be dependent upon what steps you need to take when the proc can not create the dataset. You would write code that checks if your dataset met the criteria and creates a macro variable. Then you would use code such as:

```sas
%if &condition_met=yes %then %do;
  . . . code to run the proc . . .
%end;
%else %do;
  . . . some other code if the condition is not met . . .
%end;
```

This method is more complex but does have the advantage of eliminating the warning messages in your log.

**TIP #6 Don't use macros at all when other code will work**

Let's say you are using lab data with one record per lab test. Instead of using a macro and calling each lab test one at a time, a BY statement will be more efficient for most procedures. There is some overhead each time you run a PROC. With the BY statement, you are only running the PROC once.

Let's say you have a dataset with 3 simple variables A, B and C. You want a frequency for B*C for each value of A. You could write it as a macro

```sas
%macro repeat(val=);
  proc freq data=one(where=(a=&val));
    tables b*c/noprint out=out&val;
  run;
%mend repeat;

%repeat(val=1);
%repeat(val=2);
```

This might not look too bad but what if you had many values of A? I created some very simple sample data that had 100 values for A. Each macro call took about .03 seconds. So for 100 calls, it would take about 3 seconds. However, I did all 100 values at once with

```sas
proc freq data=ones;
  by a;
  tables b*c/noprint out=outall;
run;
```

All 100 values were done in .5 seconds. That’s 1/6th of the time for the individual calls. This may not be a noticeable time savings but when you use larger datasets you will have a much bigger impact. My sample data also made it easy to call the different values. You could also write a %DO loop to go through all 100 values instead of writing them out individually. Most of the time you probably won’t be that lucky and it will take you longer just to write out all those calls. In addition, by using the BY statement, you won’t overlook any possible values of your BY variable that you didn’t know where in the data. This is especially helpful when you write your code with partial data but run it later with more data. You can build in checks to see if you get any unexpected values.
You should also notice that I only ended up with one output dataset that all my counts in it instead of 100 individual datasets. You won’t need to put the individual datasets back together again for additional processing or use additional macros.

If you need to write individual RTFs for each value of your BY variable, you can still gain the benefits from this method. The only part that would need to go into a macro call would be the creation of the RTF.

I’ve also found formats as a way to eliminate macro code and make a program more readable. I’ve seen many lab programs that have code like:

```%if &table=1 %then %do;
  if labtest='CALC' then sort_order=1;
  else if labtest='ALB' then sort_order=2;
  ...%end;
%else %if &table=2 %then %do;
  if labtest='RBC' then sort_order=1;
  else if labtest='WBC' then sort_order=2;
  ...%end;```

The more lab tests, the more difficult this code is to read. An much cleaner method would be to create a format that maps the labtest to the sort order value. Your code then simply becomes:

```proc format;
value $labsort
  'CALC'='01'
  'ALB'='02'
  'POT'='20'
  'RBC'='01'
  'WBC'='02'
  MON='17';
run;
data labs
  ...sort_order=put(labtest,$labsort);
run;```

Not only do you not have %IF %THEN code cluttering up you program, you have a lot less typing with the proc format and can easily switch the order if you need it.

**TIP #7  Don’t go overboard with the use of macros**

Does your code include things like &&&var&i&j or lots of %if logic? If so, you may have gone overboard. If you are writing a system that will be used for all studies and needs to be very flexible, then your code might need so many ampersands. However, most people take longer to write and debug such code so you often lose the benefits you were trying to gain. You need to weigh the extra time to write the complex macro code versus having separate programs. If someone else has to take over the code, it will probably be harder for that person to make even a simple change.

Using macros for repeat tables is efficient but what is really is a “repeat”? If the tables look identical but use different datasets, they really are not good candidates for using a macro. A major factor is that variable names will be different. Almost every time I’ve seen someone do this, there have been changes later than ending up taking more time. Those changes may be that one table needs a change that doesn’t apply to tables from other datasets so you end up with %if &dataset=xxx %then %do code to customize the one table. It might also be that a single dataset was updated. Since the program runs several tables, you have to run all the tables again just because one dataset was updated. I had a case like this and it took over an hour to rerun the tables since the program used several different datasets. Had the tables be split into separate programs, only the ones using the updated dataset would have been run and we could have finished in less than 10 minutes. You might consider creating a macro for just the common portion of your code and calling it from your separate programs. This has the advantage of ensuring the repeating
part is identical and only written once yet gives you the flexibility gained by separate programs.

CONCLUSION
Macros are very beneficial for repeating tables. If you write them properly, then can provide many efficiencies. This is the text for the paper's conclusion.

RECOMMENDED READING
There are several books on macros available from SAS Publishing. You can also search conference papers from PharmaSUG and other conferences at www.lexjansen.com. Over 873 results came back by searching for "macros". Another great source for learning about macros is SAS-L. This is an electronic, world-wide user group that lets you ask questions of other SAS users and search the archives. There are several ways to access SAS-L and you can look more about them by searching for conference papers on the topic.

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

- Deb Cassidy
- PPD, Inc.
- 3900 North Paramount Parkway
- Morrisville, NC 27560
- Work Phone: 919-456-5115
- Email: deborah.cassidy@ppdi.com
- Web: www.ppdi.com

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 trademarks of their respective companies.