Paper 123-31

SAS® Programming Guidelines

Lois Levin, Independent Consultant, Bethesda, Maryland

ABSTRACT

This paper presents a set of programming guidelines and conventions 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

Guidelines 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. When implemented by an organization, they represent an agreement to write in a certain way by all the programmers in a group.

Purpose of Programming Guidelines

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 of Programming Guidelines

Guidelines will be defined for the following areas:

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

****

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.

Example:

```
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.

Example:

```
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. Some specific naming guidelines follow.

Truncated Names

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

Example:

Use:

```
TRANSDAT
```

instead of:

```
TRANSDTE
```

IN= Names

When using the IN= dataset option, **name each IN variable using IN plus the first letter of the dataset name.**

Reason: It will distinguish them from regular variable names.

Example:

```
MERGE A(IN=INA) B(IN=INB);
IF INA AND INB;
```

Array Names

Begin array names with an underscore.

Reason: It will distinguish them from regular variable names.

Example: `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.**

Example:
```
FORMAT PROPTYPE PROPFMT.;
```

or **name the format with a description plus FMT.**

Example:
```
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.

Reason: This allows them to be defined outside the main SAS program. They can be changed without changing the program and they can be shared with other programs.

Example:
```
%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, the environment variables will be DD names. The JCL in a separate module may be shared by other programs.

Log and Output

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

Reason: This will keep similar files together. It will keep source code separate from output.

Example:
```
PROC PRINTTO LOG   = "&LOGPATH/ABC1.LOG";
PROC PRINTTO PRINT = "&OUTPATH/ABC1.LST"  NEW;
```

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.

Example:

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

Place a program header at the beginning of every program.

Reason: This is the introduction to the program. It provides an overview of the purpose of the program and its position within a system of programs. It provides a history of its development.

Example:

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

Dataset Labels

Use a dataset label when saving a permanent SAS dataset.

Reason: When others use the data, they will know what it is.

Example: DATA CUSTLIB.F123(Label="File 123 of Customer Data for FY2005");

Variable Labels

Use variable labels when saving a permanent SAS dataset.

Reason: When others use the data, they will know what the variables are.

Example:

```
DATA CUSTLIB.F123(Label="File 123 of Customer Data for FY2005");
        SET X;
        LABEL  VAR1='Variable 1'
               VAR2='Variable 2' ;
RUN;
```

Comments

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

Reason: Comments are essential to explain what is happening in the program, to describe the flow of the logic, or just to explain to another programmer (and sometimes, yourself) what you were thinking at the time. Comments can explain the relationship of the business logic to the program logic. Comments are important if someone else is to take over your code, if you are to return to the code after an interval, and for audit purposes.

Run Instructions

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

Reason: It provides guidance to others who may be running your program, especially non-programmers.

Example:

Note the elapsed time required for the program to run.
Note any anticipated problems such as space, memory, or contention problems.
Note any changes that need to be made for execution.

DATA= on PROC Statement

Always use DATA= on PROC statements.

Reasons:

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

Example:

```
PROC SORT DATA=X;
PROC PRINT DATA=X;
PROC CONTENTS DATA=X;
```

Intermediate Output

When creating intermediate output, always use a title statement.

Reason: It keeps track of what you are looking at in the output file.

Example:

```
PROC PRINT DATA=ORIGDATA;
TITLE 'Original Data';
RUN;

PROC PRINT 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.

Reasons:

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**

Reason: It makes the program readable. It clearly distinguishes different operations.

Example:

```
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 nested loops. It also clearly indicates that there is a matching END statement for each DO and IF statement.

Example:

```
DO something;
          DO more;
                    IF X=1 THEN DO;
                        statements;
                    END;
                    ELSE IF X  NE 1 THEN DO;
                        statements;
                    END;
          END;
END;
```

If the code extends over many lines, **add comment labels to the END statements.**

Example:

```
DO I=1 TO 10;
          DO J=1 TO 50;
                    DO K=1 TO 100;
                         statements;
                    END;              /*  end of  K  loop */
          END;              /*  end of  J loop  */
END;              /*  end of  I loop  */
```

Placement of Statements

Use a standard sequence for placing statements and group like statements together.

Reason: Placing all definitions at the top of the program establishes the environment. Placing all format definitions and macro definitions together makes them easy to find even though they may be referenced throughout the program. If parameters are defined with a %LET statement, group them together at the beginning to ensure that they will all be updated correctly. Similarly, FILENAME and LIBNAME statements should be at the top because they also may change.

Example:

- 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. RETAIN, 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. They will ensure that you have the date that you think you have.

Example:

```
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.

Reasons:

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

Example:

```
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.

Example:

```
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.)

  Example:

  ```
  DATA X;
    SET Y(KEEP=A1 A2 A3);
  ```

- **DROP intermediate variables used for calculations.**

  Example:

  ```
  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.**

  Example: `DATA X(KEEP=A1 A2 A3 NEWVAL);`

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.

Example:

Use:

```
DATA NEW;
      SET OLD(WHERE=(YEAR GT  1995));
```

instead of:

```
DATA NEW;
      SET OLD;
      IF YEAR  GT 1995;
```

IF/ELSE

Use IF/ELSE instead of IF/IF for mutually exclusive conditions.

Reason: The ELSE IF statement will check only those observations that fail the first IF condition. The second part of the example uses 2 IF statements and causes all the observations to be checked twice.

Example:

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 Sequence

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

Reason: The first ELSE condition will check only those records that fail the IF condition. If most records satisfy the IF condition, then the second statement will be executed a minimum number of times.

Example:

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

IF/ELSE Conditions

Use ELSE statements to check for all possible conditions.

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

Example:

```
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.

Example:

```
PROC SORT DATA=X(KEEP=A B C)
              OUT=XSORTED;
```

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 the data entirely in memory and not require as much temporary I/O space. In Version 9, you may be able to use multi-threading for sorting. You should experiment to find the best way to sort on your system.

Categorical Variables

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

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

Example:

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.

Reason: It is too easy to forget to change the values when you need to run later. Place all of these constant value assignments early in the code. It makes it easy to find and change them.

Example:

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: Placing output statements last may seem obvious but sometimes data are output as they are created in the program. If a new programmer takes over the program to modify it and tries to test it, he/she may write over data because it was not known where all the output is. So by placing all output statements at the end, a new programmer can comment them all out and not risk overwriting data.

Example:

```
DATA X;
  statements;
RUN;

more statements;

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

Clarity

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

- **Avoid uninitialized variables.**

  Reason: 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. It could result in incorrect or missing data.

  Example: Avoid the message:
  NOTE: Variable XX is uninitialized.

- **Avoid automatic numeric/character conversions; use PUT/INPUT to control the conversion yourself.**

  Reason: If you control the conversion, you know exactly what is happening and what variables you are working with. Otherwise the system may make changes that you have not planned.

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

- **Avoid automatic formatting;**

  Reason: This can sometimes cause loss of data. Fix the program to use the correct format.

  Example: 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.**

  Reason: It can make the log very long and messy. If you have an error you can usually see it in the first one or two examples. You usually do not need to see 20 examples of the same error. But beware -- do not use ERRORS=0; i.e. do not erase all error messages. You need to see the messages if the errors are there.

  Example: OPTIONS ERRORS=2;

- **Use OPTIONS NOOVP;**

  Reason: This will eliminate the triple (overprinted) error messages.

Exception Handling

Check for violations of correct conditions.

Reason: It will avoid the error message if the condition is violated. It will allow you to track the occurrences of violations.

Example:

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.

Reason: These rules will ensure that the observations are matched correctly and that no data gets lost or gets created incorrectly.

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 and the overall design. 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.

Example:

```
*** Main Program ***;

*--- Run the input program ---;
%INCLUDE PROGA;

*--- Run the calculation program ---;
%INCLUDE PROGB;

*--- Run the output program ---;
%INCLUDE PROGC;
```

If repeated routines are necessary, use macros.

MACROS

Writing code within macros should follow all other coding guidelines. But macros are really another language and so there are guidelines that pertain to their unique structure.

When to use a macro

Use a macro if:

- **The routine is used more than once.**
- **The routine depends on a value of a variable or parameter.**
- **The routine requires programming logic that cannot be included in a DATA step.**

%INCLUDE vs macro

Use %INCLUDE if:

- **The code will be called only once.**

  Reason: If %INCLUDE is used, the code will be compiled at each execution of the call, but if a macro is called from an autocall library, it will be compiled once and the compiled version will be executed at each call.

- **The code is a fragment or a program with no parameters.**

  Reason: If parameters are needed, they will have to be defined as global parameters for the %INCLUDE code. In a macro, parameters can be defined as local and they are more efficient than global parameters.

- **The code is external to the program.**

  Reason: %INCLUDE code must be external. Macro code may be external and called from an autocall library or it may be internal to the program.

Internal vs external macros

If a macro is particular only to the program, then define it in the program, but if it will be reused or shared with other programs, then store it externally in an autocall library.

Reasons for using an autocall library:

- Macro can be easily found
- Macro can be shared
- Macro can be changed and all calling programs are up-to-date

Example:

```
//MACROLIB DD DSN=HIGH.LEVEL.QUAL.MACROLIB,
//            DISP=(SHR,KEEP,KEEP)
```

or

```
FILENAME MACROLIB "/PATH/NAME/MACROLIB/";
```

and

```
OPTIONS SASAUTOS=MACROLIB;
```

Macro Parameters

Use general but self-documenting names for macro parameters.

Example: `%MACRO CALCMAC(DSN=,STARTDATE=);`

Macro specification

Use the macro name on the MEND statement.

Reason: It provides internal documentation and makes it easier to keep track of multiple macro definitions.

Example:

Use

```
%MEND CALCMAC;
```

instead of

```
%MEND;
```

Positional parameters vs named (keyword) parameters

If you have one or two parameters, then it is OK to use positional parameters. But if there are more, then it is best to use named (keyword) parameters.

Reasons:

- Keyword parameters are optional and positional parameters are not.
- You can specify default values of the keyword parameters in the macro code and then call with only those you want to change.
- Order is not important for keyword parameters and obviously order is essential to positional parameters.
- New keyword parameters can be added later

Example:

- One positional parameter

  ```
  %MACRO CALCMAC(DSN);
  ```

- Several keyword parameters

  ```
  %MACRO CALCMAC(INDSN=, OUTDSN=, INVARS=, OUTVARS=);
  ```

Global vs local parameters

Use local macro parameters rather than global parameters, when possible.

Reasons:

- Avoids confusion in macro variable definition.
  If you define a local macro variable when there is already a global one defined with the same name, there can be confusion.

- Local macro variables use less storage.

  - Global macro variables exist for the duration of the SAS session and can be referenced anywhere in the program -- either inside or outside a macro. Local macro variables exist only during the execution of the macro in which the variables are created and cannot be referenced outside the defining macro.

- Macro variables are stored in symbol tables, which list the macro variable name and its value. There is a global symbol table, which stores all global macro variables. Local macro variables are stored in a local symbol table that is created at the beginning of the execution of a macro. Local macro variables exist only as long as a particular macro executes; when the macro stops executing, all local macro variables for that macro cease to exist.

Example:

Use:

```
 %MACRO PRINTMAC(DSN);
       PROC PRINT DATA=&DSN;
       RUN;
%MEND PRINTMAC;
```

instead of:

```
%LET DSN=DATA1;
%MACRO PRINTMAC;
       PROC PRINT DATA=&DSN;
       RUN;
%MEND PRINTMAC;
```

Macro Comments

Use macro comments %*

Reason:

A macro comment is not constant text and is not stored in a compiled macro, thereby saving space.

Other comments may be used in a macro. SAS comments of the form *comment; are stored as constant text in a compiled macro. SAS comments of the form /*comment*/ are not stored in a compiled macro.

Macro options

When you program with macros, use the options MPRINT, SYMBOLGEN, and MLOGIC.

Reason:

They allow you to see the macro statements being executed, the resolution of the macro variables and the execution of logical statements in the macro. These statements are extremely helpful for debugging macros.

You may want to remove these statements when they are no longer needed for debugging as they make the log very large. You might want to keep MPRINT in the option list during production runs to provide documentation and an audit trail for program execution.

Example:

```
OPTIONS MPRINT
        /*  SYMBOLGEN  MLOGIC  */
        ;
```

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:
Lois831@hotmail.com