Paper IN07

When Bad Programs Happen to Good People: Shuffling, Shifting, and Structuring an Inherited SAS Program

Gary E. Schlegelmilch, US Bureau of the Census, Suitland MD

ABSTRACT

Invariably, when we go to a SAS programming class, we are taught the basics of how the language works, and ways to lay out a program. The problem is, once we leave an academic arena and into the professional one - we don't always write programs from scratch. Often, as jobs change, personnel change, and requirements change; we are often called upon to take an old program, and update it for new requirements. In this paper, we'll discuss some of the basic tools available within SAS, and how to use them effectively to update outmoded, unstructured, and just generally bad coding.

INTRODUCTION

This paper is intended to give a brief overview of some of the tools and techniques available to the SAS programmer for reformatting an older program, in order to increase its readability, and accordingly, its maintainability.

It's important to understand – there is no one, exact, completely *right* standard for *any* programming language and SAS is no exception. This paper offers no more than a few ideas for how to take a program that is difficult to read and understand – and provide some common sense approaches to making your own task of maintaining an unintelligible program easier.

No matter which platform you are working on, there are always a variety of word processing and editing tools available. One unusually powerful one for language editing, however is the Program Editor window. It contains a number of capabilities that are not readily available in other processors.

Given the following code:

```
DATA LOC1; FILENAME F1 'C:\WORKAREA\INPUT.DAT'; FILE F1; INPUT FLD1 FLD2 FLD3 FLD4
D4 TXT1 TXT2 TXT3; RUN; PROC SORT IN=LOC1; BY FLD2; RUN; DATA _NULL_; INFILE F1; PUT
@1 FLD2 @17 FLD3 @37 FLD1 @55 TXT1 @78 TXT3; RUN;
```

This is an oversimplified program; the newest programmer will recognize this is a routine which will read an ASCII file with spaces delimiting the fields, sort it by FLD2, and print selected fields into another data file.

But – what does it *do*, from a functional standpoint? No way to tell.

That's the difference between a maintainable program and a difficult one. Change requests for programs are not going to come in labeled TXT2 or FLD1. They are going to ask that the Secondary Comment field be added as a second line of the report, or that a new field, EMAIL_ADDRESS, is going to be added to the input stream.

There are two keys to building readable and maintainable software. One is to prepare the code in such a way that it flows well, and uses the self-documenting features like structure, and understandable data and step names. The other is the nonexecutable lines – comments, comments, comments. Each piece of information you have can readily be entered right into the program, the most logical place for it to be.

TOOLS IN THE SAS EDITOR

In order to use a number of the commands I outline here, you will need to have line numbers on your editor window. It's true that in a point-and-click, cut-and-paste environment, it seems an anachronism to go back to a line editor. Still, the philosophy here is to allow you to use every tool in the toolbox – not just the ones on top. There are some line commands, like indentation and text adjustment, that are only available as line commands. To get line numbers, use the Tools PMENU, and turn the Command Line on. Once you have the command line, type "NUM ON" to get line numbers added to your text. In some versions, you can configure SAS to have the command line or line numbers part of the initialization of the routine, so you only have to do it once.

Once you have line numbers, you have a wide area of commands available to you. The first I normally use and recommend is the TS, or Text Split command. By putting TS on the line number, then moving the cursor to the desired break point, it will continue everything to the right of the cursor on the next line. As an example, where "^" represents the cursor:

```
00001 DATA LOC1;^ FILENAME F1 'C:\WORKAREA\INPUT.DAT'; FILE F1; INPUT FLD1;
```

becomes:

```
00001 DATA LOC1;
00002 FILENAME F1 'C:\WORKAREA\INPUT.DAT'; FILE F1; INPUT FLD1;
```

and so on. It's a good rule of thumb to keep each executable line as a separate line of code. In addition to readability, it also makes it easer to enhance the code in the future, in the event that you need to add a line in between two others; it also makes the Data Step Debugger easier to follow.

The reverse of this, to group lines together, is the Text Flow command, TF. Text Flow will join consecutive lines into a single line. The command stops at the first blank line. If no line length is set, all lines will be put onto a single line with the maximum length of a line in the Program Editor. Or, you can set a line length in the command. As an example:

```
00001  data _null_;
TF     length FLD1
00003  FLD2
00004  FLD3
00005  FLD5 $60.;
```

would result in:

```
00001  data null_;
00002  length FLD1 FLD2 FLD3 FLD5 $60.;
```

If the line command was TF20, the results would have been:

```
00001  data null_;
00002  length FLD1 FLD2
00003  FLD3 FLD5 $60.;
```

If the lines contain leading spaces, they will be carried over to the new line; if not, data from each subsequent line will be separated by a space. *Be careful* – entering TF on a line will gather *all* following lines! The UNDO feature of the Program Editor will undo the effects of TF – but it certainly can make a mess.

To eliminate those errant spaces, a tip; do a TF to gather all the data on one line, then a TF5. That will reduce all the data to 5-character lines, building a new line for anything over five characters, and removing leading spaces. Another TF will pull everything back into one line – but without the errant spaces.

Another frequently used approach to structure is to call attention to only those things in the program you need to call attention *to*. It is a reasonable assumption that the SAS code works as it is supposed to; so the only things you need to call attention to are those that you need to research in the event of an error, those that are user defined.

An easy way to call attention to words is to type them in all capital letters. Since SAS version 7 enhanced the length of a data field name to 32 characters, it has become easier to use a clearly defined data name; however, by using upper- and lowercase lettering, they can be made even more prominent.

Using the upper example; the first two lines of the code become:

```
00001 data LOC1;
00002 filename F1 'C:\WORKAREA\INPUT.DAT';
```

Now, in a 10,000 line SAS program, retyping each individual line can be tedious. By putting LC in the line number field, it will render the entire line in lowercase; UC will render in uppercase. An entire block can be changed by using LLC or UUC on the first and last line numbers of the entire block you wish to change.

Now the first routine looks like this:

```
data LOC1;
filename F1 'C:\WORKAREA\INPUT.DAT';
file F1;
input FLD1 FLD2 FLD3 FLD4 D4 TXT1 TXT2 TXT3;
run;
```

One of the oldest techniques to show that one piece of code is subordinate or a part of another is to indent it, as it was a required part of COBOL and FORTRAN programming. Some early applications language mandated that certain types of code begin in specific input columns, so card punch machines could be set up to advance to particular column for setting up programs. Not many, if any, still use paper tape or Hollerith cards for input; still, an indentation to group lines of code is not a bad idea.

Again, you can add spaces to each individual line to put it where you want it; or by typing ")" to move the line one space to the right or "(" for one space to the left, you can move the entire line. For blocks of code, begin and end the block with "))" or "((" in the line number. You can move a specific number of spaces by typing the number afterwards. As an example:

```
00001 data LOC1;
))3   filename F1 'C:\WORKAREA\INPUT.DAT';
00003 file F1;
))    input FLD1 FLD2 FLD3 FLD4 D4 TXT1 TXT2 TXT3;
00005 run;
```

becomes:

```
00001 data LOC1;
00002    filename F1 'C:\WORKAREA\INPUT.DAT';
00003    file F1;
00004    input FLD1 FLD2 FLD3 FLD4 D4 TXT1 TXT2 TXT3;
00005 run;
```

Now, another way to simplify this routine is to take out any line not specific to the data step, leaving only the executable ones. The FILENAME statement is, in this context, providing information. So, you might do this one of two ways – by defining the FILE statement with the physical name of the file:

```
00002  file 'C:\WORKAREA\INPUT.DAT';
```

eliminating the need for the FILENAME statement altogether. My typical preference is to define the file external to the executable part of the program, for two reasons. It sets the definition aside, making it more self-documenting; and, it eliminates the need to change the program in multiple places. If file F1 is used in only one place, either is acceptable. However, if the program reads F1 in twelve different data steps, each data step would have to be changed. By using on external FILENAME statement, it defines it for the entire program.

```
b0001 data LOC1;
m     filename F1 'C:\WORKAREA\INPUT.DAT';
00003 file F1;
00004 input FLD1 FLD2 FLD3 FLD4 D4 TXT1 TXT2 TXT3;
00005 run;
```

becomes:

```
00001 filename F1 'C:\WORKAREA\INPUT.DAT';
00002 data LOC1;
00003 file F1;
00004 input FLD1 FLD2 FLD3 FLD4 D4 TXT1 TXT2 TXT3;
00005 run;
```

It is also helpful to offset groups of code with a blank line, so each function is more clearly defined to the casual glance. So, in the next example, there will be a space between lines one and two.

Blocks of code can be moved as well, by starting and ending the blocks of code to be moved or copied with “mm” or “cc” respectfully. The destination can be defined by placing an “a” or “b” on the destination line, indicating to move the lines after or before that line, respectively. Also, lines can be combined by moving or copying a single line with the destination “o”. The “o” will move every column of the copied/moved line to the new line, *providing there is not already a character in that column*. In other words, it does not overwrite any characters at the destination location. As an example:

```
m0042  filename
o0043           F_INPUT;
```

becomes:

```
00042 filename F_INPUT;
```

but:

```
m0042  filename
o0043      F_INPUT;
```

would become:

```
00042  fileF_INPUT;
```

so the command must be used with care.

If you need to look at another file, log file, or a data file – remember that the Program Editor can read any ASCII file. Just fully qualify it in the OPEN option of the PMENU, or by using the INCLUDE from the command line.

Finally, a bit of detective work is needed. Since the input file is free form, and we impact no external datasets, we can clean up the data names. Sometimes, this option is not available to us, since we must follow the dictates of an external dataset. But in this case, by working backwards from an existing report, we can determine more effective names for the fields within the program.

It is also worth noting that there are a few data fields that are not used in the final product. If we are talking about a few hundred occurrences from a data file of a few fields, the processing time and storage space is minimal. However, if we are reading half a million records and processing 650 data names to use 6 – the time and storage could be significant. Therefore, it behooves us to make it a habit to be effective.

It’s also interesting to note that the Find menu option in the Program Editor is not case sensitive - but the Change *is*. So, if you use it to change FLD1 to PURCHASER_NAME – it will find fld1, Fld1, and FLD1, and change them all to the desired output.

So, after a few mass changes, the resulting program looks like this:

```
filename STORE_INPUT 'c:\workarea\input.dat';

data WORK.LOCAL_PURCHASE (drop=PURCHASE_PRICE CUSTOMER_ADDRESS);
   file STORE_INPUT;
   input CUSTOMER_NAME PRODUCT_NAME PRODUCT_CODE PURCHASE_PRICE
         CUSTOMER_TELEPHONE CUSTOMER_ADDRESS SERIAL_NUMBER;
run;

proc sort in=WORK.LOCAL_PURCHASE;
   by PRODUCT_NAME;
run;

data _null_;
   infile STORE_INPUT;
   put @1  PRODUCT_NAME
       @17 PRODUCT_CODE
       @37 CUSTOMER_NAME
       @55 CUSTOMER_TELEPHONE
       @78 SERIAL_NUMBER;
run;
```

Functionally, exactly the same program. But which style would you rather try and modify a year from now?

TECHNIQUES IN PROGRAM FLOW

There are some very simple ways to simplify code in program flow. For instance, given the following (<perform routine> is simply to indicate some kind of SAS code):

```
data _null_;
set WORK.INFO;
if code='J' or code = 'C' or code = 'F' then do;
<perform routine1> end;
else if code='Q' then do;
<perform routine2> end;
else do;
<perform routine3> end;
run;
```

Now, from a functional perspective, this code will operate without problems. However, each time a new option is added, you will need to extend the program by another IF/ELSE condition. A simpler way, just as effective, might be to code it as follows:

```
data _null_;
   set WORK.INFO;
   select (CODE);
      when ('J','C','F') do;
         <perform routine1>;
      end;
      when ('Q') do;
         <perform routine2>;
      end;
      otherwise do;
         <perform routine3>;
      end;
   end;
run;
```

In an IF statement, the variable could also be grouped by using an IN operator, like this:

```
if code in('J','C','F') then do;
```

Here's another good rule of thumb that can make your code stand out. If you use the equal sign "=" only for assignment, and character values "eq", "ne", "gt" and so forth for comparisons; you can see at a quick glance where you're assigning a value.

Typically, we all use DO loops when we need multiple lines of code following another SAS construct like IF. However, recall that you also have the DO WHILE <condition> and DO UNTIL <condition> options available to you.

SELF-DOCUMENTATION ISN'T ENOUGH!

There isn't a programmer in existence that doesn't appreciate information about someone else's program when it comes time to modify it. Unfortunately, this is the part of the program that usually suffers the most. When modifying the routine, we're often too busy to write notes; when we're done, it's time to move on to another project. Both are habits we need to break.

Good simple rules of thumb – and many which are actually demanded by corporate and government standards, as well as CMM requirements, are as follows:

1. Each program you write or modify should have the basic information in a comment block at the beginning of the program:
   a. Program name
   b. Author
   c. Date developed, if known
   d. Inputs and outputs
   e. Statement of purpose
   f. Log of modifications

It doesn't sound like much, does it? But put it into a simple perspective; how much of that information would *you* like to have, if you need to modify the program?

2. When you learn something about a program, write it down. And the best place to do it is in a comment imbedded in the program.
3. Always use the self-documenting features of the language; meaningful data names, datasets names and macro names will always make the program easier to read.

Remember there are two kinds of comments, and either can be used on any line in the program. There's the comment statement, which starts with an asterisk "*", and ends with the semicolon; and the delimited comment, which is any sequence starting and ending with "/*" and "*/". Either can be a standalone line, or part of an existing line. The major difference is that the comment statement ends with a semi-colon, so it cannot be readily used to comment out a block of code. The delimited comment will ignore *all* characters from the "/* on, until it finds a "*/".

```
/*------------------------------------------------------------*/
/*    Set up the roles;                                       */
/*------------------------------------------------------------*/

ALTER_EGO='Clark Kent';  /* out of costume */
IDENTITY='Superman';   * in costume;
```

CONCLUSION

This barely scratches the surface of what can be done to make a program readable. Macros can easily be used to replace redundant code. Books have been written on how to make code effective. SAS's own documentation on the Program Editor is extensive. But the main idea is this; when you inherit someone else's code, there are ways to make your own life easier. They might require a little extra effort – but the result is uniformly positive.

REFERENCES

Aster, Rick, *Professional SAS Programmer's Pocket Reference*, 4th Edition, 2003, Breakfast Books.
Aster, Rick and Seidman, Rhena, *Professional SAS Programming Secrets,* 1991, McGraw-Hill.

ACKNOWLEDGMENTS

Thanks go to Marje Fecht for encouragement, and Rebecca Bryant of the USAF Air Command and Staff College for editorial support.

CONTACT INFORMATION

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

Gary E. Schlegelmilch
US Bureau of the Census, ESMPD/MCDIB
4700 Silver Hill Road
Washington DC 20233-6200
Office: (301) 763-7522
Fax: (301) 457-4437
Email: Gary.E.Schlegelmilch@census.gov
Web: www.sesug.org