Embrace Your Inner Programmer: An Overview of the Critical Elements of Outstanding Programming

Melissa R. Martinez
Ba Research International LP – Cetero Research
Present Affiliation i3 Statprobe

ABSTRACT

The purpose of this paper is to describe the critical elements to be defined when developing a programming guideline. Best programming practices should be embraced to assure that programmers can develop, debug and maintain programs easily and efficiently, avoiding user confusion and wasted time deciphering code. This paper will be useful to SAS® users of all skill levels developing, particularly beginning users. The described programming practices can be applied not only to all SAS versions, but also to all programming languages. Concise, clear and consistent programs utilize thorough comments that explain the objectives and methods of the code, conclude all DATA steps, procedures, loops and conditional statements with proper END, RUN or QUIT statements and consistently use indents and a letter case style. Additionally, clear programs utilize data set and variable names that clearly convey their contents, explicitly specify data sets to be used in procedures and conclude all DATA steps and procedures with an END, RUN or QUIT statement are a few essential practices. Beyond these basics, further education in the use of the SAS macro language and in development of data-driven programming can substantially enhance programs. Programming teams should develop a programming guideline that outlines and adheres to good programming practices. The resulting programs will be concise, clear and thoroughly documented, which facilitates development, debugging and maintenance. This paper is not an exhaustive discussion of all techniques, but touches on many of the important and often overlooked aspects of practices that should form the basis of a programming guideline.

INTRODUCTION

An outstanding SAS program is much more than a collection of code that produces expected results. While accurate, working code is fundamental to a program, other elements and practices are just as important. Consistent documentation, appropriate variable and data set names, consistency in indenting and case style, explicitly specifying data sets used in procedures and concluding all DATA steps and procedures with an END, RUN or QUIT statement are a few essential practices. Beyond these basics, further education in the use of the SAS macro language and in development of data-driven programming can substantially enhance programs. Programming teams should develop a programming guideline that outlines and adheres to good programming practices. The resulting programs will be concise, clear and thoroughly documented, which facilitates development, debugging and maintenance. This paper is not an exhaustive discussion of all techniques, but touches on many of the important and often overlooked aspects of practices that should form the basis of a programming guideline.

DOCUMENTATION

Thorough documentation in the form of header and embedded comments is the most important aspect of a programming guideline. Every program should begin with a header that summarizes the program’s function and provides basic information about the program such as the program name, location, author, date created and modification history. Throughout each program, embedded comments should explain the intended outcome of each block of code and the method chosen to accomplish the goal. Comments should be written and revised concurrently with code so that the comments always match the code. In other words, never write code without also writing accompanying comments, and don’t revise code without revising the relevant comments. It is common practice to go back and write in comments after a program is completed, but this is inefficient, sometimes counterproductive and often not done due to competing priorities. A programmer pulled away from an unfinished, undocumented program will often have a hard time remembering where he or she left off, the planning and development steps already made, known problem spots or code inserted for testing purposes not intended to remain in the final program. An even more difficult situation can arise if a colleague who does not have the advantage of understanding the original thought processes and planning has to adopt the program and must try to decipher the code at face value. Save time and energy by making it a habit to document first and code second.

LETTER CASE

A programming guideline should include instructions for the consistent use of letter case. While different programmers have various preferences on letter case (all caps, all lower, or sentence case, for example), a programming team’s suite of programs should have a common appearance. Otherwise, the programs can appear messy and lead to confusion if it unintentionally appears that one case gives special significance. Consistent use of letter case does not necessarily mean using only one letter case in all circumstances. A programming team may find it desirable to use different letter cases for specific sections or types of information to make those elements...
easily identifiable. For example, macro variables and programs may be written with a different case than regular
code to call attention to the use of the macros. No matter which system of letter case is chosen, the appearance
should be uniform throughout different programs and among all programmers.

INDENTATION
The consistent and proper use of indentation gives code clarity, neatness and readability. A programming guideline
should define the type of indents to use and on which indent level to place specific code elements. Indents can be
anything from a single space to a specific number of spaces, but the same indent should be used consistently
throughout all programs. Tabs do not translate well to other systems, such as MVS or UNIX, and therefore should
not be used as the indent spacer. The programming guideline should also define relative indent levels for different
elements of programs. Minimally, specific indent levels should be defined for the following elements:

- beginning and ending code of DATA steps, procedures and loops
- comments
- code inside DATA steps, procedures and loops
- run-over code from programming lines too long to fit on the screen (or on paper, if applicable)

Figure 1 shows one example of a section of program that adheres to the following guidelines:

- Indent using five spaces
- Beginning and ending code lines at the same indent
- Internal code indented one level in from parent, whether the parent is a DATA step, procedure or loop
- Comments at the same indent as code it accompanies

FIGURE 1: INDENTING STRUCTURE EXAMPLE

```sas
/* Data set texmex_recipes is created by setting it to the data set
   recipe_collection, but only for observations whose genre is Tex-
   Mex. */
data texmex_recipes;
  set recipe_collection;
  where upcase(genre)= "TEX-MEX";
  /* The arrays ingred, quantity and measure will be used, and
     the max number of ingredients is 25. */
  array ingred{25};
  array quantity{25};
  array measure{25};

  /* Since guests from Up North will be attending the party,
   dishes whose ingredients include jalapenos will be toned down
   by reducing the amount of jalapenos by 80%. Also, all other
   ingredients except for oil will be increased fourfold to meet
   the party quantity needs. */
  do ing_count = 1 to 25;
    if upcase(ingred{ing_count}) = "JALAPENO" then do;
      quantity{ing_count} = quantity{ing_count} * 0.2;
      spice_warning = 1;
    end;
    else if upcase(ingred{ing_count}) not = "OIL" then do;
      quantity{ing_count} = quantity{ing_count} * 4;
    end;
  end;
run;
```

PROPER ENDINGS
It should be standard practice to end all DATA steps and procedures with the appropriate concluding code. While
failing to end loops and conditional iterations with an END statement will usually result in an error, the same is not
always true for DATA steps and procedures. When SAS is compiling a DATA step or procedure and encounters a
new DATA step or procedure, it will assume the previous step is complete and begin compiling the next.
Programmers should not rely on this default to produce expected results. For example, a call to a macro variable
created by the CALL SYMPUT routine during a DATA step will not properly resolve the first time the program is run if
the DATA step is not concluded with a RUN statement. Regular macro variables and programs are resolved during
compilation, but the CALL SYMPUT routine is a DATA step process, and the macro variable will not be created until
it encounters a RUN statement. Figure 2 shows a sample of code using the CALL SYMPUT routine that is improperly ended, and Figure 3 shows the output from this code the first time the program is run.

FIGURE 2: IMPROPERLY ENDED DATA STEP

```sas
/* Data set dinner_texmex is created by setting it to the data set
texmex_recipes where the recipes are to be included in the dinner
party (onmenu="yes"). */
data dinner_texmex;
  set texmex_recipes;
  where upcase(onmenu)="YES";
  /* The dinner_name macro variable is given the value "Wimpy
Tex-Mex". */
call symput('dinner_name',"Wimpy Tex-Mex");
/* The recipe name and course of dishes to be included at the
dinner party are printed. */
title "Menu for &dinner_name Dinner Party"
proc print data=dinner_texmex noobs;
  var recipe_name course;
run;
```

FIGURE 3: SAS OUTPUT FROM FIGURE 2

```
Menu for &dinner_name Dinner

<table>
<thead>
<tr>
<th>recipe_name</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strawberry Margaritas</td>
<td>1</td>
</tr>
<tr>
<td>Guacamole with Tricolor Tortilla Chips and Salsa</td>
<td>2</td>
</tr>
<tr>
<td>Chicken Tortilla Soup</td>
<td>3</td>
</tr>
<tr>
<td>Beef Fajitas</td>
<td>4</td>
</tr>
<tr>
<td>Flan with Caramel Sauce</td>
<td>5</td>
</tr>
</tbody>
</table>
```

If this program were run a second time, the macro variable ‘dinner_name’ would be resolved when SAS encounters the RUN statement at the end of the PRINT procedure. However, this is clearly not the proper way to resolve macro variables. Simply adding the RUN statement at the end of each DATA step and procedure will reliably and properly conclude the steps and procedures and will avoid unresolved macros as illustrated in this example.

PROPER DATA SET AND VARIABLE NAMES

Data set and variable names should be clearly indicative of their contents, even when that means a long name must be used. Later versions of SAS generally allow data set and variable names to contain up to 32 characters (see SAS Help and Documentation for a complete list of naming rules). A programming guideline should allow for programmers to make use of that space to create data set and variable names that make sense and can be understood by other users of the program, so long as doing so is not prohibited by related software (such as version control software), company or team policy, client specifications or regulatory requirements. Nondescriptive variable names such as “x” and “y” give no indication of what they represent, creating the potential for programmers to lose track of variables and waste time by trying to trace nondescriptive variables throughout the program. While it may take longer to type “destination” or “employee_ID”, these variable names are clear and readily comprehended. Create a culture of clarity by encouraging the use of descriptive data set and variable names in the programming guideline.

Each data set in a program should have a unique name. Sometimes a data set is written over itself several times using code such as:

```sas
data final_table;
  set final_table;
  ...;
run;
```
This can cause multiple problems because the data set is never cleared or refreshed. When the data set is created it may not contain errors, but if there are five subsequent DATA steps each setting back to the data set, it is much harder to capture errors that may occur. In a program that uses unique data set names, it will be obvious if a data set was not created properly because the log will indicate that the data set does not exist or that it contains zero observations when another DATA step or procedure attempts to use it. Reused data sets may still exist and produce output, yet the information in the data set could be very different from what is expected. Additionally, there may be times when interim data sets need to be viewed to see what is occurring during a DATA step. With unique data set names this is easily done, but when a data set is rewritten several times it can be difficult to track down interim steps without running selected portions of code. However, if portions of the program are highlighted and run and the initial creation of the data set is not included, the data set could end up with more observations than expected, produce undesirable results and not accurately exhibit what the DATA steps actually do when the entire program is run. Because of the numerous problems with reusing data set names within a program, a programming guideline should require unique data set names for all data sets within a program.

**DATA SET SPECIFICATION**

A programming guideline should dictate that the data set used by SAS procedures always be explicitly defined. If no data set is specified in a procedure, SAS will attempt to use the last data set that was accessed. While the structure of some programs may be such that the default will produce expected results, programmers should never rely on the default to capture the correct data set. An error in the creation of a data set intended to be used for a procedure may not be evident if the procedure still runs without error. The procedure will use the last valid data set accessed, which may produce output from the procedure using the wrong data set. If the data set is explicitly defined in a procedure, a problem with the intended data set will be immediately evident. Additionally, suppose DATA steps were added to a program between an existing DATA step and a procedure implicitly using that data set. The procedure would then be attempting to use the data created by the new DATA step, which may not produce the expected results. Also, in the event that portions of code are highlighted and run, or portions of code are excluded before running a program, the data set a procedure uses may not be the desired one. Procedures should always explicitly define the data set to use to avoid unexpected results or errors.

**CONCISENESS**

A programming guideline should express a preference for developing and maintaining concise code. If a programming objective can be accomplished in three steps, it should not be extended to eight steps. Different conditional programming options, such as if/then/else, select and where statements, offer different advantages and should be chosen based on clarity and conciseness. It is usually unnecessary to create variables for intermediate calculations that will never be used on their own. Further, only one statement should be on each line, because multiple statements on a single line can be confusing or obscure. Adhere to the “one semicolon per line” rule. All programming objectives do not necessarily need to be accomplished in one program; break up portions of code into different programs where appropriate. For example, when programming for a project requiring dozens of different tables, try grouping related tables into a single program, but don’t try to produce every table from one program. Programs that are excessively lengthy can be very difficult to develop, debug and maintain.

Figure 4 is an example of actual code that contains unnecessary extra steps as well as unclear variable names. Figure 5 shows an example of portions of the actual replacement code that was rewritten more concisely and clearly, adhering to that team’s programming guidelines. Notice the following problems with the programming in Figure 4:

- No comments to document purpose or techniques
- No use of indentation
- Generic/nondescriptive data set and variable names
- PRINT procedure does not specify the data set to use
- Three new data sets created unnecessarily
- Several interim variables created for one-time use
- Several different format statements used instead of a single one
- Multiple statements on a single line
- No RUN statements on DATA steps
FIGURE 4: PORTION OF ACTUAL PROGRAM NOT USING GOOD PROGRAMMING PRACTICES

```sas
DATA FIVE; SET FOUR;
DATA SIX; SET FIVE;
XWLL = ' ';
XWLU = ' ';
XWLL = WLLWPCT;
XWLU = WLUPPCT;
XWI = '(' || XWLL || ';' || XWLU || ')';
DATA SEVEN; SET SIX;
XTTL = ' ';
XTTU = ' ';
XTTL = T1SLWPCT;
XTTU = T1SUPPCT;
XTTI = '(' || XTTL || ';' || XTTU || ')';
PROC PRINT LABEL; VAR TESTMEAN REFMEAN RATIO XTTI POWER PVALUE;
ID TITLE;
LABEL TESTMEAN = 'TEST LEAST SQUARES MEAN'
REFMEAN = 'REFERENCE LEAST SQUARES MEAN'
RATIO = '100* TEST/REFERENCE RATIO'
XTTI = '90% CI'
POWER = 'POWER OF ANOVA'
PVALUE='P VALUE';
FORMAT RATIO BEST4.;
FORMAT  POWER 7.5;
FORMAT TESTMEAN BEST8.; FORMAT REFMEAN BEST8.; FORMAT STE BEST8.;
```

FIGURE 5: CODE FROM FIGURE 4 REWRITTEN ADHERING TO THE TEAM’S PROGRAMMING GUIDELINE

```sas
DATA NON;
SET NONTRANS;
--------------------(other SAS code)---------------------
/* THE 90% CONFIDENCE LIMITS ARE CALCULATED. THE T VALUE IS
FOUND IN THE FIRST STATEMENT. THEN THE LOWER AND UPPER 90%
CONFIDENCE LIMITS ARE FOUND BY SUBTRACTING AND ADDING
(REPECTIVELY) THE T VALUE TIMES THE STANDARD ERROR TO THE TEST
MEAN, AND THEN DIVIDING THE RESULTS BY THE REFERENCE MEAN.
FINALLY THE RESULT IS MULTIPLIED BY 100 TO GET THE LIMITS. THE
CONF_INTERVAL VARIABLE IS CREATED TO PRINT THE CONFIDENCE
INTERVAL NEATLY IN THE FORM (LOW; UP), WITH NO MORE THAN THREE
SIGNIFICANT FIGURES. */
T90 = TINV(.95,DF);
LOW_90PERCENT = 100 * ((TESTMEAN - T90*STDERR)/REFMEAN);
UP_90PERCENT = 100 * ((TESTMEAN + T90*STDERR)/REFMEAN);
CONF_INTERVAL = '(' || PUT(LOW_90PERCENT,BEST4.) || ';' ||
                 PUT(UP_90PERCENT,BEST4.) || ')';
--------------------(other SAS code)---------------------
RUN;
/* THE VARIABLES LISTED IN THE VAR STATEMENT ARE PRINTED. THIS PAGE IS
A CONCISE SUMMARY OF THE NON-TRANFORMED DATA. */
PROC PRINT LABEL DATA=NON;
   VAR TESTMEAN REFMEAN RATIO CONF_INTERVAL POWER PROBT;
   ID DEPENDENT;
   FORMAT RATIO BEST4. TESTMEAN REFMEAN BEST8.;
RUN;
```
MACROS AND DYNAMIC PROGRAMMING
In addition to creating a programming guideline that will institute uniform, clear, concise programs, programming teams should also take advantage of the SAS macro facility and dynamic, data-driven code. These tools will make programs less repetitive, easily reusable among different data sets and projects and less open to user error. Since global macro variables are available across different data sets, procedures and programs, they can reduce redundancy and the potential for human error. For example, the same title may be used by all procedures throughout several different programs. Without the use of macros, this title must be written into each program and then updated in every program when changes are made. A better solution is to define the title using a global macro variable and simply call that macro variable in every place where the title is necessary. Not only will this assure that the same title is used in each of the programs, it will also require a change only to the macro variable definition to change the title in every place it occurs. This is just one example of the many uses of macro variables.

Macro programs are sets of code that can be stored and called from other programs. They can hold anything from a few calculations to a set of DATA steps and procedures. Macro programs are especially useful for code that is used in several different instances; they reduce redundancy and increase efficiency.

Data-driven code determines information from the input data set instead of hard-coding variables. For example, a data-driven program can determine the number of variables that will be stored in an array, save that number as a macro variable, and then create the array using the macro variable. Data-driven programming does require more planning, design and advanced programming techniques, but the result is programs that are versatile, efficient and less open to human error.

CONCLUSION
Quality programs are created when a programming team implements a programming guideline that embraces a commitment to clarity, conciseness and consistency. Creating a culture that encourages good programming practices may take time while programmers develop new habits, but the result will be neater, more efficient programs with less development time spent deciphering and debugging unclear code.

ACKNOWLEDGMENTS
Thank you to all who reviewed, inspired, and critiqued the ideas presented in this paper, including Debra Anderson, Gwen Flory, Ben Johnson, Danny Martinez, Mark Meyers, Vijay Moolaveesala, Hafeez Rahman, Deborah Sampson, Ph.D., Jennifer Savage Sales, Jeff Smith and Bob Turner.

CONTACT INFORMATION
Your comments and questions are valued and encouraged. Contact the author at:
Melissa R. Martinez
i3 Statprobe
1250 Capital of Texas Highway, Building 1, Suite 250
Austin, Texas 78746
Work Phone: 512-347-2776
E-mail: Melissa.Martinez@i3statprobe.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.