Clean-up, Comments and Code – Making it Maintainable

Clay and Lori Martin, Martin Consulting, Susquehanna, PA

ABSTRACT

It was exciting writing that new program: getting the code to run without errors and getting the results for which you've worked so hard. Time to move on to the next task – or is it? What about those warning messages and those strange notes in the log? What about documenting and commenting the program? Can someone other than you easily understand what the code is doing?

I'll do it later, you say.

Will you remember it *later*? Will you be around *later*? Granted, time spent on making code maintainable is not exciting. However, it is valuable because the code is easier to:

- fix when there are problems
- upgrade to support new requirements
- enhance to handle new requests
- turn over to someone else.

This paper will focus on making your code more maintainable. It is aimed at an audience of SAS programmers, applications developers and consultants. Topics covered include:

- How to clean up messages in the log.
- The ins and outs of comments.
- Overview of code standards, readability, and reuse.
- Planning for debugging.

INTRODUCTION

This presentation shows you how to increase your productivity by attention to detail during coding and debugging. You're productivity will be increased because you will not be spending as much time reacquainting yourself with existing code. Additionally, you will not be spending as much time correcting errors that popup later in the life cycle of the software due to software or system upgrades. Another added benefit is that these techniques are that it makes it easier to turnover a project so you can move on to a new one.

You will use warnings and notes to identify underlying problems with design or data. You will provide in-code documentation to reduce the time spent identifying the function of existing code. You will use comments to provide visual cues to aid you in finding the critical sections of code. You will use coding techniques to make your programs more error free and maintainable. You will use debugging techniques to reduce the time spent finding an error and reduce the chances of adding errors during the debugging process.

The techniques described in this paper apply equally well to new code and code you've inherited from someone else

CLEAN-UP

The program reads the data and produces the correct output without errors. It's done right, Maybe? The two other types of messages (warnings and notes) in the SAS log may indicate underlying problems with the design or data. Always review the SAS log thoroughly. Never assume that because there are no error messages that there are no problems with the code.

LOG MESSAGE SUPPRESSION

Warning, there are a number of options, that when present, suppress normal log content. System options that may interfere are *nodsnferr, nofmterr, noreplace, noerrorabend*. These all suppress error messages. Furthermore the option *nonotes* will also suppress log content. Adding a *proc options* statement to your program will identify what options are in effect.

The presence of "?" and "??" on input statements or functions will suppress errors. The level of error suppression varies but both propagate missing values. Because you need to know about the presence of errors, it is inadvisable to leave these in production code. If your program works right with the expected data, you certainly want it to notify you of any changes.

WARNINGS

Always take WARNINGS on the log seriously. Warning messages may be from a known reason, such as an update that you expected. It's bad technique to ignore a warning even if the reason for the message is known. The profusion of warnings, or the mere fact that everyone's used to seeing warnings for this program can mask a new error condition (that last upgrade?). If a program generates 50 warning every time it is run, will anyone notice the day that there are 51? If the log contains many known warnings it increases the chances that a new unknown warning will be missed.

If the reason for the message is unknown, obviously an investigation must ensue to uncover the reason. There are techniques for debugging that increase the chance of a quick solution, and lessen the chance of introducing new bugs. Either way, the code should be updated to handle the situation without generating the warning message(s).

Format is already on library

This message is ignored most of the time. If you are updating formats you will get this message when the format was already built and placed in the catalog you are referencing. If a typo is loading a format to $FRED instead of $RED, it might be cause for concern. The regular appearance of this message in the log could mask the occurrence of the message when an error occurs.

```
WARNING: Format $FRED is already on the
library.
```

If you are expecting $FRED to be in the library, you may think you don't want this warning message cluttering your log. One way to avoid this message is to create all of your formats in the work library. When you have completed creating the formats, use PROC CATALOG with a COPY statement to move the formats to the permanent library.

```
proc catalog cat=work.formats;
  copy out=permlib.formats;
       select fred.formatc;
run;
```

It is important to use the select statement. This serves as a safety net to catch inadvertent errors earlier in the process. The lines of a program are written only once, but run many times. Extra time spent coding is always recouped against time spent tracking down errors, or rerunning production.

Variable is not initialized

Ignoring this message surely can't hurt? The program assigns a value to the variable in the data step, right? So there's nothing to worry about? Warnings are not issued without a reason. When a variable does not have an initial value, it is assigned the value of "missing". By default the missing value for a numeric variable is a dot, ".", while the missing value for a character type is a blank, " ". Each of these missing values has their own set of problems.

The missing value for a numeric variable sorts lower than 0. So comparisons and sorts may work unexpectedly. An arithmetic operation involving a missing value can create more missing values if the programmer doesn't go to lengths to handle the problem. For instance

```
data valdata (keep = k);
/* a test that may not always succeed */
If _n_ = 15 then x = 32;
…
k = x + 1;/* much later x is used with k*/
…
```

In this example the variable x will have a missing value in all iterations other than during the 15th observation. You can imagine less obvious expression, but importantly, x has a missing value when the if statement's expression evaluates to false.

You can make one or more changes to correct this lapse in logic. You can add an else clause to the if statement or assign a default value to x before the *if* statement. Another solution is to use a *retain* statement to initialize the value of x. This will work if the logic of the problem allows the previous value assigned to x to persist in an iteration where the if statement fails after it has succeeded once. Note if the previous value of x can be passed along, that invalidates the need for the if statement except in specialized situations. Keep in mind that the assignment of k involving x may be separated by many lines from the if statement. You or another might not note the "sometimes" assignment of x, especially if the log contains many missing value warnings.

If missing values are in your data, then you must deal with them as a normal part of your process. Some methods include the use of input options *missover* or *stopover*, carefully choosing whether to use sum assignment (x+y) statement or use the *SUM* function. However, missing values generated by the programmer can propagate into the data, and thus can create new errors when other programmers update the code (sort example). Therefore you should always initialize all variables your program. This way you account for missing values of any input variables so you don't leave potential problems for others (or yourself at a later date).

Apparent invocation of macro xxx not resolved

When you see this message, the code in a macro was not executed. The rest of the program may finish without any other error or warning messages. Even though this is a warning message, you should determine why the macro was not executed. Some causes of this problem follow:

- It may be that the SASAUTOS= option was reset to another library. This library may not contain a copy of the macro.
- You have misspelled the macro name.
- AUTOSOURCE system option is turned off.
- AUTOSOURCE is on, but you have specified an incorrect path in the SASAUTOS= system option.
- You are using the autocall facility but have given the macro and file different names.
- In the Windows environment, you are using the autocall facility but didn't identify the file with the .SAS extension included.
- There is a syntax error within the macro definition.

NOTES

Many times notes just indicate a interesting happening that the SAS compiler has handled for you. Simplifies your job right? But by letting the compiler handle it, you are ceding control over to it. This has a number of downsides. First, you are the paid to think, modern advertising claims aside, programs are just bunches of code written by someone else (that you don't even know) and are not as qualified as people. As of this writing, no programs can think! The SAS compiler does not know the meaning of the data or the purpose of the project. The writers of that compiler can change its function in a latter release. Second, the writers of the compiler may decide to enforce, as an error, something that was ignored before. You could have hundreds or thousands of errors to fix before the next production run. Lastly, by letting operations on the data occur without the corresponding logic in you code, you mask the original intent of the process. This makes the program difficult to maintain and more prone to bugs if the program gets updated, or reused.

Missing values were generated …

The cause of missing values should be identified. Sometimes missing data is a sign of a much more serious problem. The question is: Is the missing value on the output a simple transfer of a missing value on the input, or was the missing value created in the data step (see "Variable is not Initialized" above). It requires careful thought to determine how missing values should be handled for each program.

Why should you handle missing values? Isn't that the problem of the person providing the data? Assuming the missing values from the input data, you need to consult the person responsible for the data source to see if missing values are valid. If they are, it behooves you to include a *missover* option on the *input* statement to keep your data aligned. If they are not, then a *stopover* option makes more sense. Even if missing values are allowable you must consider if you want them to be propagated into later data steps. You can prevent propagation by testing for missing values. If you find them you can assign a default value and perform any additional logic (such as writing out a message or incrementing a counter).

When you see the "missing values" message, you must determine which variables have values that are missing. With a large data set this can be hard to debug, as the message doesn't tell you what variable is missing. In the book, *Cody's Data Cleaning Techniques Using SAS Software*, you can use a macro like the following to identify the culprit:

```
/************************/
/* create test data set  */
/************************/
data test;
  input x y a $ x1-x3 z $;
  datalines;
  . 2 x 3 4 5 y
  2 999 y 999 1 999 j
  999 999 r 999 999 999 x
  1 2 3 4 5 . 7
  ;
/*************************************/
/* program to detect a specified value  */
/*************************************/
%Macro find_x(dsn,num);
 data _null_;
   set &dsn;
   file print;
```

```
    /*length 32 for V7 or later*/
    length varname $ 8;
    array nums[*] _numeric_;
    length varname $ 8;
    if _n_ = 1 then put "Occurances of
       value '&num' in data set: &dsn";
    do i = 1 to dim(nums);
      if nums[i] = &num then do;
        call vname(nums[i],varname);
        put "Variable: " varname
          " Oberservation: " _n_;
      end;/*if*/
    end;/*do loop*/
    drop i;
run;
%mend;/*find_x*/

%find_x(test,.);
```

Character to numeric conversions

These occur when a variable already identified as a character is involved in an arithmetic operation or is assigned a numeric value. This could be an obvious mistake, but more likely it occurred subtly when a macro variable (they are always characters) is assigned to a numeric variable in a data step. If the character value continues to be a number, you might think this is ok, but the implication to a later reader of the program is that both variables are numeric. If you use a *put* statement to do the conversion, then if the character value turns out to be “abc” instead of “15” then a serious error message is generated. Keep in mind someone else may have updated the macro (without changing anything significant of course!). Without the use of the *put*, such an obvious failure is only marked with the innocuous character to numeric conversion note. This could lead to the propagation of missing values.

Numeric to character conversions

These conversions usually occur when numbers are being tucked (concatenated) into strings. This may be ok, but volume of these messages can mask the one that is an error. Good programming practice says do the conversion properly with a *put* statement. This practice insures that the one conversion error that occurs as part of a bug will be identified immediately. Moreover using a *put* statement gives the programmer control over the format of the result. Control is good!

Input file is empty

This seems like it should be more than a note, but alas it is not. A program is intended to process data. If the data is not there, then conceptually, from the practical standpoint, something is wrong. How many other programs will continue to process no data before the error is caught. For this reason you should implicitly check for empty input files.

Though an input file may be intended to update another, why are there no records to add? Was it a bad business day, management would like to know that. Was there an error in the tape vault? The process should stop until the data is found. It would be meaningless to continue and process nothing when data does exist, data center staff would like to know that. Did someone make an innocent change in part of the process that caused this data to be deleted, a programmer should know about that.

You can easily add code to check if obs is equal to zero. If this is the case you can generate a useful message and end the process with and error code. This lets everyone know that an unusual condition occurred. Someone will no doubt find this information important.

COMMENTS

Design documents are nice, specs are ok, and user documentation is necessary. But for a programmer working with existing code, in-code documentation, known as commenting, is essential. In-code documentation has a few major advantages over other forms. One, it is *with* the code. If you are a gardener, isn't it better to label the rows instead of hiking back to the house to look at a diagram every time you need to tell the difference between oregano to make sauce, and pennyroyal to keep fleas from dogs bed. Two, it is written by programmers for programmers. Thus, it can get to the point without the literary wrappers required for non-programmers.

Top-notch programmers expect to work on a lot of projects. If the projects are worthwhile, they contain many modules and are written over a long period of time. After the initial project is complete, then comes the added work that was forgotten until a) the user saw the working product, b) last minute bugs are found, or c) requirements for the next version are specified. If you want to work on four modules for the rest of your career, sure you can remember all of them. But that's not much of a career, and maybe you should look into an assembly job. The complexity of modern software systems insures that you can't hold all of the meaning of all of the code *in your head*. Comments allow for "offline" storage of crucial information about a chunk of code.

If you have a lot of experience, why should you spend time putting in comments? Well it cuts down the time you spend explaining things to your juniors. It makes it a lot easier to turn over the code so that you can move on to a more interesting and technically advanced project. And you get paid the same rate for comments as lines of code. Most junior programmers learn the trade looking at existing code. The competence of the total body of programmers keeps companies from hiring liberal arts graduates to do their coding. The good example you set in your code improves the worth of those that follow you. Besides, when the junior programmers muck it up, who has to drop their project to bail them out? A good point to remember: even though a program be only 3 lines long, one day it will have to be maintained.

PROLOG

A prolog is a large comment block at the beginning of a program. It is a single area to describe the operation of the code and to provide information about the requirements to run the program. Because of the prominent location (at the top of the program) it is also a fine place to keep maintenance information.

A good (and complete) description of the operation of the program is most useful when looking at the code six or more months after the last time you had to deal with it. This is especially true when you were not the author. Yes any competent programmer can look over the code, and after a period of time deduce the operational characteristics. How long this takes depends on how well the in-code documentation was done. But it requires less time to read the description and then peruse the code. If you are trying to reuse code, some timesaving here and there can add up quickly over a few weeks. If you are less experienced that the author, the timesaving can be large. Any time saved works directly toward your productivity.

Besides a description, other items of information to include in a prolog are:

- Input datasets
- Output datasets
- Temporary datasets
- Macros used
- Exceptional error conditions/requirements to avoid errors

- Programmer name/date created
- Copyright information

You can simplify adding prologs to new code by using a standard version and then modifying it to suit the current program. The standard version can be stored in a file, or as a template in an integrated development environment. Having a structured prolog has the additional benefit of allowing tools to scan and extract prolog information.

MATCHING "ENDS"

Hardcopy aside, you can only see about 20 lines of code at once. In a large block of code with various structures within structures, it is sometimes hard to know if the *end* you are looking at is the *end* of the loop or the *end* of the *then* block contained within the loop (and four other do-end blocks). This becomes even more important when its necessary to add more do-end blocks to existing code. A small comment tacked on to the *end* statement helps identify which *do* the *end* belongs to. Consider the following.

```
if x = y then do;
   …
  if z ne 0 then do;
     …
     do I  = z to 100;
       …
       if w < z then do;
          …
       end;
       else do;
         …
       end;
     end;
     if x = 10 then do;
       …
     end;
/*** what_block_am_I_in ***/
  end;
end;
```

Yes, on one page most of us could point to an end and say what block it belongs to, but try that by covering the top half so you can not see it. Now that the top is out of sight you can not even use the indent level to decide where one block starts and if the line is contained within it. Now consider the following.

```
if x = y then do;
   …
   if z ne 0 then do;
      …
      do I  = z to 100;
        …
        if w < z then do;
           …
        end;/*if w*/
        else do;
           …
        end;/*else*/
     end;/*do*/
     if x = 10 then do;
       …
     end;/*if x=10*/
/*** what_block_am_I_in ***/
   end;/*if z*/
    …
end;/*if x*/
```

These small comments tacked on to the *end* statements allow them to be paired with their corresponding *do* statements regardless of how little you can see. This commenting idea may not be very useful on blocks that are not heavily nested, but if more do-end blocks are added to the existing code, it could prove useful.

USEFUL VERSUS USELESS COMMENTS

Comments themselves are not necessarily effective. To be useful they must convey information that is not obvious. Comments should not repeat the form of a piece of code, but rather convey its purpose. You may have seen a comment like the one below.

```
/*If x is greater than 10*/
if x > 10 then do;
```

Though this type of comment may appeal to the literary instincts of some managers, it helps the programmer not one bit. It doesn't tell why it is important that x have a value of ten. It doesn't even say why the number 10 is significant or what the interpretation of x is. A more useful comment might look like this

```
/*Charge greater than the minimum */
if x > 10 then do;
```

Or better yet

```
/*Charge greater than the minimum flush
the */
/* record and charge $1
*/
if x > 10 then do;
```

Positioning and visibility of comments can also convey additional information, and provide new functionality. Comments placed at the end (right side) of a line of code say "I'm just referring to this line. Comments placed in blocks of stars separate major functionality changes, and allow you to scroll quickly through code and spot it's major functional breaks. Comments placed far to the right serve as flags for meaningful subsections within a block, for instance when-select blocks.

```
x = 0;           /* Reset x */

do x = 0 to 100; /**Percentage loop**/


/*small point*/


/*** More important ***/


/************  Minor break **************/

/***************************************/
/**      Significant Subsection        **/
/***************************************/


/***************************************/
/******** Major Subsection Header  ******/
/***************************************/
```

CODE

There are a few coding techniques that can be employed to make programs more error free and maintainable. The use of naming conventions can add a consistency of meaning within a project. This allows more to be understood while looking at a small chunk of it without resorting to scrolling to another area to look something up. Naming conventions for files allow the programmer to quickly locate a specific program or macro. Indention provides graphic clues to the structure of the program as well as visually annotating relationships between the elements of a program.

The overall design of project's segments allows for reuse, saving the programmer's time during the coding process. A common look to all modules subtly suggests meaning thus giving the programmer less to actively hold in their thoughts. Debugging techniques can speed the process of removing errors while lessening the chance that the debugging process will instill new bugs.

Naming Conventions

Given the file name length restrictions on the mainframe, eight-character limit on variable names (in V6), there does not seem to be a lot of room for naming conventions. But a lot of good can still be done within those limitations. You can't use a file if you can't find it. If you take care with variable names you can help yourself spot errors.

Files

Though many times on the mainframe you are restricted as to where things can be stored (levels of data set names). Where applicable you can use the various levels to help define what is stored. Working from the high level qualifier down, the words you use should go from least specific to most specific. For instance you could divide all programs into two categories, program and test. These would make good high level qualifiers. Any project could have detail, summary, and reporting segments of code. Because these are less specific they should be a lower level qualifier. Both test and production, or accounting and performance could involve these three subcategories.

On desktop computers and UNIX mini's you have a different structure for storing files. On these systems the entire path to a specific file could be considered part of that individual file's name. Applying the above ideas still generates a useful subsetting of groups of files. One other note about desktop computers, if you work on programs destined for the mainframe, you should organize your directory structure with the same directory names as the DSN levels. This allows you to quickly convert slashes to periods. It is also useful not to have to remember and relate two different file location organizations to insure that you are working with the correct file during debugging and testing.

Variables

Variable names, other than the odd loop counter, should reflect what they represent in a program. Variables for specialty types, such as date or time types, should indicate that in the name. Try to be consistent. For example, use the variable prefix to indicate the function of the variable and the suffix the type of variable. If you have date variables for a reporting routine they may look like this:

Rptfiscdt where rpt = report routine
Fisc=fiscal
Dt=date

Whatever makes the most sense for your environment should be determined up front so that all variable names can be consistent and other programmers involved know how to read and create variables for the project. Ambiguous names should be avoided: FLAG1, COUNT and RC. What is being flagged? What is being counted? Which reason code is being returned? It's better to use IS_FOUND, ERRCOUNT, and ALLOC_RC. These names minimize the time you spend thinking about what the variable represents.

Although version7 and higher allow variable names up to 32 characters, many sites are still using version 6 for production. SAS version 6 only allows eight characters for a variable name. Therefore, using one or two up to identify type seems at first to be rather wasteful. But given that you make the last two characters indicative of type, eg dt for date and tm for time the rest of the name length can better be used to indicate the source of the value. For instance chrgdate becomes chargedt. And since dt for date becomes standard through out your code, the dt successfully and obviously replaces date in mind of the programmer. This way you don't end up with chrgdate, startdat, finishdt, and inundate all referring to a date value. This technique is especially useful in large project worked on by many.

Even loop counters can have a consistency that makes for intuitive understanding a statement deeply nested in multiple loops intuitive. Since the bad old days of Fortran, I, J, and K have been the names of choice for loop counters. If you always use them in that order, outer loop to inner loops, it makes understanding the following statement easier.

```
Result = (rate[K] * cost[J] + oldcharge[I]
) +
   (space[I] - discount[K] * cyls[J]);
```

Macros

Macro names should be the same as the file (member) that they are stored in. This allows the autocall facility to be used. It also makes it easier to find the macro when debugging or upgrading. Only one macro should exist in each file (member).

*Do not prefix macro variable names with AF, DMS, or SYS***.** These letter combinations are used by SAS as prefixes for automatic variables. No errors occur and SAS will not prevent you from using AF, DMS, or SYS as a prefix. However, using these strings as prefixes may create a conflict between the names you specify and the name of an automatic macro variable. There is also the possibility of the conflict with automatic macro variables added in later SAS releases.

INDENTION

Indention greatly improves the readability of code given that it meets two criteria. First it must be applied consistently. Second its meaning must be consistent. With SAS programs having right hand boundaries on mainframes, economy of screen space must also be taken into account.

If sections of a program use indention, and other parts do not, the programmer cannot rely on indention to provide reliable information. Similarly if different indention styles are used with in the same body of code, the information content ascribed to the depth of the indention is lost. If indention is used within a body of code it should be applied consistently throughout.

Individual statements in SAS can be broken into two distinct groups. Statements that contain statements and those statements that are contended. Indenting the contained statements by two characters can differentiate these types of statements. One character is not visually significant in long blocks, so two are used. Lines of SAS code can be similarly broken down. Lines that contain statements, and those that are continuations of previous statements. Lines that are continuations should be indented four spaces from the line that begin the statement. Four spaces are much easier to distinguish from two spaces than three would be. Note the use of indenting below.

```
if employed then do;
  rate = time / worth;
  salery = time * worth;
  do I = 1 to 365;
    day = coffee !! 'email' !! problems[I]
         !! lunch !! erands !!
        (whatever * project[I]) !!
        coffee !! 'meeting' !! cleanup;
    put 'day ' !! calender[I] !!
        ' was like ' !! day;
    totprob = put(problems[I],10.)
        !! totprob;
  end; /*do*/
  rating = salary * (365 / totprob);
  put 'rating for this year is ' rating;
end; /*if*/
```

REUSE

The best way to save time while coding is to reuse sections of code you have already written. With care, most sections of code can be written to ease the task of reusing them. In addition to the timesavings gained by reusing code, the code you reuse is (hopefully) bug-free. Reusing code does not necessarily mean keeping a library of code snips. Sometimes it just means copying blocks of code over and over when you are doing the same thing to a few different data sources.

Written to be copied

When a block of code is going to be copied to multiple places in a program it should be written so that a few confined search and replace operations can suit it for it's new position. Consider the blocks of code below. The first (CICS) is written so that it can be block copied and changed to work for the DB2 processing in the second block. This takes a little planning up front, but the time spent is far less than hand coding both (and maybe 5 other similar blocks).

```
/*************************************/
/**    PRODUCE EXCEPTION REPORTS      **/
/*************************************/
                            /** CICS **/
DATA _NULL_;
MERGE CICSUM1  CICSUM8 (RENAME=(XYZCHRG=OLD_CHG
XYZLTCB=OLD_QNT));
BY XYCUST1;
IF _N_ = 1 THEN PUT @13
    'CICS Charge/Quantity Difference Exceptions';
IF LAST.XYCUST1 THEN DO ;
   IF XYZCHRG > OLD_CHG THEN DO;
     PUT @5 XYCUST1 @15
         'This period CICS charge ' !!
         'exceeds  the last by '
         OLD_CHG -  XYZCHRG ;
   END;
   ELSE IF XYZCHRG < OLD_CHG THEN DO;
     PUT @5 XYCUST1 @15
         'This period CICS charge ' !!
         'less than the last by '
         XYZCHRG -  OLD_CHG ;
   END;
   IF XYZLTCB > OLD_QNT THEN DO;
     PUT @5 XYCUST1 @15
         'This period CICS quantity' !!
         'exceeds the last by '
         OLD_QNT -  XYZLTCB ;
   END;
   ELSE IF XYZLTCB < OLD_QNT THEN DO;
     PUT @5 XYCUST1 @15
         'This period CICS quantity ' !!
         'less than the last by '
         XYZLTCB -  OLD_QNT ;
   END;
END;/*LAST*/
RUN;



                            /** DB2 **/
DATA _NULL_;
MERGE DB2SUM1 DB2SUM8
(RENAME=(XYZCHRG=OLD_CHG
XYZLTCB=OLD_QNT));
BY XYCUST1;
IF _N_ = 1 THEN PUT @13
    'DB2 Charge/Quantity Difference
Exceptions';
IF LAST.XYCUST1 THEN DO ;
   IF XYZCHRG > OLD_CHG THEN DO;
     PUT @5 XYCUST1 @15
         'This period DB2 charge ' !!
         'exceeds the last by '
         OLD_CHG -  XYTCHRG ;
   END;
   ELSE IF XYZCHRG < OLD_CHG THEN DO;
     PUT @5 XYCUST1 @15
         'This period DB2 charge less ' !!
         'than the last by '
         XYZCHRG -  OLD_CHG ;
   END;
   IF XYZLTCB > OLD_QNT THEN DO;
     PUT @5 XYCUST1 @15
         'This period DB2 quantity ' !!
         'exceeds the last by '
         OLD_QNT -  XYZLTCB ;
   END;
   ELSE IF XYZLTCB < OLD_QNT THEN DO;
     PUT @5 XYCUST1 @15
         'This period DB2 quantity '
         'less than the last by '
         XYZLTCB -  OLD_QNT ;
   END;
END;/*LAST*/
RUN;
```

After the code has been copied, two search-and-replaces finish the block (CICS to DB2, and CIC to DB2). Creating a lot of code by copying blocks and doing replacements tends to make more uniform code (also variable/field names). This in turn makes it easier to copy and alter sections of the program later. If you use the same prefix, postfix, or whole words in comments, data set, and variable names, it increases the reusability and understandability. Code that does the same task should look similar.

Self Contained

Many tasks are similar across multiple projects or programs. Sorting arrays, checking for empty input files and specifying system options are all tasks that must be performed regularly. Tasks such as this are well suited to becoming entries in library of code snippets. To make them easy to use they need to be self-contained. Even though some lines contained within these snips may be better placed at the top of the data

statement, they can be included. After the block is inserted, the out of place lines can be moved from the snip to where they best belong. The time savings comes in by having code that references the correct variable names and labels with useful comments all sitting at the top of the snip waiting to be moved.

You should place comments in your snip identifying what needs to go where. These techniques allow the snip to be deployed into the new program without you having to study the code snip. Additionally, if you are keeping a snip of code for multiple uses, it deserves the effort needed to make it clean and well documented. By spending more time on the snip, you save more time every time you use it. In the end your final code will have a more polished and consistent look. That consistent look helps others to read your code, and you to debug or add features to a program.

DEBUGGING STATEMENTS

Wouldn't it be nice if everything ran correctly the first time? Because you can't catch all the errors all the time, an important task is debugging. Many languages have debuggers packaged with them. Despite their availability, the time-honored method of leaving "a trail of bread crumbs" still is the stalwart for determining what is going on within a segment of code. In this case the "bread crumbs" being left are *put* statements. You might use these to determine if your program has passed a point, or to determine a variable's value at a particular point in the process.

Where did it come from?

When you are inserting messages into the output it is important to identify where in the program they came from. This is simple when you are just identifying where you are in your program. The text of the message is intended to convey a location. Don't be overly brief or general, as you will probably add more debug code before your task is done. Avoid messages like "process done". It is better to add a few words now than have to go back and find and fix a lot of debug code because your output was flooded with "process done" messages from 3 different sections of your code. The message "Initial summary on data set xyz begin" will be more useful. If you have multiple modules it is a good idea to identify the module in the message. The message "ABC123XX: Initial summary on data set xyz begins" will be more useful.

If you are looking for the value of a variable, it is important to identify the variable name whose value is being listed. If you used the message "ABC123XX: 55" you might end up wondering was x = 55 or y? The message "ABC123XX: after 0 test x is 55" will be more useful. The debugging cycle usually involves returning to the broken code and adding more debugging statements. You can quickly end up with too much output from debug statements that don't identify themselves. Using these methods will save you a lot of time, going back to modify debug statements, because you can't tell what is what in the output.

Beware of Blanks

Character variables should be bracketed by characters in the output if you suspect they may contain or be padded by blanks. The string "abc" and "abc " look the same in the message "ABC123XX: name is abc". Blanks can be problematic if you are checking to see if "abc" was equal to "abc ", which it is not. If you suspect blanks you can make your message like so:

```
Put "ABC123XX: name is (" !! abc !! ")";
```

Mark It Well

With a suitable amount of debug statements and a bit of luck, you've found the problem. Now its time to remove all the debug code. Its very embarrassing to have a bit of debug code get executed in a finished product because it was not removed. Sometimes code is placed in sections of the program that did not get executed while debugging. In order to be able to find all the debug code, both *put* statements and locations where you have hardcoded some value should be marked. *Put* statements can have the word debug in their text. Hard coded values, or assignments and other "non output" code change should to be marked with a comment that uses the word debug. And any code that is commented out should have the word debug added to the commented area.

If you mark all your debug related changes and additions with the word debug, it becomes simple to search out all these occurrences. Usually the word "debug" does not appear in production code so it becomes the perfect flag to identify changes that were made during testing. Code changes that are not marked can easily be overlooked. After all if you made dozens of changes you can't always remember them all. Moreover, you can tell them apart from changes made to fix problems. Sometimes debug additions in one part of a project can hang around for weeks while other parts are tested or debugged. Only consistent marking of these debug statements assures that they can *all* be found. And after all letting a search find the lines is quicker than scrolling through thousands of lines of code.

CONCLUSION

This presentation has shown you how to use some simple techniques during coding and debugging to increase maintainability. You have seen how to avoid errors that popup later in the lifecycle of the software. You have learned commenting techniques and naming conventions to help yourself and others easily understand your code later on. By applying what you've learned here, you will not spend as much time turning over projects or being paged to fix bugs.

For more information on additional techniques for building maintainable code see the book, *The Next Step: Integrating the Software Life Cycle with SAS Programming.* The book, *Cody's Data Cleaning Techniques Using SAS Software,* contains many methods for getting to know your data and how to deal with potential problems bad data may cause.

REFERENCES

SAS Institute Inc., *SAS Macro Reference, Version 6*, Cary, NC, SAS Institute., 1996.

Cody, Ron, *Cody's Data Cleaning Techniques Using SAS Software*, Cary, NC: SAS Institute Inc., 1999.

Gill, Paul, *The Next Step: Integrating the Software Life Cycle with SAS Programming*, Cary, NC: SAS Institute Inc., 1997.

CONTACT INFORMATION

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

Clay and Lori Martin
Martin Consulting
312 Myrtle Street
Susquehanna, PA 18847-1522
570.853.0940
Email: claymartin@keymail.com
Lorimartin@keymail.com