Paper CC-002

A Macro to Unravel Macros

Sarah Woodruff, Westat, Rockville, MD

ABSTRACT

Do you find it tedious to pinpoint problems in your macro code? Are you annoyed when the log points to the line number of the macro call rather than the code in the macro that generated the error? Would you like a little help? You want to produce macros that are utilitarian black boxes, but you need to understand what the code is actually doing as part of the development process, and that can be frustrating.

This paper presents a simple wrapper macro that is used to enclose existing code, which can call any number of other macros. When you use this technique along with the mPrint and mFile options, the output file produced by mPrint/mFile contains a completely "de-macroed" version of the program. You can then examine this output as a working program, thus making it easier to both understand and edit.

Keywords: macro, mPrint, mFile, debugging

INTRODUCTION

Debugging SAS® code is challenging enough when it is a collection of procedures and data steps. The task becomes that much more challenging when the code contains macros. This paper will present a short macro that simplifies the process of working on macro code.

THE PROBLEM

Whether you are writing macro code exclusively for your own use or working on a macro to be used by many people, it is important for the finished product to work smoothly and as intended. Normally, the SAS log is the one of the most crucial tools in debugging, providing descriptions of errors and other potential problems. Even when those descriptions seem obtuse, they at least provide a reasonable starting point for solving the problem. This is not the case when attempting to debug macros. Compare the log messages given when the same data manipulation is performed both outside and within a macro.

Let's assume that subjects are being enrolled into a study and we are going to want to look at several subsets of those subjects: by the ID number assigned at enrollment, by the date on which enrollment took place, by age and specifically everyone enrolled within the last ten days.

```
DATA enrollment;
        INPUT ID $ Enroll $ Age $;
        DATALINES;
        100 20090316 24
        101 20090316 18
        102 20090317 17
        103 2009     21
        104 20090323 22
        105 20090324 16
        106 20090325 16
        107 20090325 19
        108 20090326 20
        ;
RUN;
```

These subsets could be created through the use of data steps:

```
/*Enter parameters to create desired subsets of enrollment data based on ID, date or age*/
DATA enroll_id enroll_date enroll_age;
    SET enrollment;
                Enroll_Num = input (Enroll,YYMMDD8.);
                IF 100 le ID le 105 THEN OUTPUT enroll_id;
                IF mdy(3,23,2009) le Enroll_Num le mdy(3,28,2009) THEN OUTPUT enroll_date;
                IF 16 le Age le 20 THEN OUTPUT enroll_age;
```

```
RUN;
/*Create subset of all subjects enrolled in the last ten days*/
DATA enroll_range;
       SET enrollment;
              WHERE enroll gt (today() - 10);
RUN;
```

However, since the subsetting of this data will need to happen routinely, it makes sense to write a macro to do the work. In this way, the data ranges can be changed without having to edit the source code.

```
%macro Enroll_Subset (first_id, last_id, first_month, first_day, first_year,
                      last_month, last_day, last_year, first_age, last_age);
       DATA enroll_id enroll_date enroll_age;
       SET enrollment;
                     Enroll_Num = input (Enroll,YYMMDD8.);
                     IF &first_id le ID le &last_id THEN OUTPUT enroll_id;
                     IF mdy(&first_month,&first_day,&first_year) le Enroll_Num le
                            mdy(&last_month,&last_day,&last_year) THEN OUTPUT enroll_date;
                     IF &first_age le Age le &last_age THEN OUTPUT enroll_age;
       RUN;

       DATA enroll_range;
              SET enrollment;
                     WHERE enroll gt (today() - 10);
       RUN;
%mend Enroll_Subset;

%Enroll_Subset (100,105,3,23,2009,3,28,2009,16,20);
```

Now let's examine the log for both of these pieces of code being run.

In the scenario where the data steps have the ranges hard-coded, the log looks like this:

```
16   /*Enter parameters to create desired subsets of enrollment data based on ID, date or age*/
17   DATA enroll_id enroll_date enroll_age;
18       SET enrollment;
19           Enroll_Num = input (Enroll,YYMMDD8.);
20           IF 100 le ID le 105 THEN OUTPUT enroll_id;
21           IF mdy(3,23,2009) le Enroll_Num le mdy(3,28,2009) THEN OUTPUT enroll_date;
22           IF 16 le Age le 20 THEN OUTPUT enroll_age;
23   RUN;

NOTE: Character values have been converted to numeric values at the places given by:
(Line):(Column).  20:19   22:18
NOTE: Invalid argument to function INPUT at line 19 column 22.
ID=103 Enroll=2009 Age=21 Enroll_Num=. _ERROR_=1 _N_=4
NOTE: Mathematical operations could not be performed at the following places. The results of the
operations have been set to missing values. Each place is given by: (Number of times) at
(Line):(Column). 1 at 19:22
NOTE: There were 9 observations read from the data set WORK.ENROLLMENT.
NOTE: The data set WORK.ENROLL_ID has 6 observations and 4 variables.
NOTE: The data set WORK.ENROLL_DATE has 5 observations and 4 variables.
NOTE: The data set WORK.ENROLL_AGE has 6 observations and 4 variables.
NOTE: DATA statement used (Total process time):
      real time           0.46 seconds
      cpu time            0.04 seconds
24   /*Create subset of all subjects enrolled in the last ten days*/
25   DATA enroll_range;
```

```
26       SET enrollment;
27           WHERE enroll gt (today() - 10);
ERROR: Where clause operator requires compatible variables.
28   RUN;
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.ENROLL_RANGE may be incomplete.  When this step was stopped there were
0 observations and 3 variables.
```

First, we are given line and column references for the automatic character to numeric conversion that has taken place in order for the ID and Age comparisons to work. Though this may seem convenient, it is typically better to not let SAS take this shortcut, as it could produce unexpected results with unanticipated data. It is better for conversions to be done explicitly, and these log messages show us precisely where that is happening so we know where to target the fix.

Second, this log very explicitly tells us that the incoming data contains a "bad" value; in this case, that takes the form of an enrollment date that contains only a year rather than a year, month and day. This example data set was kept small for demonstration purposes, but this could just as easily have been applied to hundreds or millions of records where picking out such a value by sight would not have been practical.

Third, when the WHERE clause runs into a data compatibility issue, the ERROR message is located directly below the location in the code where the problem takes place. This facilitates being able to examine the data and make the necessary changes to achieve a compatible comparison.

Let's now compare this to the log messages received when the macro is run:

```
31   %macro Enroll_Subset (first_id, last_id, first_month, first_day, first_year,
32                         last_month, last_day, last_year, first_age, last_age);
33       DATA enroll_id enroll_date enroll_age;
34           SET enrollment;
35               Enroll_Num = input (Enroll,YYMMDD8.);
36               IF &first_id le ID le &last_id THEN OUTPUT enroll_id;
37               IF mdy(&first_month,&first_day,&first_year) le Enroll_Num le
38                   mdy(&last_month,&last_day,&last_year) THEN OUTPUT enroll_date;
39               IF &first_age le Age le &last_age THEN OUTPUT enroll_age;
40       RUN;
41
42       DATA enroll_range;
43           SET enrollment;
44               WHERE enroll gt (today() - 10);
45       RUN;
46   %mend Enroll_Subset;
47
48   %Enroll_Subset (100,105,3,23,2009,3,28,2009,16,20);

NOTE: Character values have been converted to numeric values at the places given by:
(Line):(Column).  1:144   2:113
NOTE: Invalid argument to function INPUT at line 1 column 91.
ID=103 Enroll=2009 Age=21 Enroll_Num=. _ERROR_=1 _N_=4
NOTE: Mathematical operations could not be performed at the following places. The results of the
operations have been set to missing values. Each place is given by: (Number of times) at
(Line):(Column).  1 at 1:91
NOTE: There were 9 observations read from the data set WORK.ENROLLMENT.
NOTE: The data set WORK.ENROLL_ID has 6 observations and 4 variables.
NOTE: The data set WORK.ENROLL_DATE has 5 observations and 4 variables.
NOTE: The data set WORK.ENROLL_AGE has 6 observations and 4 variables.
NOTE: DATA statement used (Total process time):
      real time           0.18 seconds
      cpu time            0.12 seconds
ERROR: Where clause operator requires compatible variables.
```

```
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.ENROLL_RANGE may be incomplete.  When this step was stopped there were
0 observations and 3 variables.
WARNING: Data set WORK.ENROLL_RANGE was not replaced because this step was stopped.
```

First, the resolved macro code is not passed to the log so it is not possible to directly see what actual values are being used in the specified ranges.

Second, though the text of the notes, errors and warnings are basically the same, they do not give the specific locations of the problem discussed. Rather, they reference the line in which the macro is called. While this is less of an issue in a shorter piece of code like this, it becomes much more problematic the longer the macro becomes. Even with something shorter, it is still much more useful to have the location of the problem pinpointed. This same issue is true with the ERROR message regarding the WHERE clause incompatibility. This example has only one such clause, but if it had more, this would be that much more frustrating.

The information given on the same step is much less specific and helpful when the manipulation is performed within a macro. Along with the lack of comparable log messages, the same code is not returned to the log and thus the macro ends up appearing as a "black box". This makes it challenging to determine whether the problem is with the data or whether is it in the macro code itself.

POTENTIAL TOOLS

There are certain tools and options within SAS to aid in the solving of problems within macro code with varying degrees of utility.

The use of %PUT is one example of a coding tool. By including a %PUT statement along with a macro variable of interest, the value to which it resolves will appear in the log. This can also be used to print specific messages to the log based on conditions met when the program executes. This would allow you to ensure that correct values are being passed through the macro and to check on whether steps are executing as expected.

A useful system option for macro debugging is mPrint. By specifiying `options mprint;` before running the macro, the lines of resolved code are printed in the log. Here is an excerpt from the example used above:

```
MPRINT(ENROLL_SUBSET):   DATA enroll_id enroll_date enroll_age;
MPRINT(ENROLL_SUBSET):   SET enrollment;
MPRINT(ENROLL_SUBSET):   Enroll_Num = input (Enroll,YYMMDD8.);
MPRINT(ENROLL_SUBSET):   IF 100 le ID le 105 THEN OUTPUT enroll_id;
MPRINT(ENROLL_SUBSET):   IF mdy(3,23,2009) le Enroll_Num le mdy(3,28,2009) THEN OUTPUT
enroll_date;
MPRINT(ENROLL_SUBSET):   IF 16 le Age le 20 THEN OUTPUT enroll_age;
MPRINT(ENROLL_SUBSET):   RUN;
```

While this does show the code with fully resolved macro variables, it does not give any further information on the line numbers in which an error is occurring or give us the ability to run this resolved code on its own. Thus it aids in a visual inspection for the source of errors, but that may not be sufficient depending on the complexity of the problem.

If you were interested in seeing the resolution of each macro variable printed to the log, specifying `options symbolgen;`will do this. Without potentially cluttering the code in the log, this is an easy way to track each of the values being passed through. This is how the SYMBOLGEN output appears for the above macro code:

```
SYMBOLGEN:  Macro variable FIRST_ID resolves to 100
SYMBOLGEN:  Macro variable LAST_ID resolves to 105
SYMBOLGEN:  Macro variable FIRST_MONTH resolves to 3
SYMBOLGEN:  Macro variable FIRST_DAY resolves to 23
SYMBOLGEN:  Macro variable FIRST_YEAR resolves to 2009
SYMBOLGEN:  Macro variable LAST_MONTH resolves to 3
SYMBOLGEN:  Macro variable LAST_DAY resolves to 28
SYMBOLGEN:  Macro variable LAST_YEAR resolves to 2009
SYMBOLGEN:  Macro variable FIRST_AGE resolves to 16
SYMBOLGEN:  Macro variable LAST_AGE resolves to 20
```

Some of the most detailed information can be gleaned with the use of `options mLogic;` because it sends to the log information on how each step of the macro is executed. This allows you to see information not only on how each variable was resolved, but also whether evaluated conditions are true or not, the number of iterations of any loops and the start and end of each execution. mLogic is particularly useful when debugging instances of nested macros, where the code of one macro references another; however, it can generate a considerable amount of output.

While each of these can shine light on a different aspect of the puzzle or provide different types of information, all still leave you working within the framework of the macro and its limited log messages. Instead, how about solving the problem by leaving the macro call behind yet leaving the macro code intact?

A MORE COMPLETE SOLUTION

One system option useful in debugging that was not discussed above is mFile as this sends the resolved macro code to a separate file. While the techniques discussed earlier can simply be specified in an options statement and can be used alone or together, mFile must be used in conjunction with mPrint and requires the specification of a fileref to have a place to save the resolved output. Though extremely useful, a limitation with the use of mFile is that only the resolved macro code is sent to the output file. Since a program usually contains more than just macros and since those other data steps and/or procs may well be part of the problem, it would be useful to see all of the code, including the resolved macro, in one complete program.

The more complete solution comes in the form of the following macro wrapper. You may be thinking, “My problem lies in dealing with macro code. How can adding more macro code be helpful?” The macro wrapper is simple enough to not add further complication and well worth the few extra lines of code in terms of the information it can yield:

```
filename mPrint "drive\path\fullprogram_nomacros.sas";
options mPrint mFile;

%macro simplify();
       %include "drive\path\fullprogram_withmacros.sas";
%mend simplify;

%simplify();
```

When this wrapper code is run on the example macro discussed above, the resulting program code sent to the output file looks like this:

```
DATA enrollment;
INPUT ID $ Enroll $ Age $;
DATALINES;
;
RUN;
DATA enroll_id enroll_date enroll_age;
SET enrollment;
Enroll_Num = input (Enroll,YYMMDD8.);
IF 100 le ID le 105 THEN OUTPUT enroll_id;
IF mdy(3,23,2009) le Enroll_Num le mdy(3,28,2009) THEN OUTPUT enroll_date;
IF 16 le Age le 20 THEN OUTPUT enroll_age;
RUN;
DATA enroll_range;
SET enrollment;
WHERE enroll gt (today() - 10);
RUN;
```

Other than the loss of indentation, this code is the same as the non-macro version of the program described at the beginning. It will produce the same log messages with their same level of detail regarding lines, columns and placement of errors. Based on that information, changes can be made to the macro, to other data steps or procs or even to the data itself as needed. The wrapper macro program can be rerun as edits are made to the original program in order to continue the debugging process and ensure that the output is correct.

Though this does require an extra step, the crucial difference here is the use of the %INCLUDE; by using this statement to incorporate the program of interest, mFile acts on all of the code rather than just on the macros that are part of the program. In this way, every piece of the program gets sent to the new file, thus facilitating the process of looking at how each interacts with the others.

By having this much more detailed log information the following can be seen:

1. There is an incomplete enrollment data for subject 103. That will need to be corrected.
2. The data for ID and Age are coming in as character data. Since comparisons are desired based on ranges of those variables, it would make sense to explicitly convert them to numeric variables.
3. The WHERE condition needs to utilize the numeric version of the date.

Below would be one way to accomplish these changes:

```
%macro Enroll_Subset (first_id, last_id, first_month, first_day, first_year,
                                    last_month, last_day, last_year, first_age, last_age);

        DATA enrollment_revised;
                SET enrollment;
                        ID_Num = input (ID, 3.);
                        Enroll_Num = input (Enroll,YYMMDD8.);
                        Age_Num = input (Age, 3.);
        RUN;

        DATA enroll_id enroll_date enroll_age;
        SET enrollment_revised;
                        Enroll_Num = input (Enroll,YYMMDD8.);
                        IF &first_id le ID_Num le &last_id THEN OUTPUT enroll_id;
                        IF mdy(&first_month,&first_day,&first_year) le Enroll_Num le
                                mdy(&last_month,&last_day,&last_year) THEN OUTPUT enroll_date;
                        IF &first_age le Age_Num le &last_age THEN OUTPUT enroll_age;
        RUN;

        DATA enroll_range;
                SET enrollment_revised;
                        WHERE Enroll_Num gt (today() - 10);
        RUN;
%mend Enroll_Subset;

%Enroll_Subset (100,105,3,23,2009,3,28,2009,16,20);
```

When this code is passed back through the %SIMPLIFY macro wrapper and that fully resolved program is run, the log now looks like this:

```
1    DATA enrollment;
2    INPUT ID $ Enroll $ Age $;
3    DATALINES;
NOTE: The data set WORK.ENROLLMENT has 9 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           0.98 seconds
      cpu time            0.03 seconds
13   ;
14   RUN;
15   DATA enrollment_revised;
16   SET enrollment;
17   ID_Num = input (ID, 3.);
18   Enroll_Num = input (Enroll,YYMMDD8.);
19   Age_Num = input (Age, 3.);
```

```
20   RUN;
NOTE: There were 9 observations read from the data set WORK.ENROLLMENT.
NOTE: The data set WORK.ENROLLMENT_REVISED has 9 observations and 6 variables.
NOTE: DATA statement used (Total process time):
      real time           0.06 seconds
      cpu time            0.01 seconds
21   DATA enroll_id enroll_date enroll_age;
22   SET enrollment_revised;
23   IF 100 le ID_Num le 105 THEN OUTPUT enroll_id;
24   IF mdy(3,23,2009) le Enroll_Num le mdy(3,28,2009) THEN OUTPUT enroll_date;
25   IF 16 le Age_Num le 20 THEN OUTPUT enroll_age;
26   RUN;
NOTE: There were 9 observations read from the data set WORK.ENROLLMENT_REVISED.
NOTE: The data set WORK.ENROLL_ID has 6 observations and 6 variables.
NOTE: The data set WORK.ENROLL_DATE has 5 observations and 6 variables.
NOTE: The data set WORK.ENROLL_AGE has 6 observations and 6 variables.
NOTE: DATA statement used (Total process time):
      real time           0.06 seconds
      cpu time            0.04 seconds
27   DATA enroll_range;
28   SET enrollment_revised;
29   WHERE Enroll_Num gt (today() - 10);
30   RUN;
NOTE: There were 0 observations read from the data set WORK.ENROLLMENT_REVISED.
      WHERE Enroll_Num>(TODAY()-10);
NOTE: The data set WORK.ENROLL_RANGE has 0 observations and 6 variables.
NOTE: DATA statement used (Total process time):
      real time           0.03 seconds
      cpu time            0.01 seconds
```

There are no more warnings or errors and all the notes are giving expected information that indicates the program is working smoothly.

CONCLUSION

The SAS log is typically the “go-to” source for information on why code is not working as expected. It can give excellent information on the location and type of problem being encountered. However, a straightforward log is not of much help when attempting to debug macro code. There are both code tools and system options which can facilitate the process, but each of them utilizes the log as the window through which to view the information they provide. The macro wrapper program described here combines the utility of the mPrint and mFile options with %INCLUDE to create a small additional step that places all the program code, not just the resolved macro code, in the newly created file. By having all the pieces of the program resolved and in one active program, it is easier to specifically identify and correct problems.

ACKNOWLEDGMENTS

I would like to thank Ed Heaton for his inspiration and ongoing involvement. I would also like to thank Mike Rhoads at Westat for his guidance and input as well as my employer for its institutional support.

CONTACT INFORMATION

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

Sarah Woodruff
Westat
1600 Research Boulevard
Rockville, MD 20850
Phone: 240-314-7562
E-mail: sarahwoodruff@westat.com