Paper 241-28

SAS® Programming Conventions

Lois Levin, Independent Consultant, Bethesda, Maryland

ABSTRACT

This paper presents a set of programming conventions and guidelines that can be considered in developing code to ensure that it is clear, efficient, transferable and maintainable. These are not hard and fast rules but they are generally accepted good programming practices. These techniques can be used in all types of SAS® programs, on all platforms, by beginners and experts alike.

INTRODUCTION

Conventions are arbitrary choices to be used in coding programs. They are not efficiency techniques per se, although they should result in efficient development and execution. They are not correct or incorrect methods, although they are often commonly accepted practices. They are not the author's favorite way of doing things. They are objective criteria to determine one approach to writing code. They represent an agreement to write in a certain way by all the programmers in a group.

PURPOSE OF PROGRAMMING CONVENTIONS

The implementation of standard programming techniques results in:

- Clarity of program code
- Maintainability of code
- Survivability of code
- Ability to transfer code among members of the group
- Ability to transfer code among other programs
- Ability for a knowledgeable outsider to read code and understand it.

CATEGORIES

Conventions will be defined for the following areas:

- Naming Conventions
- Compatibility
- Documentation
- Appearance
- Efficiency
- Maintainability

NAMING CONVENTIONS

Establishing naming conventions for programs, libraries, datasets, and variable names enable each of these to be identified and categorized easily. It organizes all the elements of the system. It enables code and data to be exchanged among programs without extensive rewriting.

Programs can be named sequentially indicating their sequence.

```
e.g.  ABC1.sas
      ABC2.sas
```

Or they can be named logically, but then their sequence must be well documented.

Libraries and catalogs should be named logically and include the letters LIB or CAT in the name.

```
e.g.  MACROLIB
      FORMATLIB
```

Dataset names will depend on the application and the facility standards.

Variable names should be logical. Naming conventions may be determined for specific applications or for department standards.

The important thing is to develop a standard for all components of a system so that all of the programmers can understand each other's modules and all the modules are compatible.

Abbreviated Names

In Version 8, long names are permitted, but if you wish to shorten names, truncate rather than abbreviate.

```
e.g. Use:
      TRANSDAT

      instead of :
       TRANSDTE
```

IN= Names

Name each IN variable using IN plus the first letter of the dataset name.

```
e.g. MERGE A(IN=INA) B(IN=INB);
```

Array Names

Begin array names with an underscore to distinguish them from regular variable names.

```
e.g. ARRAY     _X (10)  X1-X10;
```

Format Names

If a format applies to only one variable, then name the format with the variable name plus FMT.

e.g. FORMAT PROPTYPE PROPFMT.;

or

name the format with a description plus FMT.

e.g. FORMAT PROPTYPE MISSFMT.;

COMPATIBILITY

Sometimes programs will be transferred to other platforms. It is possible to write them so that the transfer requires a minimum of conversion. Special features of one system can be generalized so that they can be easily translated to work on another system. Compatibility also includes compatibility with other programs in a group.

Environment Variables

On Unix, use environment variables to declare standard path or library names. These statements can be stored in a separate program that can be %INCLUDE-ed in all application programs.

```
%let saspath       = %sysget(APPL_SAS);
%let macropath     = %sysget(APPL_MACROLIB);
%let formatpath    = %sysget(APPL_FORMATLIB);
%let datapath      = %sysget(APPL_DATA);
%let logpath       = %sysget(APPL_LOG);
%let outpath       = %sysget(APPL_OUTPUT);
```

On a mainframe, convert the environment variable names to DD names.

Log and Output

Send log and lst files (or other output) to separate directories/libraries using standard path/library names.

```
e.g.  PROC PRINTTO LOG=&logpath/abc1.log;
      PROC PRINTTO LST=&outpath/abc1.lst;
```

Options

- Do not hard-code LINESIZE and PAGESIZE.

  Reason: They vary by printer.

- Do not use COMPRESS as a global option.

  Reason: COMPRESS does not always save space. Decide to compress based on the characteristics of each individual dataset.

Character Sets

Do not use non-printing characters or special fonts.

Reason: They vary by printer and platform.

```
e.g.   Use:
        IF X   NOT=  Y;
       or
        IF X  NE  Y;

       instead of:
        IF X  ^=  Y;
```

DOCUMENTATION

Clear and descriptive internal program documentation is essential. It allows any programmer to read the program and understand what is being done. It minimizes confusion and saves enormous time in maintenance.

Program Header

A program header should appear at the beginning of every program. Here is an example:

```
***************************************************************
* PROGRAM NAME   :                                            *
* DESCRIPTION    :                                            *
* CALLED BY      :                                            *
* CALLS TO       :                                            *
* PROGRAMMER     :                                            *
* DATE WRITTEN   :                                            *
***************************************************************
* INPUT FILES    :                                            *
* OUTPUT FILES   :                                            *
***************************************************************
* MODIFICATIONS  :                                            *
* -------------- :                                            *
* DATE           :                                            *
* CHANGE #       :                                            *
* PROGRAMMER     :                                            *
* DESCRIPTION    :                                            *
***************************************************************
```

Dataset Labels

Use a label when saving a permanent SAS dataset.

```
e.g. DATA CUSTLIB.F123(Label="File 123 of Customer
       Data for FY2000");
```

Variable Labels

Store variable labels for all permanent SAS datasets.

```
e.g. DATA CUSTLIB.F123(Label="File 123 of Customer
       Data for FY2000");
       SET X;
       LABEL   VAR1='Variable 1'
               VAR2='Variable 2' ;
```

Comments

- Include a comment before each major DATA step and before each major PROC step.
- If you are doing something tricky, explain it.
- If you are doing something non-standard, explain why.
- Identify the activity being performed.
- Write as if someone new were reading the code for the first time.

Run Instructions

Include special instructions for use, if necessary, in the program documentation or in a separate Runbook.

e.g. Note the run time of the program.
Note any anticipated problems such as space or memory problems.

Macro Parameters

Use general but self-documenting names for macro parameters.

e.g. %macro whatever(dsn=,startdate=);

DATA= on PROC Statement

Always use DATA= on PROC statements.

Reason: It makes programs easy to follow.
It ensures correct dataset referencing.
It provides internal documentation.

e.g.
```
PROC SORT DATA=X;
PROC PRINT DATA=X;
PROC CONTENTS DATA=X;
```

Intermediate Output

When creating intermediate output, always use a title statement so that you know what you are looking at in the output file.

e.g.
```
PROC PRINT DATA=origdata;
TITLE ‘Original Data’;
RUN;

PROC CONTENTS DATA=trandata;
TITLE ‘Transposed Data’;
RUN;
```

APPEARANCE

A program should look neat and organized. This is not just obsessive–compulsive behavior on the part of intense programmers. It makes the code easy to follow and understand. Appearance is very much a matter of personal style but the important thing is to adopt a style and use it consistently. Some specific suggestions follow.

Single Statement per Line

Use one statement per line.

Reason: It is easier to see.
It is easier to change.
It is easier to comment out lines for testing.

Blank Lines

- Skip a line before every DATA and PROC statement.
- Separate blocks of code to make the program readable.

e.g.
```
DATA X;
    statements;
RUN;

PROC procname;
    statements;
RUN;
```

Alignment and Indentation

- Left-justify DATA, PROC, OPTIONS statements. Indent all statements within.
- Indent statements within a DO loop. Align END with DO.
- Indent statements within an IF block. Align END with IF and ELSE.

Reason: This ‘inverted C’ look is the best (and sometimes the only) way to follow the logic of the nested loops. It also clearly indicates that there is a matching END statement for each DO and IF statement.

e.g.
```
DO something;
    DO more;
        IF X=1 then DO;
            statements;
        END;
        ELSE IF X  NE 1 then DO;
            statements;
        END;
    END;
END;
```

Placement of Statements

Use a standard sequence for placing statements and group like statements together to make them easy to find.

Within a program:

- OPTIONS statement first (unless the options must change later).
- %LET statements
- FILENAME, LIBNAME statements.
- PROC FORMAT
- Macro definitions
- Input steps
- Calculations
- Output last

Within a DATA step:

- All non-executable statements first (e.g. LENGTH, KEEP).
- All executable statements next.

Date Formats

Use date formats when reporting SAS dates.

Reason: They are easier to interpret than SAS date values.

```
e.g.  PROC PRINT DATA=X;
      FORMAT TRANSDAT DATE9.;
```

INPUT Statement

Use @col VARNAME for input when possible. For all types of input, use a separate line for each variable.

Reason: It is easier to see.
It is easier to change.
It is easier to comment out lines for testing.

```
e.g.  INPUT   @1   A1   $3.
              @4   A2   5.3
              ;

      INPUT   A1 $
              A2
              A3 $
              ;
```

RUN Statement

Use RUN; after each DATA and PROC step.

Reason: It clearly ends blocks of code. Also the SAS log will show comments and notes with the corresponding step.

```
e.g. PROC SORT DATA=X;
          BY YEAR;
     RUN;

     PROC SORT DATA=Y;
          BY YEAR;
     RUN;
```

Upper/Lower Case

Use upper/lower case for labels and comments. Yes, you can do this on a mainframe too.

Reason: It looks nice.

EFFICIENCY

Efficiencies are not required and inefficient code will still run. But some efficiency techniques have become so basic that they are considered standard practice and those are included here.

DROP and KEEP

- When inputting a flat file, input only the variables needed.
- When inputting a SAS dataset, use a KEEP statement to keep only the variables needed. (Note: DROP will work, but KEEP provides good documentation.)

  e.g. DATA X(KEEP=A1 A2 A3);

- DROP intermediate variables used for calculations.

```
e.g.  DATA X;
       DROP I J TEMPVAR;
           DO I= 1 to 3;
                DO J=1 to 5;
                     TEMPVAR = I;
                     NEWVAL=TEMPVAR * J;
                END;
            END:
      RUN;
```

- When outputting a dataset, KEEP only the variables needed.

```
e.g. DATA X(KEEP=A1 A2 A3 NEWVAR1
         NEWVAR2);
```

WHERE and IF

When subsetting a SAS dataset use WHERE rather than IF, if possible.

Reason: WHERE subsets the data before entering it into the Program Data Vector. IF subsets the data after inputting the entire dataset.

e.g. Use:
```
DATA NEW;
     SET OLD(WHERE=(YEAR GT  1995));
```

instead of:
```
DATA NEW;
     SET OLD;
     IF YEAR  GT 1995;
```

IF/THEN/ELSE

When stepping through an IF condition, check the most likely condition first.

e.g.
```
IF YEAR LT THISYR THEN OUTPUT OUTOLD;
     ELSE IF YEAR EQ THISYR THEN OUTPUT
                    OUTCUR;
     ELSE OUTPUT OUTBAD;
```

IF/ELSE instead of IF/IF

Use IF/ELSE for mutually exclusive conditions.

Reason: The ELSE/IF will check only those observations that fail the first IF condition. In the second part of the example, all observations will be checked twice.

e.g. Use:
```
IF  GENDER  EQ  ‘F’  THEN  OUTPUT  OUTF;
ELSE  IF  GENDER  EQ  ‘M’  THEN  OUTPUT
   OUTM;
```

instead of:
```
IF  GENDER  EQ  ‘F’  THEN  OUTPUT  OUTF;
IF  GENDER  EQ  ‘M’  THEN  OUTPUT  OUTM;
```

IF/ELSE Conditions

IF/ELSE should check for all conditions.

Reason: This allows you to identify and capture bad data. For complex IF conditions, it ensures that you are capturing all possible conditions.

e.g.
```
IF ANSWER = ‘Y’ THEN OUTPUT OUTY;
   ELSE IF ANSWER = ‘N’ THEN OUTPUT OUTN;
   ELSE OUTPUT OUTDK;
```

Sort

- Sort only the variables needed.

  Reason: It is faster.

  e.g. PROC SORT DATA=X(KEEP=A B C);

Note: There are many ways to optimize sorting but they depend on the size and the configuration of your system. Systems with large amounts of memory can sort in memory and not require as much temporary I/O space. You should experiment to find the best way to sort on your system.

Categorical Variables

Use character values for categorical variables or for flags instead of numeric values.

Reason: It saves space. A character ‘1’ uses one byte; a numeric 1 uses eight bytes.

e.g. Use:
```
AFLAG=’1’
```

instead of:
```
AFLAG=1;
```

MAINTAINABILITY

The most important reason for making a program clear and understandable is to make it maintainable. These guidelines will allow a program to be generalized so that it can be used more than once. A program is also easier to maintain if the log is easy to follow. These guidelines also explain how to make the log clean and prevent extraneous messages.

Use of Constants

Define constants within a %LET statement. Do not hard-code values within the program.

e.g. Use:
```
%LET STARTYR = 2000;
%LET ENDYR = 2020;
DO I=&STARTYR TO &ENDYR;
```

instead of:
```
DO I = 2000 TO 2020;
```

Nested Calls

Use no more than 2 layers of nested macro calls.

Reason: It is too easy to get lost after 2 calls.

Output

Do not create permanent datasets scattered within the program. Create them all at the end of the program.

Reason: If others run the program for testing, they may not know where all the output occurs and may overwrite the permanent data.

```
e.g. DATA X;
        statements;
     RUN;

     more program statements;

     * --- Output ---*;
     DATA PERMLIB.X;
        SET X;
     RUN;
     *---End of Program ---*;
```

Clarity

Notes/Warnings -- Avoid unnecessary notes or warning messages in the log. Even though they are not ERRORS, they can often lead to ambiguities, confusion, or actual errors.

- Avoid uninitialized variables. This might mean that a variable you think is in the dataset is not, or you have spelled the variable name wrong. Either way, this is an error.

  e.g. Avoid the message:
  NOTE: Variable XX is uninitialized.

- Avoid automatic numeric/character conversions; use PUT/INPUT to have the program do the work.

  e.g. Avoid the message:
  NOTE: Character values have been converted to numeric values at the places given by.

- Avoid automatic formatting; this can sometimes cause loss of data. Fix the program to use the correct format.

  e.g. Avoid the message:
  NOTE: At least one W.D format was too small for the number to be printed.
  The decimal may be shifted by the "BEST" format.

- Avoid excessive repetition of error messages.

  e.g. Use OPTIONS ERRORS=2;

  But do not use ERRORS=0; i.e. do not erase all error messages. You need to see the messages if the errors are there.

- Use OPTIONS NOOVP;

  This will eliminate the triple error messages.

Exception Handling

Check for violations of correct conditions.

e.g.. Division by zero

Use:
```
    IF B NE 0  THEN  X  =  A/B;
    ELSE X = .;
```
instead of:
```
    X = A/B;
```

Unambiguous Merging

- Always use a BY statement.
- Never have the same variables on more than one dataset (except the BY variables).
- Do not merge more than 2 datasets at a time.
- Do not allow the message:
  NOTE: MERGE statement has more than one
  data set with repeats of BY values.
  Consider this an error message.

Program Flow

If there are several programs in a system, create a main program to call each one rather than having each program call the next.

Reason: Others can read the main program and see the big picture. Otherwise they would have to read each program to get an idea of the entire system. Depending on the platform, the main program could be a shell script or a JCL stream as well as a SAS program.

```
e.g.   %include proga
       %include progb
       %include progc
```

If repeated routines are necessary, use a macro.

Use of Macros

Use macros if:
- The routine is used more than once.
- The routine depends on a value of a variable.
- The routine requires programming logic that cannot be included in a DATA step.

Use macro libraries (with the AUTOCALL facility) if:
- The routine is used by more than one program
- The routine changes often.

CONCLUSION

These guidelines are meant to be suggestions and reminders of things to consider when developing a system. They are by no means all-inclusive. Also, they are not meant to eliminate personal style from programming. But keep in mind that programs will be read and used by others and it helps to write so that we can communicate with each other.

CONTACT INFORMATION

The author may be contacted at:
LoisL@erols.com