The Elements of SAS® Programming Style

Frank C. Dilorio, ASG, Inc., Cary, NC

"Style is a simple way of saying complicated things"
Jean Cocteau

"GOOD" PROGRAMMING STYLE ...

1. Results in correct performance of the required task
2. Correctly balances machine and programmer resources without compromising the code's readability/maintainability. The balance is determined by program and system context.
3. Highlights program and system flow and logic
4. Results in effective (and usually resource-efficient) code

THIS PAPER

1. Identifies coding habits and conventions which encourage the development of "good" programs. There are no absolutes, just "strongly held opinions" based on real-world experience.
2. Emphasizes Base SAS (AF, GRAPH, *et al.* not dealt with *directly*)
3. Organization: pre-coding issues; general principles; DATA step; PROCs, debugging

PRE-CODING: DESIGN

1. **Identify Task Components.** Sources are formal specifications as well as "oral history." Determine whether multiple programs are required: would a single, standalone program be too large? Does the single program perform too many functions (some of which may later need to be commented out?). Be aware of the coding overhead incurred and the increased need for consistency between programs (entity names, dataset structure, OS dataset and directory structure, for example). There are similar considerations when using multiple datasets.
2. **Clearly Understand the Data.** For both "raw" and SAS datasets, be aware of sort or grouping order; dataset engine and associated limitations (e.g., no random access [POINT option in SET statement] for transport or tapes); missing value representation (a ".", or other convention, such as 9-filled fields, for example); date value representation, identification of fields used for linking tables.
3. **Avoid Monolithic Code.** DATA steps with many (many = ???) statements are seldom needed. Consider usage and impact of %INCLUDEd files, macros, macro variables, and SQL on program size, readability, maintainability and performance.
4. **Avoid Excessive-Nesting of Code.** %INCLUDEs within %INCLUDEs, macros within macros are hard to read, devilish to debug, and can usually be coded better.
5. **Develop Incrementally.** Core, critical features first, then options and tuning. Beware of the possibility of "second system effects"/"featuritis."
6. **Assume Change.** Build code which is readily expandable. Change in the task specifications does not necessarily imply bad planning or design.
7. **Consider Tuning Options Early On.** These features can have a significant impact on program/system design. Decide whether to use indexing, dataset compression, the stored program facility, *et al.*

PRE-CODING: ENVIRONMENT

1. **Consider Execution Mode Alternatives.** Consider mixing Display Manager (DM), interactive, and batch modes during development, regardless of mode for production program. DM often best for testing smaller pieces of code. However, be aware of "carryover" of macro variable settings, titles, and footnotes from multiple executions of code in the same DM session: take care to reset macro variables, delete temporary datasets, and so on. Also be aware of differences between interactive and batch mode settings for options such as page and line size (PS and LS, respectively).
2. **Be Aware of SAS' "Portability".** SAS code is *very*, not *totally*, portable. Consider differences between versions and operating environments. Non-SAS issues: file organization; toolset; command file (VMS COM, TSO CLIST, MVS PROC, etc.) duplication incurs additional overhead. Also beware of collating sequence (EBCDIC v. ASCII) differences and the effects on sort order.
3. **Use of Operating System Utilities.** Don't confine yourself to SAS tools (Display Manager, etc.). Organize files by subdirectory. Compress with tools such as ZIP, ARC (in the absence of the ZIP/ARC engine, you must un-ZIP/ARC before using the dataset if you cannot use UNIX, UNIX-like pipes to decompress on the fly). Change management with Panvalet, VMS CMS). String searches with Unix GREP, VMS SEARCH, DOS/Norton TS, for example.
4. **Assess the SAS Environment.** Know what add-ons are available (AF, ACCESS, etc.); SAS version (differences are significant: macro capability, SQL, value-based keys, procedure options, and so on); location of system utility macros, location of common "scratch" areas; print and "liveware" resources. If developing for multiple platforms, investigate these issues for all systems.
5. **Periodically Review the Literature.** Browse Technical Reports P-222 and P-245 (and others) for new and enhanced features/PROCs. Items in 6.07 and 6.08 such as SQL dictionary views and tables ("meta-data"), OTHER=[*format*] in PROC FORMAT, the INDEX dataset option, and the *-l/-r/-c* format modifiers can save significant amounts of coding effort and make you rethink the way you approach the task at hand. Never assume you know everything about a PROC! Know when features become irrelevant (e.g., looping SET isn't more efficient in Version 6.06[+]).

CODING PRINCIPLES: GENERAL

1. **Chose Sensible Entity Names.** Variable and dataset names should readily convey their content - PCTCHG88 is more meaningful than CHG. Avoid "cute" names - ENORMOUS may be a funny name for a dataset but will probably become tiresome and uninformative after a while. Know when to *avoid* this rule - it may be easier to key survey questions to their numbers rather than assign a name to them. Q23A clearly indicates Question 23, Part A, for example.
2. **Chose Data Types Carefully.** Compare character, "numeric looking" character (numbers used only for categories, such as FIPS state codes), "character looking" numeric (dates with

embedded slashed). Character values are, generally, more efficiently handled than numeric.

3. **Avoid "Clever Coding."** Resist the impulse to "clever code" if the program will be inherited by a novice programmer or if the program needs to be modified frequently. If "clever code" is your only recourse, document it thoroughly (see "Coding Principles: Documentation," below).

Obscure, but legitimate
```
x = a>b * c>d;
```

Better, clearer
```
if a>b & c>d then x =1;
   else           x = 0;
```

4. **Don't overcode.** Mix DATA steps, PROCs (especially SORT, MEANS, SQL). *When* to do this is, unfortunately, largely a matter of intuition and experience. 3GL programmers new to SAS tend to have an especially tough time with this feature of SAS programming.

5. **Know the defaults.** Be aware of SAS's assumptions about your work environment. Remember ... defaults are "the vain attempt to avoid errors by inactivity."

   In procedures: computational method; display options; impact of missing values

   In DATA step: variable length and order; timing of observation outputting (impact of OUTPUT, RETURN, DELETE, REMOVE statements); impact of missing values on calculations

   Both: avoid reliance on the "rule of last use". Always specify DATA= or its equivalent in PROCs. Avoid using _LAST_ in SET/MODIFY/MERGE/UPDATE statements.

   Elsewhere: macro, %INCLUDED code expansion and display; centering and other aesthetic issues; default/reserved LIBNAMES.

6. **Be aware of "multiple use" statements.** WHERE, FORMAT, LABEL, BY may be used in DATA steps or PROCs. But their effect may vary by location (e.g., using a user-written format to control order of BY variables in SORT has no effect, nor is a warning or note printed; WHERE as dataset option affects execution differently than if used as a standalone statement in the DATA step).

Coding Principles: Presentation

1. **Use Blank Space to Your Advantage.** Indent loops and IF-THEN-ELSE sequences; use multiple statements per line to clarify logic; try to use only one statement per line.

Difficult to read
```
do i = 1 to 20;
 x(i) = y(i+1);
if x(i)> 20 then y=3;else y=.;
end;
```

Better, clearer
```
do i = 1 to 20;
   x(i) = y(i+1);
   if x(i) > 20 then y=3;
      else           y=.;
end;
```

```
* Where's the error? ;
tot = ((x1*rt1)+(x1*rt2) +
 (x3*rt3)) / (3 * adjtot);
input (hi_1-hi_20 low1-low20
      meas_1-meas_20)(20*5.
      20*3. 20*5.);
```

** Error is easily spotted here;*

```
tot = ((x1 * rt1) +
       (x1 * rt2) +
       (x3 * rt3)) /
      (3 * adjtot) ;
input (hi_1   -hi20  )(5.)
      (low_1 -low20  )(3.)
      (meas_1-meas_20)(5.);
```

2. **Break the Rules Occasionally.** Deliberate violation of the above practice can draw attention to unfinished or questionable parts of the program.

```
if rate > 10 then do;
   put 'rate ok for ' patid= $8.;
   end;
   else do;

***** what to do with other rates? *******;

        end;
```

3. **Clearly Separate Units of Work.** Use RUN/QUIT statements after DATA steps and PROCs. Use PAGE statement to force page feeds in the SAS Log between significantly different sections of the program. RUN/QUIT are *required* in interactive environments. Their optional use in batch is a good habit to develop.

4. **Use Special Attributes Sparingly.** In SOURCE and notepad catalog entries: sparing use of special attributes (color, reverse video) draws attention to important parts of the program. Don't rely on these attributes: the sections should also be noticeable and highlighted when printed.

5. **Shrink, Modularize Code Via Macros.** Eliminate blocks of similar, repeated statements.

Hard-coded
```
data;
...
piece = scan(line,1,'');
if piece='Attr:' then do;
   _nattr + 1;
   output;
   end;
   else if piece='Format'
        then do;
        _nfmt + 1;
        output;
        end;
```

Using Macros
```
%macro readstr(var,str);
if piece= "&str." then do;
   _n&var. + 1;
   output;
   end;
%mend;

data; ...
piece = scan(line,1,'');
%readstr(attr, Attr:);
else %readstr(fmt,Format);
```

Coding Principles: Program Execution

1. **Run Production Code Error-Free.** All notes and warnings should be readily explained (items such as missing values generated by operations on incomplete data, unreferenced labels, unitialized variables, mismatched data types).

2. **Examine Output Dataset Dimensions.** Output dataset dimensions (number of rows and columns) should be examined for "ballpark" (or exact, depending on the application) accuracy.

3. **Check Missing Value Distributions.** Use PROC FREQ, PRINT, or REPORT or PUT statements to examine the distribution of missing values for critical variables.

4. **Reconcile SAS Log to Output.** Don't assume that because your output looks reasonable that there is nothing worth noting in the Log! Some procedures print warnings on the output, others in the Log. It is your responsibility to look in both places for the warnings.

Program
```
proc chart;
vbar dept / discrete;
run;
```

Output
```
Display is a horizontal, rather than vertical
chart.
```

SAS Log
```
Contains the message "Vbar for DEPT is not
possible [due to too many levels of the
variable]."
```

Coding Principles: Documentation

1. **Locate Documentation Throughout the Program.** Use comments in a header, before a DATA step or procedure, at start of new section of a DATA step, within or before complex or important statements. Also, anywhere where memory-jogging is helpful (don't slap a PostIt note on a listing - write a comment in the program!).

2. **Avoid Overcommenting.** Comments can be taken to excess. Don't stating the obvious (e.g., * Now multiply A by B;) or use vacuous remarks (e.g., * This SQL call is ugly as sin;).

3. **Document Features External to the Program.** Include non-programming references, such as external references ("handling of duplicate records done as per meeting with Clinical on July 5th", "program uses %INCLUDEd code from files ...,"); oddities ("following workaround found in Version 6.06 usage note 12387"); authorship and revision history.

4. **Comment Throughout Development.** Insert comments throughout the course of program development. During early stages of development, only insert critical items. Add headers, narrative, and so on later on, when code is more stable. Be aware of the potential for code-comment mismatches, especially when extensive commenting is done throughout development.

5. **Use Labels.** Variable and dataset labels improve readability of PROC CONTENTS output. They can clarify issues of measurement (e.g., WGT is "Weight, in kilograms") and content (e.g., dataset CUM95 is "Cumulative totals for fiscal year 1995").

DATA Step: General

1. **Group Unexecutable Statements.** In longer DATA steps, consider grouping related unexecutable statements: DROP, KEEP, LENGTH, RETAIN, ATTRIB, RENAME. Note the possible impact on variable order of the relocating of these statements.

2. **Always Declare Character Variables.** Use LENGTH or ATTRIB statements to explicitly assign lengths to character variables, especially when they are created by a call to a character-handling function. The function *may* return the correct length without the declaration, but who wants to find out the hard way, or to remember all the rules for using the functions? Declaring also ensures consistency across versions and platforms.

3. **Clearly identify RETAINed variables.** They are handled differently than other variables, so make them easy to recognize (possibly by starting or flanking with underscores). Employ similar treatment for any other "special" variables (counters, error flags, etc.).

4. **Use ARRAYs.** Identify repeated code, then try to minimize it by using ARRAYs:

Without arrays
```
diff1 = m2 - m1;
diff2 = m3 - m2;
...
diff10 = m11 - m10;
```

Similarities exploited by using arrays
```
array diff{10};
array m{11};
do i = 1 to 10;
   diff{i} = m{i+1} - m{i};
end;
drop i;
```

No matter how many differences required, the array-based solution will always require the same number of statements. The same cannot be said for the array-less solution on the left: 50 differences would require 50 statements. The potential for miscoding increases rapidly.

5. **Avoid Unnecessary DATA Steps.** A DATA step should not consist solely of KEEP, DROP, and/or WHERE statements. These activities can be performed "on the fly" in PROCs by using dataset options and variable selection statements. Instead of

```
data males;
set master;
where gender = 'M';
keep id age race income1-income5;
run;

proc print data=males;
run;
```

use the following, which is more compact and machine-friendly:

```
proc print data=master(where=(gender='M');
keep id age race income1-income5;
run;
```

6. **Understand How the Supervisor Works.** If complex dataset combinations are required, or if you use the macro language you should be aware of how the SAS supervisor works. A good discussion of this is in Henderson, *et al.*, SUGI Proceedings, 1991 (and elsewhere!). Supervisor knowledge is particularly important when using multiple SET or MERGE statements in a DATA step.

DATA Step: Calculations

1. **Simplify Complex Calculations.** Use intermediate variables, breaking the calculation into several statements. Use multiple lines, blank space, and parentheses to draw attention to the calculation's elements. Use parentheses so you don't have to remember the evaluation hierarchy (powers, then division? right to left or left to right? ...) Paren's also highlight and reinforce program logic - you can identify operations in complex calculations at a glance.

2. **Simplify Expressions.** Consider implementing complex logic with DO groups:

Acceptable
```
if (cond1 | cond2) &
  (cond3 & (cond4 |
   cond5)) then ...
```

Better
```
if cond1 | cond2 then do;
   if cond3 & (cond4 | cond5)
      then ...
```

3. **Use Functions.** Functions simplify prosaic (counting, descriptive statistics) as well as complicated tasks. Use them whenever possible. Nest function calls sparingly.

Excessive function nesting
```
if reverse(scan(reverse(address),1) in
   ('NY','NJ','CT') then do;
```

More statements, but easier to follow
```
length r_add r_add1 r_add2 $30;

r_add  = reverse(address);
r_add1 = scan(r_add,1);
r_add2 = reverse(r_add1);
if r_add2 in ('NY', 'NJ', 'CT') then do;

drop r_add r_add1 r_add2;
```

Note the use of the second REVERSE: we could have used 'YN', 'JN', AND 'TC' in the IF statement's expression but chose to make the program's intent a bit more obvious.

4. **Avoid Mixed Data Type Calculations.** They may not be as inefficient in Version 6.xx as Version 5.xx. The very *suggestion* of their use, though, should still give you shudders. Use PUT and INPUT functions to get operands into the correct data types.

5. **Use Date, Time, and Date-Time Constants.** These special constants are easier to both read and write than their actual, "raw" unformatted values. Use '2jul93'd instead of 12241, '12:15't instead of (12*60*60) + (15*60).

6. **Consider Different Recoding Strategies.** Select among: IF-THEN-ELSE sequence of assignment statements; user-written format followed by PUT function; creation and subequent use of data-driven format via CNTLIN dataset. If you use user-written formats and the new variable must be numeric, post-process the PUT function output with the INPUT function.

Hard-coded grouping
```
if area in (1,2,4,5) then region = '1';
   else if area = 3 then  region = '2';
   else if area >= 6 then region = '3';
   else                   region = '4';
```

Format-driven assignment
```
proc format;
value areareg 1,2,4,5 = '1'
              3       = '2'
              6 -high = '3'
              other   = '?';
run;

data; ...
region = put(area, areareg.);
```

Comment: Here, as in some of the other examples, the more effective and easier to read example requires more coding effort than the less desirable techniques. Note that if AREAREG is placed in a permanent format library it can be shared across programs, thus aiding the reliability and consistency of the recoding.

7. **Eliminate Unnecessary Operations.** Review code for operations which need to be performed only once during processing of a dataset (example 1, below). Also, identify needless repetition of assignment statements (example 2) and unnecessary I/O (example 3). These examples lend credibility to the argument that well-crafted, elegant code is often machine-efficient code as well.

☹ NOW needlessly calculated in each pass of the DATA step
```
data ...;
now = today();
```

NOW calculated only once
```
data ...;
retain now;
if _n_ = 1 then now = today();
```

☹ UPCASE function needlessly called in each statement
```
if         upcase(st) = 'VA' then ...
   else if upcase(st) = 'WV' then ...
      else if upcase(st) = 'PA' then ...
```

UPCASE function called once. Result is used in comparisons
```
upst = upcase(st);
if         upst = 'VA' then ...
   else if upst = 'WV' then ...
   else if upst = 'PA' then ...
```

☹ INPUT reads many variables, only to discard them
```
input id v1-v100;
if ^(1 <= id <= 120) then output;
```

INPUT reads the raw data incrementally
```
input id @;
if ^(1 <= id <= 120) then do;
   input v1-v100;
   output;
   end;
```

8. **Overwrite Variables Sparingly.** Be careful when overwriting variables, especially if you won't be able to re-create the old value from other variables in the dataset. Overwriting also makes debugging difficult.

9. **Use IN and NOTIN.** Reduce coding effort by using the IN and NOTIN operators. These expressions can be used with more conventional comparisons (second example, below).

Acceptable
```
if x=1 | x=5 | x=7 then ...
if 1 <= x <= 3 | x=8 | x=10 | x=11 then ...;
```

Better
```
if x in (1,5,7) then ...
if 1 <= x <= 3 | x in (8,10,11) then ...;
```

10. **Use IF-THEN-ELSE.** Using IF-THEN-ELSE rather than repeated IF-THEN statements results in faster execution. It is also a clearer expression of program logic.

Acceptable
```
if x = 1 then new = 'AFR';
if x = 2 then new = 'EUR';
if x = 4 then new = 'SUB';
if x = 5 then new = 'ANT';
```

Better
```
if         x = 1 then new = 'AFR';
   else if x = 2 then new = 'EUR';
   else if x = 4 then new = 'SUB';
   else if x = 5 then new = 'ANT';
```

DATA STEP: FLOW OF CONTROL

1. **Jump Carefully.** Use GOTO, LINK, CONTINUE, and LEAVE sparingly. If possible, place all LINKed code at the bottom of the DATA step. Note, with a comment, the use of these and any other statements (STOP, ABORT) which interrupt the normal, top-to-bottom flow of DATA step execution. This is addressed below.

2. **Highlight Flow Interruptions.** Use comments to highlight the interruption of normal, top-down flow (GOTO, LINK, DELETE, RETURN, STOP, ABORT). This is especially important in long DATA steps, where these critical actions sometimes get buried in the sheer mass of code. In the following program fragment, the asterisks are purely an aesthetic consideration - they draw

attention to the STOP statement and are a visual cue that something important is happening in the DO group.

```
if _nerrors > &errlimit. then do;
   put 'Error limit exceeded. DATA step
        halted at observation ' _n_ -l
        comma5.;
   stop; **********************************;
   end;
```

3. **Beware the "Subsetting" IF.** Know why quotation marks are needed around the "subsetting" IF. IF without a THEN clause merely indicates the DATA step should continue executing. *Sometimes* this means outputting, sometimes not. The presence or absence of OUTPUT statement controls apparent subetting behavior.

Subsetting IF

```
data ...;
...
* Implied OUTPUT is executed only because
  we're at the bottom of the DATA step;
if condition;
run;
```

"Subsetting" IF

```
data ...;
...
* Implied OUTPUT does not occur because an
  explicit OUTPUT is used in the step
  (below). This statement is merely the first
  test required for outputting.;
if condition1;

* Only here are observations actually
   written.;
if condition2 then output;
drop t1-t25;
run;
```

Using Procedures

1. **Let procedures Do the Work.** Whenever possible, let procedures do the dirty work, both for calculation and display of data. Consider, for example, why you are writing a multicolumn report with PUT statements in a DATA step when, in Version 6.06 and above, you can use PROC REPORT.
2. **Use new PROCs gradually.** Take as many defaults as possible at first, then add options and assess their impact. If it is available, consider using SAS/ASSIST as a program generator for new or seldom-used procedures.
3. **Reconcile Log and output.** (yet again — see "Coding Principles: Program Execution," above).
4. **Use the DATASETS Procedure.** This avoids unnecessary I/O if you need to modifying variable formats and labels, renaming entities. No need to pass the data if all you are doing is changing information in the "header" portion of the dataset.
5. **Use Output Datasets.** Take advantage of the efficiencies realized by use of output datasets: CORR datasets can be used as input to REG, GLM, and other linear modeling procedures. MEANS datasets are often useful as an intermediate step in data management and reporting calculations.
6. **Be Aware of Tradeoffs.** Understand the tradeoffs between procedures with roughly similar capabilities: REG v. GLM, CLUSTER v. FASTCLUS, GLM v. VARCOMP v. ANOVA, etc.
7. **Use SQL.** You should develop enough familiarity with SQL to know when it is more effective and efficient than traditional DATA step/PROC coding sequences.

Debugging

1. **Write Well and You Don't Need to Debug.** Well-written code *preempts* many syntactical and logical errors.
2. **Use PROCs to Debug DATA Steps.** Debugging DATA steps need not be confined to the DATA step itself. The process often involves a mixture of DATA step coding and procedures (PRINT, FSxxxx, FREQ, and DIR/LIBNAME windows in Display Manager). Version 6.07(+) has DATA step debugger similar in look and feel to that of AF. It may be easier to use procs and/or Display Manager windows to examine the data: frequency of "bad" variable using FREQ, histogram of identifiers using CHART, quick examination of subsets of the data with FSVIEW, dump of all or part of the data using PRINT or REPORT.
3. **Use PUT and LIST Statements.** PUT, LIST, and the _ERROR_ and _ALL_ variables can be used to display variable values and raw data during DATA step execution. Consider toggling the display with a macro variable:

```
* DBUG can identify level (i.e. amount of debugging output, not
  simply whether or not to write output.;

%let dbug = 2;

data master;
...
if &dbug. > 1 then put ... ;
if &dbug. > 3 then put ... ;
```

4. **Make Messages Informative.** A cryptic or inaccurate diagnostic message is useless, maybe even counterproductive. Take the time to make messages informative — attach descriptive text to values being written out; use indentation to highlight events taking place in DO loops.

Poor

```
if condition then put index=;
```

Better

```
if condition then put 'Index out'
   ' of range in inner loop: ' index;
```

5. **Yet Again ... Reconcile Log and Output.** (see "Coding Principles: Program Execution," above). Look for messages about missing values being generated, character-numeric conversions, uninitialized variables, invalid input data, subscript problems.
6. **Use System Options to Control Debugging Output.** System options to be aware of during debugging: ERRORS, MPRINT, DSNFERR, FMTERR, ERRORABEND, SOURCE2, MLOGIC, MAUTOSOURCE.

References

1. Sall, "Elements of SAS Style" Tech Report A-105 (1978)
2. SAS-L programming style discussion, Fall 1988
3. Dilorio, "Good Code, Bad Code: Strategies for Program Design," NESUG 1989
4. Dilorio, *SAS Applications Programming: A Gentle Introduction*, 1991.
5. Dilorio, *Quick Start to Data Analysis with SAS, 1995.*

Questions? Comments?

Frank Dilorio
ASG, Inc.
2000 Regency Pkway, Suite 355
Cary, NC 27511

919-467-0505 (work)
919-467-2469 (fax)
fcd@asg-inc.com